Symfony2 Docs Documentation

Release 2

Sensio Labs

January 10, 2016

Contents

CHAPTER 1

Quick Tour

Get started fast with the Symfony2 Quick Tour:

1.1 Quick Tour

1.1.1 The Big Picture

Start using Symfony2 in 10 minutes! This chapter will walk you through some of the most important concepts behind Symfony2 and explain how you can get started quickly by showing you a simple project in action.

If you've used a web framework before, you should feel right at home with Symfony2. If not, welcome to a whole new way of developing web applications!

Tip: Want to learn why and when you need to use a framework? Read the "Symfony in 5 minutes" document.

Downloading Symfony2

First, check that you have installed and configured a Web server (such as Apache) with PHP 5.3.2 or higher.

Ready? Start by downloading the "Symfony2 Standard Edition", a Symfony distribution that is preconfigured for the most common use cases and also contains some code that demonstrates how to use Symfony2 (get the archive with the *vendors* included to get started even faster).

After unpacking the archive under your web server root directory, you should have a `Symfony/` directory that looks like this:

```
www/ <- your web root directory
    Symfony/ <- the unpacked archive
        app/
            cache/
            config/
            logs/
            Resources/
        bin/
        src/
            Acme/
                DemoBundle/
                    Controller/
                    Resources/
                    ...
```

```
        vendor/
            symfony/
            doctrine/
            ...
        web/
            app.php
            ...
```

Note: If you downloaded the Standard Edition *without vendors*, simply run the following command to download all of the vendor libraries:

```
php bin/vendors install
```

Checking the Configuration

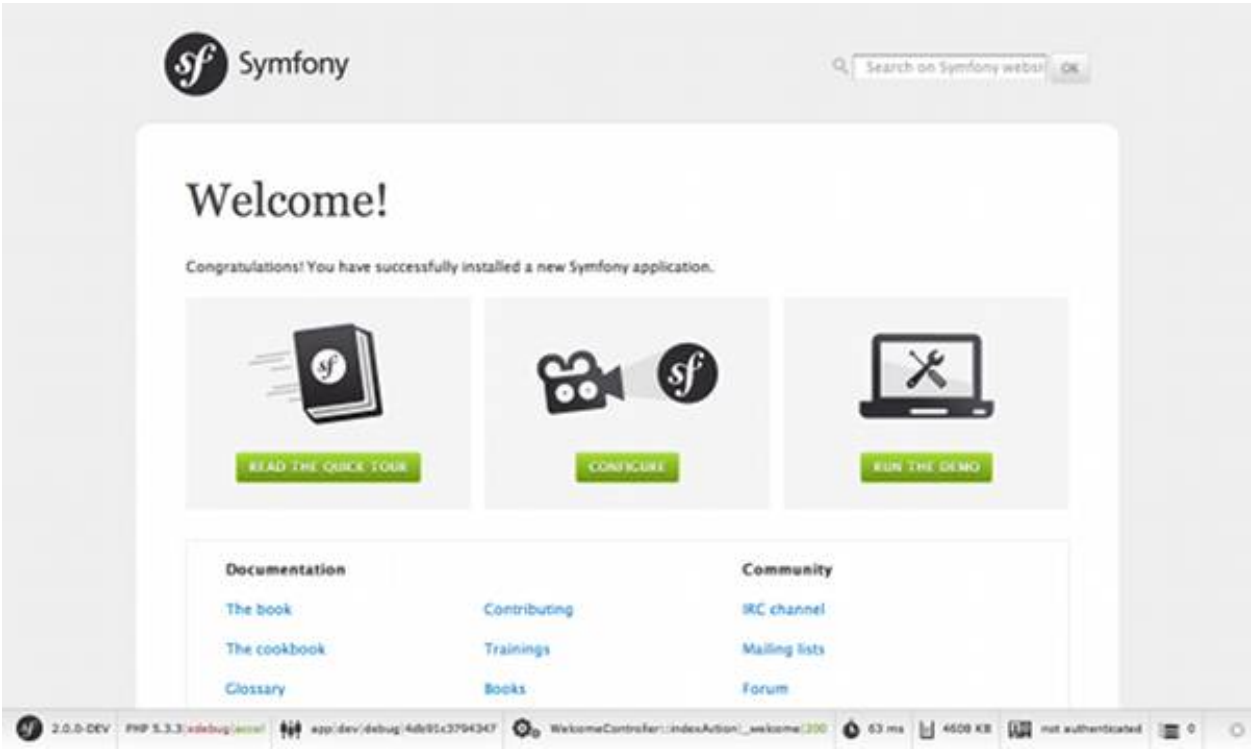
Symfony2 comes with a visual server configuration tester to help avoid some headaches that come from Web server or PHP misconfiguration. Use the following URL to see the diagnostics for your machine:

```
http://localhost/Symfony/web/config.php
```

If there are any outstanding issues listed, correct them. You might also tweak your configuration by following any given recommendations. When everything is fine, click on "*Bypass configuration and go to the Welcome page*" to request your first "real" Symfony2 webpage:

```
http://localhost/Symfony/web/app_dev.php/
```

Symfony2 should welcome and congratulate you for your hard work so far!



Understanding the Fundamentals

One of the main goals of a framework is to ensure Separation of Concerns. This keeps your code organized and allows your application to evolve easily over time by avoiding the mixing of database calls, HTML tags, and business logic in the same script. To achieve this goal with Symfony, you'll first need to learn a few fundamental concepts and terms.

Tip: Want proof that using a framework is better than mixing everything in the same script? Read the "Symfony2 versus Flat PHP" chapter of the book.

The distribution comes with some sample code that you can use to learn more about the main Symfony2 concepts. Go to the following URL to be greeted by Symfony2 (replace *Fabien* with your first name):

```
http://localhost/Symfony/web/app_dev.php/demo/hello/Fabien
```

What's going on here? Let's dissect the URL:

- `app_dev.php`: This is a front controller. It is the unique entry point of the application and it responds to all user requests;
- `/demo/hello/Fabien`: This is the *virtual path* to the resource the user wants to access.

Your responsibility as a developer is to write the code that maps the user's *request* (`/demo/hello/Fabien`) to the *resource* associated with it (the `Hello Fabien!` HTML page).

Routing

Symfony2 routes the request to the code that handles it by trying to match the requested URL against some configured patterns. By default, these patterns (called routes) are defined in the `app/config/routing.yml` configuration file. When you're in the `dev` *environment* - indicated by the app_**dev**.php front controller - the `app/config/routing_dev.yml` configuration file is also loaded. In the Standard Edition, the routes to these "demo" pages are placed in that file:

```
# app/config/routing_dev.yml
_welcome:
```

```
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Welcome:index }

_demo:
    resource: "@AcmeDemoBundle/Controller/DemoController.php"
    type:     annotation
    prefix:   /demo

# ...
```

The first three lines (after the comment) define the code that is executed when the user requests the "/" resource (i.e. the welcome page you saw earlier). When requested, the `AcmeDemoBundle:Welcome:index` controller will be executed. In the next section, you'll learn exactly what that means.

Tip: The Symfony2 Standard Edition uses YAML for its configuration files, but Symfony2 also supports XML, PHP, and annotations natively. The different formats are compatible and may be used interchangeably within an application. Also, the performance of your application does not depend on the configuration format you choose as everything is cached on the very first request.

Controllers

A controller is a fancy name for a PHP function or method that handles incoming *requests* and returns *responses* (often HTML code). Instead of using the PHP global variables and functions (like `$_GET` or `header()`) to manage these HTTP messages, Symfony uses objects: `Symfony\Component\HttpFoundation\Request` and `Symfony\Component\HttpFoundation\Response`. The simplest possible controller might create the response by hand, based on the request:

```
use Symfony\Component\HttpFoundation\Response;

$name = $request->query->get('name');

return new Response('Hello '.$name, 200, array('Content-Type' => 'text/plain'));
```

Note: Symfony2 embraces the HTTP Specification, which are the rules that govern all communication on the Web. Read the "Symfony2 and HTTP Fundamentals" chapter of the book to learn more about this and the added power that this brings.

Symfony2 chooses the controller based on the `_controller` value from the routing configuration: `AcmeDemoBundle:Welcome:index`. This string is the controller *logical name*, and it references the `indexAction` method from the `Acme\DemoBundle\Controller\WelcomeController` class:

```
// src/Acme/DemoBundle/Controller/WelcomeController.php
namespace Acme\DemoBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class WelcomeController extends Controller
{
    public function indexAction()
    {
        return $this->render('AcmeDemoBundle:Welcome:index.html.twig');
    }
}
```

Tip: You could have used the full class and method name - `Acme\DemoBundle\Controller\WelcomeController::indexA` - for the `_controller` value. But if you follow some simple conventions, the logical name is shorter and allows for more flexibility.

The `WelcomeController` class extends the built-in `Controller` class, which provides useful shortcut methods, like the **:method:'Symfony\\Bundle\\FrameworkBundle\\Controller\\Controller::render'** method that loads and renders a template (`AcmeDemoBundle:Welcome:index.html.twig`). The returned value is a Response object populated with the rendered content. So, if the needs arise, the Response can be tweaked before it is sent to the browser:

```
public function indexAction()
{
    $response = $this->render('AcmeDemoBundle:Welcome:index.txt.twig');
    $response->headers->set('Content-Type', 'text/plain');

    return $response;
}
```

No matter how you do it, the end goal of your controller is always to return the `Response` object that should be delivered back to the user. This `Response` object can be populated with HTML code, represent a client redirect, or even return the contents of a JPG image with a `Content-Type` header of `image/jpg`.

Tip: Extending the `Controller` base class is optional. As a matter of fact, a controller can be a plain PHP function or even a PHP closure. "The Controller" chapter of the book tells you everything about Symfony2 controllers.

The template name, `AcmeDemoBundle:Welcome:index.html.twig`, is the template *logical name* and it references the `Resources/views/Welcome/index.html.twig` file inside the `AcmeDemoBundle` (located at `src/Acme/DemoBundle`). The bundles section below will explain why this is useful.

Now, take a look at the routing configuration again and find the `_demo` key:

```
# app/config/routing_dev.yml
_demo:
    resource: "@AcmeDemoBundle/Controller/DemoController.php"
    type:     annotation
    prefix:   /demo
```

Symfony2 can read/import the routing information from different files written in YAML, XML, PHP, or even embedded in PHP annotations. Here, the file's *logical name* is `@AcmeDemoBundle/Controller/DemoController.php` and refers to the `src/Acme/DemoBundle/Controller/DemoController.php` file. In this file, routes are defined as annotations on action methods:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

class DemoController extends Controller
{
    /**
     * @Route("/hello/{name}", name="_demo_hello")
     * @Template()
     */
    public function helloAction($name)
    {
        return array('name' => $name);
    }
```

```
    // ...
}
```

The `@Route()` annotation defines a new route with a pattern of `/hello/{name}` that executes the `helloAction` method when matched. A string enclosed in curly brackets like `{name}` is called a placeholder. As you can see, its value can be retrieved through the `$name` method argument.

Note: Even if annotations are not natively supported by PHP, you use them extensively in Symfony2 as a convenient way to configure the framework behavior and keep the configuration next to the code.

If you take a closer look at the controller code, you can see that instead of rendering a template and returning a `Response` object like before, it just returns an array of parameters. The `@Template()` annotation tells Symfony to render the template for you, passing in each variable of the array to the template. The name of the template that's rendered follows the name of the controller. So, in this example, the `AcmeDemoBundle:Demo:hello.html.twig` template is rendered (located at `src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig`).

Tip: The `@Route()` and `@Template()` annotations are more powerful than the simple examples shown in this tutorial. Learn more about "annotations in controllers" in the official documentation.

Templates

The controller renders the `src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig` template (or `AcmeDemoBundle:Demo:hello.html.twig` if you use the logical name):

```
{# src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig #}
{% extends "AcmeDemoBundle::layout.html.twig" %}

{% block title "Hello " ~ name %}

{% block content %}
    <h1>Hello {{ name }}!</h1>
{% endblock %}
```

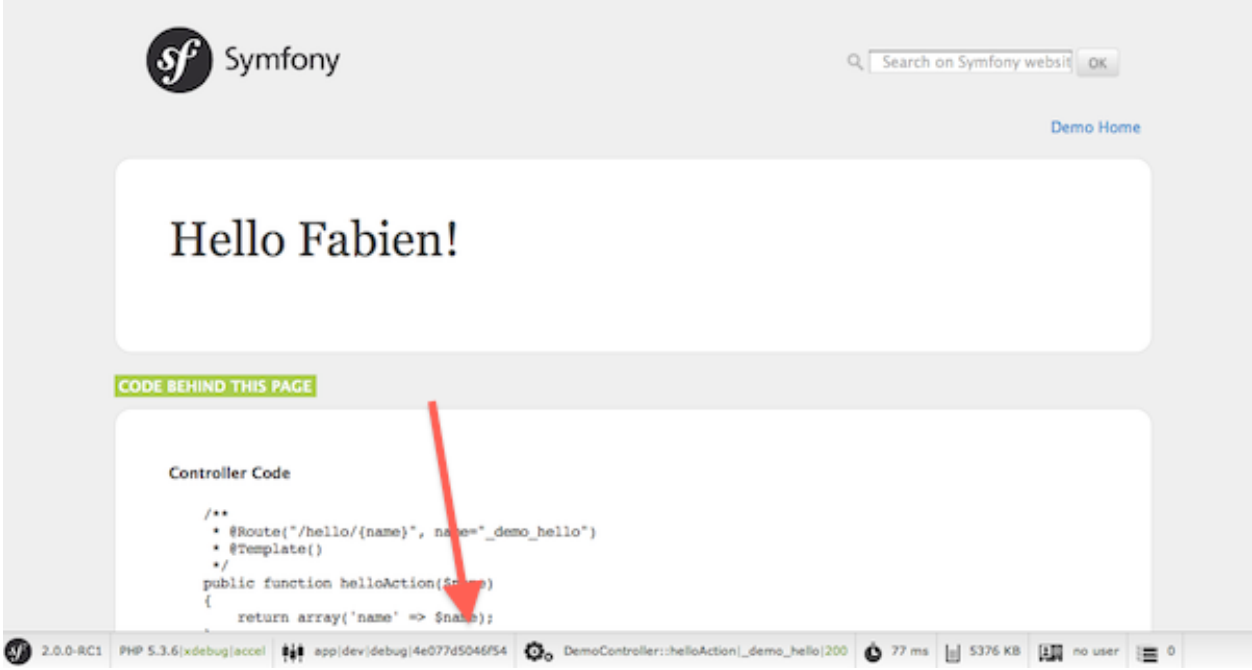
By default, Symfony2 uses Twig as its template engine but you can also use traditional PHP templates if you choose. The next chapter will introduce how templates work in Symfony2.

Bundles

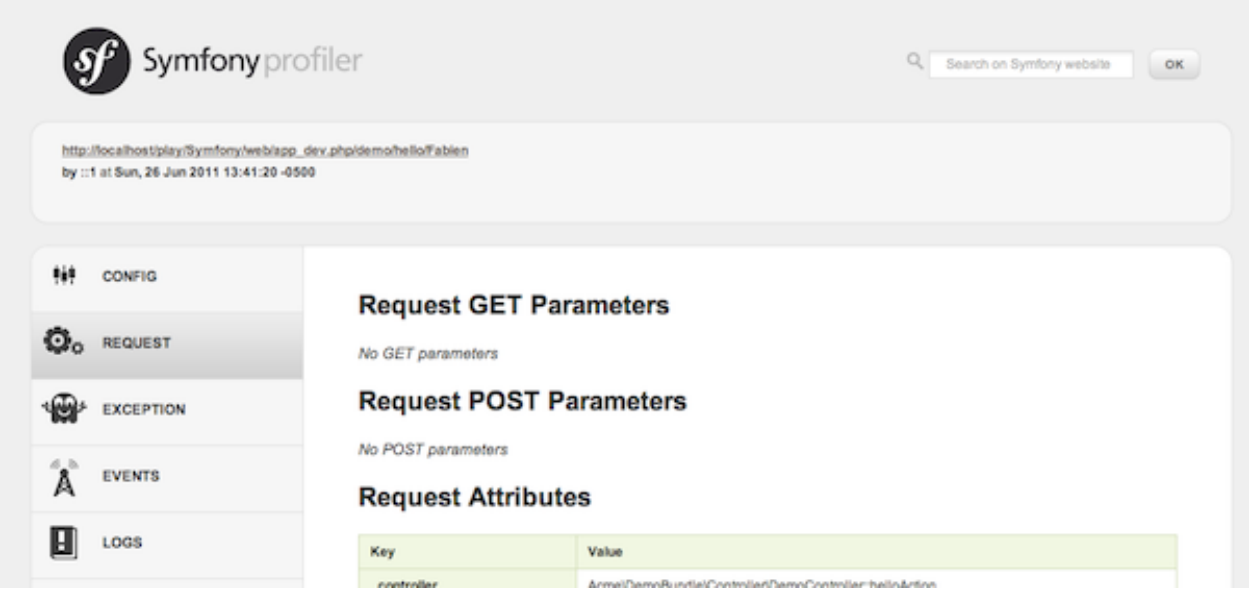
You might have wondered why the bundle word is used in many names we have seen so far. All the code you write for your application is organized in bundles. In Symfony2 speak, a bundle is a structured set of files (PHP files, stylesheets, JavaScripts, images, ...) that implements a single feature (a blog, a forum, ...) and which can be easily shared with other developers. As of now, we have manipulated one bundle, `AcmeDemoBundle`. You will learn more about bundles in the last chapter of this tutorial.

Working with Environments

Now that you have a better understanding of how Symfony2 works, take a closer look at the bottom of any Symfony2 rendered page. You should notice a small bar with the Symfony2 logo. This is called the "Web Debug Toolbar" and it is the developer's best friend.



But what you see initially is only the tip of the iceberg; click on the weird hexadecimal number to reveal yet another very useful Symfony2 debugging tool: the profiler.



Of course, you won't want to show these tools when you deploy your application to production. That's why you will find another front controller in the `web/` directory (`app.php`), which is optimized for the production environment:

```
http://localhost/Symfony/web/app.php/demo/hello/Fabien
```

And if you use Apache with `mod_rewrite` enabled, you can even omit the `app.php` part of the URL:

```
http://localhost/Symfony/web/demo/hello/Fabien
```

Last but not least, on the production servers, you should point your web root directory to the `web/` directory to secure your installation and have an even better looking URL:

```
http://localhost/demo/hello/Fabien
```

Note: Note that the three URLs above are provided here only as **examples** of how a URL looks like when the production front controller is used (with or without mod_rewrite). If you actually try them in an out of the box installation of *Symfony Standard Edition* you will get a 404 error as *AcmeDemoBundle* is enabled only in dev environment and its routes imported in *app/config/routing_dev.yml*.

To make you application respond faster, Symfony2 maintains a cache under the `app/cache/` directory. In the development environment (`app_dev.php`), this cache is flushed automatically whenever you make changes to any code or configuration. But that's not the case in the production environment (`app.php`) where performance is key. That's why you should always use the development environment when developing your application.

Different environments of a given application differ only in their configuration. In fact, a configuration can inherit from another one:

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

web_profiler:
    toolbar: true
    intercept_redirects: false
```

The `dev` environment (which loads the `config_dev.yml` configuration file) imports the global `config.yml` file and then modifies it by, in this example, enabling the web debug toolbar.

Final Thoughts

Congratulations! You've had your first taste of Symfony2 code. That wasn't so hard, was it? There's a lot more to explore, but you should already see how Symfony2 makes it really easy to implement web sites better and faster. If you are eager to learn more about Symfony2, dive into the next section: "The View".

1.1.2 The View

After reading the first part of this tutorial, you have decided that Symfony2 was worth another 10 minutes. Great choice! In this second part, you will learn more about the Symfony2 template engine, Twig. Twig is a flexible, fast, and secure template engine for PHP. It makes your templates more readable and concise; it also makes them more friendly for web designers.

Note: Instead of Twig, you can also use PHP for your templates. Both template engines are supported by Symfony2.

Getting familiar with Twig

Tip: If you want to learn Twig, we highly recommend you to read its official documentation. This section is just a quick overview of the main concepts.

A Twig template is a text file that can generate any type of content (HTML, XML, CSV, LaTeX, ...). Twig defines two kinds of delimiters:

- `{{ ... }}`: Prints a variable or the result of an expression;

- `{% ... %}`: Controls the logic of the template; it is used to execute `for` loops and `if` statements, for example.

Below is a minimal template that illustrates a few basics, using two variables `page_title` and `navigation`, which would be passed into the template:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Webpage</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Tip: Comments can be included inside templates using the `{# ... #}` delimiter.

To render a template in Symfony, use the `render` method from within a controller and pass it any variables needed in the template:

```
$this->render('AcmeDemoBundle:Demo:hello.html.twig', array(
    'name' => $name,
));
```

Variables passed to a template can be strings, arrays, or even objects. Twig abstracts the difference between them and lets you access "attributes" of a variable with the dot (`.`) notation:

```
{# array('name' => 'Fabien') #}
{{ name }}

{# array('user' => array('name' => 'Fabien')) #}
{{ user.name }}

{# force array lookup #}
{{ user['name'] }}

{# array('user' => new User('Fabien')) #}
{{ user.name }}
{{ user.getName }}

{# force method name lookup #}
{{ user.name() }}
{{ user.getName() }}

{# pass arguments to a method #}
{{ user.date('Y-m-d') }}
```

Note: It's important to know that the curly braces are not part of the variable but the print statement. If you access variables inside tags don't put the braces around.

Decorating Templates

More often than not, templates in a project share common elements, like the well-known header and footer. In Symfony2, we like to think about this problem differently: a template can be decorated by another one. This works exactly the same as PHP classes: template inheritance allows you to build a base "layout" template that contains all the common elements of your site and defines "blocks" that child templates can override.

The `hello.html.twig` template inherits from `layout.html.twig`, thanks to the `extends` tag:

```
{# src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig #}
{% extends "AcmeDemoBundle::layout.html.twig" %}

{% block title "Hello " ~ name %}

{% block content %}
    <h1>Hello {{ name }}!</h1>
{% endblock %}
```

The `AcmeDemoBundle::layout.html.twig` notation sounds familiar, doesn't it? It is the same notation used to reference a regular template. The `::` part simply means that the controller element is empty, so the corresponding file is directly stored under the `Resources/views/` directory.

Now, let's have a look at a simplified `layout.html.twig`:

```
{# src/Acme/DemoBundle/Resources/views/layout.html.twig #}
<div class="symfony-content">
    {% block content %}
    {% endblock %}
</div>
```

The `{% block %}` tags define blocks that child templates can fill in. All the block tag does is to tell the template engine that a child template may override those portions of the template.

In this example, the `hello.html.twig` template overrides the `content` block, meaning that the "Hello Fabien" text is rendered inside the `div.symfony-content` element.

Using Tags, Filters, and Functions

One of the best feature of Twig is its extensibility via tags, filters, and functions. Symfony2 comes bundled with many of these built-in to ease the work of the template designer.

Including other Templates

The best way to share a snippet of code between several distinct templates is to create a new template that can then be included from other templates.

Create an `embedded.html.twig` template:

```
{# src/Acme/DemoBundle/Resources/views/Demo/embedded.html.twig #}
Hello {{ name }}
```

And change the `index.html.twig` template to include it:

```
{# src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig #}
{% extends "AcmeDemoBundle::layout.html.twig" %}

{# override the body block from embedded.html.twig #}
{% block content %}
```

```
    {% include "AcmeDemoBundle:Demo:embedded.html.twig" %}
{% endblock %}
```

Embedding other Controllers

And what if you want to embed the result of another controller in a template? That's very useful when working with Ajax, or when the embedded template needs some variable not available in the main template.

Suppose you've created a `fancy` action, and you want to include it inside the `index` template. To do this, use the `render` tag:

```
{# src/Acme/DemoBundle/Resources/views/Demo/index.html.twig #}
{% render "AcmeDemoBundle:Demo:fancy" with { 'name': name, 'color': 'green' } %}
```

Here, the `AcmeDemoBundle:Demo:fancy` string refers to the `fancy` action of the `Demo` controller. The arguments (`name` and `color`) act like simulated request variables (as if the `fancyAction` were handling a whole new request) and are made available to the controller:

```
// src/Acme/DemoBundle/Controller/DemoController.php

class DemoController extends Controller
{
    public function fancyAction($name, $color)
    {
        // create some object, based on the $color variable
        $object = ...;

        return $this->render('AcmeDemoBundle:Demo:fancy.html.twig', array('name' => $name, 'object' =
    }

    // ...
}
```

Creating Links between Pages

Speaking of web applications, creating links between pages is a must. Instead of hardcoding URLs in templates, the `path` function knows how to generate URLs based on the routing configuration. That way, all your URLs can be easily updated by just changing the configuration:

```
<a href="{{ path('_demo_hello', { 'name': 'Thomas' }) }}">Greet Thomas!</a>
```

The `path` function takes the route name and an array of parameters as arguments. The route name is the main key under which routes are referenced and the parameters are the values of the placeholders defined in the route pattern:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hello/{name}", name="_demo_hello")
 * @Template()
 */
public function helloAction($name)
{
    return array('name' => $name);
}
```

Tip: The `url` function generates *absolute* URLs: `{{ url('_demo_hello', { 'name': 'Thomas' }) }}`.

Including Assets: images, JavaScripts, and stylesheets

What would the Internet be without images, JavaScripts, and stylesheets? Symfony2 provides the `asset` function to deal with them easily:

```
<link href="{{ asset('css/blog.css') }}" rel="stylesheet" type="text/css" />

<img src="{{ asset('images/logo.png') }}" />
```

The `asset` function's main purpose is to make your application more portable. Thanks to this function, you can move the application root directory anywhere under your web root directory without changing anything in your template's code.

Escaping Variables

Twig is configured to automatically escapes all output by default. Read Twig documentation to learn more about output escaping and the Escaper extension.

Final Thoughts

Twig is simple yet powerful. Thanks to layouts, blocks, templates and action inclusions, it is very easy to organize your templates in a logical and extensible way. However, if you're not comfortable with Twig, you can always use PHP templates inside Symfony without any issues.

You have only been working with Symfony2 for about 20 minutes, but you can already do pretty amazing stuff with it. That's the power of Symfony2. Learning the basics is easy, and you will soon learn that this simplicity is hidden under a very flexible architecture.

But I'm getting ahead of myself. First, you need to learn more about the controller and that's exactly the topic of the next part of this tutorial. Ready for another 10 minutes with Symfony2?

1.1.3 The Controller

Still with us after the first two parts? You are already becoming a Symfony2 addict! Without further ado, let's discover what controllers can do for you.

Using Formats

Nowadays, a web application should be able to deliver more than just HTML pages. From XML for RSS feeds or Web Services, to JSON for Ajax requests, there are plenty of different formats to choose from. Supporting those formats in Symfony2 is straightforward. Tweak the route by adding a default value of `xml` for the `_format` variable:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hello/{name}", defaults={"_format"="xml"}, name="_demo_hello")
```

```
     * @Template()
     */
    public function helloAction($name)
    {
        return array('name' => $name);
    }
```

By using the request format (as defined by the `_format` value), Symfony2 automatically selects the right template, here `hello.xml.twig`:

```
<!-- src/Acme/DemoBundle/Resources/views/Demo/hello.xml.twig -->
<hello>
    <name>{{ name }}</name>
</hello>
```

That's all there is to it. For standard formats, Symfony2 will also automatically choose the best `Content-Type` header for the response. If you want to support different formats for a single action, use the `{_format}` placeholder in the route pattern instead:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hello/{name}.{_format}", defaults={"_format"="html"}, requirements={"_format"="html|xml|j
 * @Template()
 */
public function helloAction($name)
{
    return array('name' => $name);
}
```

The controller will now be called for URLs like `/demo/hello/Fabien.xml` or `/demo/hello/Fabien.json`.

The `requirements` entry defines regular expressions that placeholders must match. In this example, if you try to request the `/demo/hello/Fabien.js` resource, you will get a 404 HTTP error, as it does not match the `_format` requirement.

Redirecting and Forwarding

If you want to redirect the user to another page, use the `redirect()` method:

```
return $this->redirect($this->generateUrl('_demo_hello', array('name' => 'Lucas')));
```

The `generateUrl()` is the same method as the `path()` function we used in templates. It takes the route name and an array of parameters as arguments and returns the associated friendly URL.

You can also easily forward the action to another one with the `forward()` method. Internally, Symfony makes a "sub-request", and returns the `Response` object from that sub-request:

```
$response = $this->forward('AcmeDemoBundle:Hello:fancy', array('name' => $name, 'color' => 'green'));

// do something with the response or return it directly
```

Getting information from the Request

Besides the values of the routing placeholders, the controller also has access to the `Request` object:

```
$request = $this->getRequest();

$request->isXmlHttpRequest(); // is it an Ajax request?

$request->getPreferredLanguage(array('en', 'fr'));

$request->query->get('page'); // get a $_GET parameter

$request->request->get('page'); // get a $_POST parameter
```

In a template, you can also access the `Request` object via the `app.request` variable:

```
{{ app.request.query.get('page') }}

{{ app.request.parameter('page') }}
```

Persisting Data in the Session

Even if the HTTP protocol is stateless, Symfony2 provides a nice session object that represents the client (be it a real person using a browser, a bot, or a web service). Between two requests, Symfony2 stores the attributes in a cookie by using native PHP sessions.

Storing and retrieving information from the session can be easily achieved from any controller:

```
$session = $this->getRequest()->getSession();

// store an attribute for reuse during a later user request
$session->set('foo', 'bar');

// in another controller for another request
$foo = $session->get('foo');

// use a default value of the key doesn't exist
$filters = $session->set('filters', array());
```

You can also store small messages that will only be available for the very next request:

```
// store a message for the very next request (in a controller)
$session->setFlash('notice', 'Congratulations, your action succeeded!');

// display the message back in the next request (in a template)
{{ app.session.flash('notice') }}
```

This is useful when you need to set a success message before redirecting the user to another page (which will then show the message).

Securing Resources

The Symfony Standard Edition comes with a simple security configuration that fits most common needs:

```
# app/config/security.yml
security:
    encoders:
```

```
        Symfony\Component\Security\Core\User\User: plaintext

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]

    providers:
        in_memory:
            memory:
                users:
                    user:  { password: userpass, roles: [ 'ROLE_USER' ] }
                    admin: { password: adminpass, roles: [ 'ROLE_ADMIN' ] }

    firewalls:
        dev:
            pattern:  ^/(_(profiler|wdt)|css|images|js)/
            security: false

        login:
            pattern:  ^/demo/secured/login$
            security: false

        secured_area:
            pattern:    ^/demo/secured/
            form_login:
                check_path: /demo/secured/login_check
                login_path: /demo/secured/login
            logout:
                path:   /demo/secured/logout
                target: /demo/
```

This configuration requires users to log in for any URL starting with `/demo/secured/` and defines two valid users: `user` and `admin`. Moreover, the `admin` user has a `ROLE_ADMIN` role, which includes the `ROLE_USER` role as well (see the `role_hierarchy` setting).

Tip: For readability, passwords are stored in clear text in this simple configuration, but you can use any hashing algorithm by tweaking the `encoders` section.

Going to the `http://localhost/Symfony/web/app_dev.php/demo/secured/hello` URL will automatically redirect you to the login form because this resource is protected by a `firewall`.

You can also force the action to require a given role by using the `@Secure` annotation on the controller:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Route("/hello/admin/{name}", name="_demo_secured_hello_admin")
 * @Secure(roles="ROLE_ADMIN")
 * @Template()
 */
public function helloAdminAction($name)
{
    return array('name' => $name);
}
```

Now, log in as `user` (who does *not* have the `ROLE_ADMIN` role) and from the secured hello page, click on the "Hello

resource secured" link. Symfony2 should return a 403 HTTP status code, indicating that the user is "forbidden" from accessing that resource.

Note: The Symfony2 security layer is very flexible and comes with many different user providers (like one for the Doctrine ORM) and authentication providers (like HTTP basic, HTTP digest, or X509 certificates). Read the "Security" chapter of the book for more information on how to use and configure them.

Caching Resources

As soon as your website starts to generate more traffic, you will want to avoid generating the same resource again and again. Symfony2 uses HTTP cache headers to manage resources cache. For simple caching strategies, use the convenient `@Cache()` annotation:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;

/**
 * @Route("/hello/{name}", name="_demo_hello")
 * @Template()
 * @Cache(maxage="86400")
 */
public function helloAction($name)
{
    return array('name' => $name);
}
```

In this example, the resource will be cached for a day. But you can also use validation instead of expiration or a combination of both if that fits your needs better.

Resource caching is managed by the Symfony2 built-in reverse proxy. But because caching is managed using regular HTTP cache headers, you can replace the built-in reverse proxy with Varnish or Squid and easily scale your application.

Note: But what if you cannot cache whole pages? Symfony2 still has the solution via Edge Side Includes (ESI), which are supported natively. Learn more by reading the "HTTP Cache" chapter of the book.

Final Thoughts

That's all there is to it, and I'm not even sure we have spent the full 10 minutes. We briefly introduced bundles in the first part, and all the features we've learned about so far are part of the core framework bundle. But thanks to bundles, everything in Symfony2 can be extended or replaced. That's the topic of the next part of this tutorial.

1.1.4 The Architecture

You are my hero! Who would have thought that you would still be here after the first three parts? Your efforts will be well rewarded soon. The first three parts didn't look too deeply at the architecture of the framework. Because it makes Symfony2 stand apart from the framework crowd, let's dive into the architecture now.

Understanding the Directory Structure

The directory structure of a Symfony2 application is rather flexible, but the directory structure of the *Standard Edition* distribution reflects the typical and recommended structure of a Symfony2 application:

- `app/`: The application configuration;
- `src/`: The project's PHP code;
- `vendor/`: The third-party dependencies;
- `web/`: The web root directory.

The `web/` Directory

The web root directory is the home of all public and static files like images, stylesheets, and JavaScript files. It is also where each front controller lives:

```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

The kernel first requires the `bootstrap.php.cache` file, which bootstraps the framework and registers the autoloader (see below).

Like any front controller, `app.php` uses a Kernel Class, `AppKernel`, to bootstrap the application.

The `app/` Directory

The `AppKernel` class is the main entry point of the application configuration and as such, it is stored in the `app/` directory.

This class must implement two methods:

- `registerBundles()` must return an array of all bundles needed to run the application;
- `registerContainerConfiguration()` loads the application configuration (more on this later).

PHP autoloading can be configured via `app/autoload.php`:

```
// app/autoload.php
use Symfony\Component\ClassLoader\UniversalClassLoader;

$loader = new UniversalClassLoader();
$loader->registerNamespaces(array(
    'Symfony'          => array(__DIR__.'/../vendor/symfony/src', __DIR__.'/../vendor/bundles'),
    'Sensio'           => __DIR__.'/../vendor/bundles',
    'JMS'              => __DIR__.'/../vendor/bundles',
    'Doctrine\\Common' => __DIR__.'/../vendor/doctrine-common/lib',
    'Doctrine\\DBAL'   => __DIR__.'/../vendor/doctrine-dbal/lib',
    'Doctrine'         => __DIR__.'/../vendor/doctrine/lib',
    'Monolog'          => __DIR__.'/../vendor/monolog/src',
    'Assetic'          => __DIR__.'/../vendor/assetic/src',
    'Metadata'         => __DIR__.'/../vendor/metadata/src',
));
$loader->registerPrefixes(array(
    'Twig_Extensions_' => __DIR__.'/../vendor/twig-extensions/lib',
    'Twig_'            => __DIR__.'/../vendor/twig/lib',
```

```
));

// ...

$loader->registerNamespaceFallbacks(array(
    __DIR__.'/../src',
));
$loader->register();
```

The `Symfony\Component\ClassLoader\UniversalClassLoader` is used to autoload files that respect either the technical interoperability standards for PHP 5.3 namespaces or the PEAR naming convention for classes. As you can see here, all dependencies are stored under the `vendor/` directory, but this is just a convention. You can store them wherever you want, globally on your server or locally in your projects.

Note: If you want to learn more about the flexibility of the Symfony2 autoloader, read the "The ClassLoader Component" chapter.

Understanding the Bundle System

This section introduces one of the greatest and most powerful features of Symfony2, the bundle system.

A bundle is kind of like a plugin in other software. So why is it called a *bundle* and not a *plugin*? This is because *everything* is a bundle in Symfony2, from the core framework features to the code you write for your application. Bundles are first-class citizens in Symfony2. This gives you the flexibility to use pre-built features packaged in third-party bundles or to distribute your own bundles. It makes it easy to pick and choose which features to enable in your application and optimize them the way you want. And at the end of the day, your application code is just as *important* as the core framework itself.

Registering a Bundle

An application is made up of bundles as defined in the `registerBundles()` method of the `AppKernel` class. Each bundle is a directory that contains a single `Bundle` class that describes it:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
    );

    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
        $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }
```

```
    return $bundles;
}
```

In addition to the `AcmeDemoBundle` that we have already talked about, notice that the kernel also enables other bundles such as the `FrameworkBundle`, `DoctrineBundle`, `SwiftmailerBundle`, and `AsseticBundle` bundle. They are all part of the core framework.

Configuring a Bundle

Each bundle can be customized via configuration files written in YAML, XML, or PHP. Have a look at the default configuration:

```
# app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }

framework:
    #esi:             ~
    #translator:      { fallback: %locale% }
    secret:          %secret%
    charset:         UTF-8
    router:          { resource: "%kernel.root_dir%/config/routing.yml" }
    form:            true
    csrf_protection: true
    validation:      { enable_annotations: true }
    templating:      { engines: ['twig'] } #assets_version: SomeVersionScheme
    default_locale:  %locale%
    session:
        auto_start:     true

# Twig Configuration
twig:
    debug:            %kernel.debug%
    strict_variables: %kernel.debug%

# Assetic Configuration
assetic:
    debug:          %kernel.debug%
    use_controller: false
    bundles:        [ ]
    # java: /usr/bin/java
    filters:
        cssrewrite: ~
        # closure:
        #     jar: %kernel.root_dir%/java/compiler.jar
        # yui_css:
        #     jar: %kernel.root_dir%/java/yuicompressor-2.4.2.jar

# Doctrine Configuration
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
        port:     %database_port%
        dbname:   %database_name%
        user:     %database_user%
```

```
        password: %database_password%
        charset:  UTF8

    orm:
        auto_generate_proxy_classes: %kernel.debug%
        auto_mapping: true

# Swiftmailer Configuration
swiftmailer:
    transport: %mailer_transport%
    host:      %mailer_host%
    username:  %mailer_user%
    password:  %mailer_password%

jms_security_extra:
    secure_controllers:  true
    secure_all_services: false
```

Each entry like `framework` defines the configuration for a specific bundle. For example, `framework` configures the `FrameworkBundle` while `swiftmailer` configures the `SwiftmailerBundle`.

Each environment can override the default configuration by providing a specific configuration file. For example, the `dev` environment loads the `config_dev.yml` file, which loads the main configuration (i.e. `config.yml`) and then modifies it to add some debugging tools:

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

framework:
    router:   { resource: "%kernel.root_dir%/config/routing_dev.yml" }
    profiler: { only_exceptions: false }

web_profiler:
    toolbar: true
    intercept_redirects: false

monolog:
    handlers:
        main:
            type:  stream
            path:  %kernel.logs_dir%/%kernel.environment%.log
            level: debug
        firephp:
            type:  firephp
            level: info

assetic:
    use_controller: true
```

Extending a Bundle

In addition to being a nice way to organize and configure your code, a bundle can extend another bundle. Bundle inheritance allows you to override any existing bundle in order to customize its controllers, templates, or any of its files. This is where the logical names (e.g. `@AcmeDemoBundle/Controller/SecuredController.php`) come in handy: they abstract where the resource is actually stored.

Logical File Names When you want to reference a file from a bundle, use this notation: `@BUNDLE_NAME/path/to/file`; Symfony2 will resolve `@BUNDLE_NAME` to the real path to the bundle. For instance, the logical path `@AcmeDemoBundle/Controller/DemoController.php` would be converted to `src/Acme/DemoBundle/Controller/DemoController.php`, because Symfony knows the location of the `AcmeDemoBundle`.

Logical Controller Names For controllers, you need to reference method names using the format `BUNDLE_NAME:CONTROLLER_NAME:ACTION_NAME`. For instance, `AcmeDemoBundle:Welcome:index` maps to the `indexAction` method from the `Acme\DemoBundle\Controller\WelcomeController` class.

Logical Template Names For templates, the logical name `AcmeDemoBundle:Welcome:index.html.twig` is converted to the file path `src/Acme/DemoBundle/Resources/views/Welcome/index.html.twig`. Templates become even more interesting when you realize they don't need to be stored on the filesystem. You can easily store them in a database table for instance.

Extending Bundles If you follow these conventions, then you can use bundle inheritance to "override" files, controllers or templates. For example, you can create a bundle - `AcmeNewBundle` - and specify that its parent is `AcmeDemoBundle`. When Symfony loads the `AcmeDemoBundle:Welcome:index` controller, it will first look for the `WelcomeController` class in `AcmeNewBundle` and then look inside `AcmeDemoBundle`. This means that one bundle can override almost any part of another bundle!

Do you understand now why Symfony2 is so flexible? Share your bundles between applications, store them locally or globally, your choice.

Using Vendors

Odds are that your application will depend on third-party libraries. Those should be stored in the `vendor/` directory. This directory already contains the Symfony2 libraries, the SwiftMailer library, the Doctrine ORM, the Twig templating system, and some other third party libraries and bundles.

Understanding the Cache and Logs

Symfony2 is probably one of the fastest full-stack frameworks around. But how can it be so fast if it parses and interprets tens of YAML and XML files for each request? The speed is partly due to its cache system. The application configuration is only parsed for the very first request and then compiled down to plain PHP code stored in the `app/cache/` directory. In the development environment, Symfony2 is smart enough to flush the cache when you change a file. But in the production environment, it is your responsibility to clear the cache when you update your code or change its configuration.

When developing a web application, things can go wrong in many ways. The log files in the `app/logs/` directory tell you everything about the requests and help you fix the problem quickly.

Using the Command Line Interface

Each application comes with a command line interface tool (`app/console`) that helps you maintain your application. It provides commands that boost your productivity by automating tedious and repetitive tasks.

Run it without any arguments to learn more about its capabilities:

```
php app/console
```

The `--help` option helps you discover the usage of a command:

```
php app/console router:debug --help
```

Final Thoughts

Call me crazy, but after reading this part, you should be comfortable with moving things around and making Symfony2 work for you. Everything in Symfony2 is designed to get out of your way. So, feel free to rename and move directories around as you see fit.

And that's all for the quick tour. From testing to sending emails, you still need to learn a lot to become a Symfony2 master. Ready to dig into these topics now? Look no further - go to the official Book and pick any topic you want.

- The Big Picture >
- The View >
- The Controller >
- The Architecture

CHAPTER 2

Book

Dive into Symfony2 with the topical guides:

2.1 Book

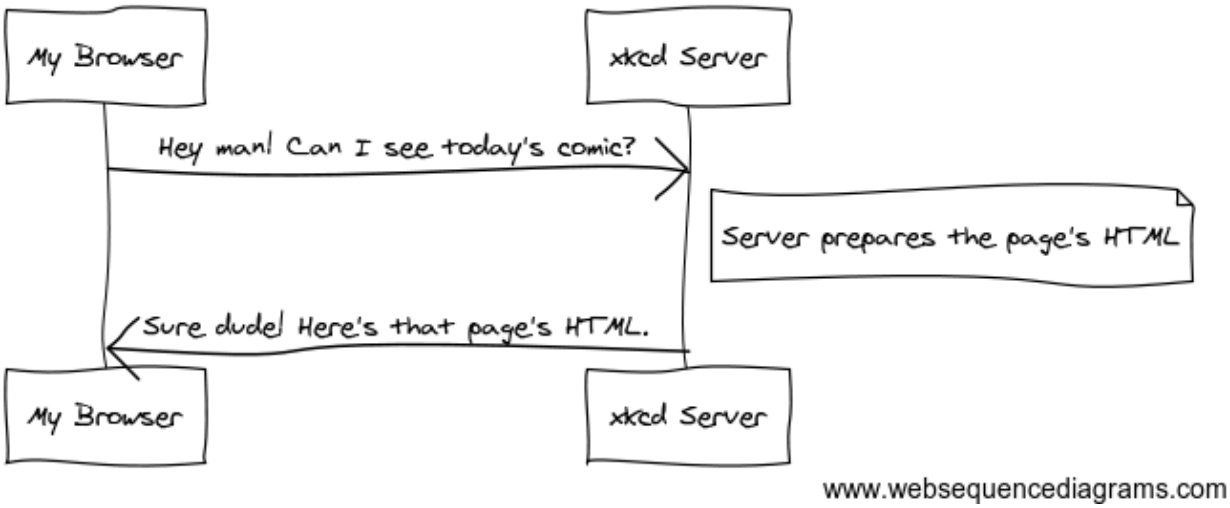
2.1.1 Symfony2 and HTTP Fundamentals

Congratulations! By learning about Symfony2, you're well on your way towards being a more *productive*, *well-rounded* and *popular* web developer (actually, you're on your own for the last part). Symfony2 is built to get back to basics: to develop tools that let you develop faster and build more robust applications, while staying out of your way. Symfony is built on the best ideas from many technologies: the tools and concepts you're about to learn represent the efforts of thousands of people, over many years. In other words, you're not just learning "Symfony", you're learning the fundamentals of the web, development best practices, and how to use many amazing new PHP libraries, inside or independent of Symfony2. So, get ready.

True to the Symfony2 philosophy, this chapter begins by explaining the fundamental concept common to web development: HTTP. Regardless of your background or preferred programming language, this chapter is a **must-read** for everyone.

HTTP is Simple

HTTP (Hypertext Transfer Protocol to the geeks) is a text language that allows two machines to communicate with each other. That's it! For example, when checking for the latest xkcd comic, the following (approximate) conversation takes place:



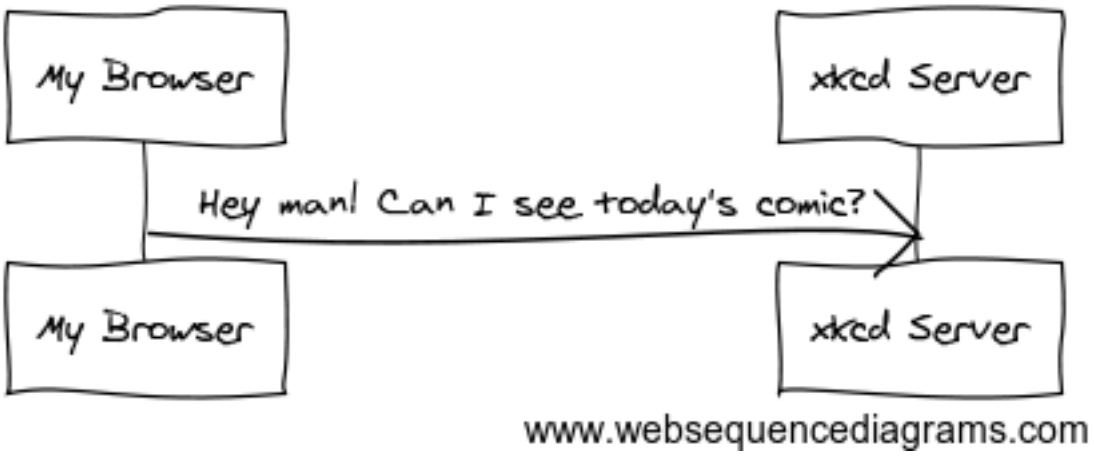
And while the actual language used is a bit more formal, it's still dead-simple. HTTP is the term used to describe this simple text-based language. And no matter how you develop on the web, the goal of your server is *always* to understand simple text requests, and return simple text responses.

Symfony2 is built from the ground-up around that reality. Whether you realize it or not, HTTP is something you use everyday. With Symfony2, you'll learn how to master it.

Step1: The Client sends a Request

Every conversation on the web starts with a *request*. The request is a text message created by a client (e.g. a browser, an iPhone app, etc) in a special format known as HTTP. The client sends that request to a server, and then waits for the response.

Take a look at the first part of the interaction (the request) between a browser and the xkcd web server:



In HTTP-speak, this HTTP request would actually look something like this:

```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

This simple message communicates *everything* necessary about exactly which resource the client is requesting. The first line of an HTTP request is the most important and contains two things: the URI and the HTTP method.

The URI (e.g. `/`, `/contact`, etc) is the unique address or location that identifies the resource the client wants. The HTTP method (e.g. `GET`) defines what you want to *do* with the resource. The HTTP methods are the *verbs* of the request and define the few common ways that you can act upon the resource:

GET	Retrieve the resource from the server
POST	Create a resource on the server
PUT	Update the resource on the server
DELETE	Delete the resource from the server

With this in mind, you can imagine what an HTTP request might look like to delete a specific blog entry, for example:

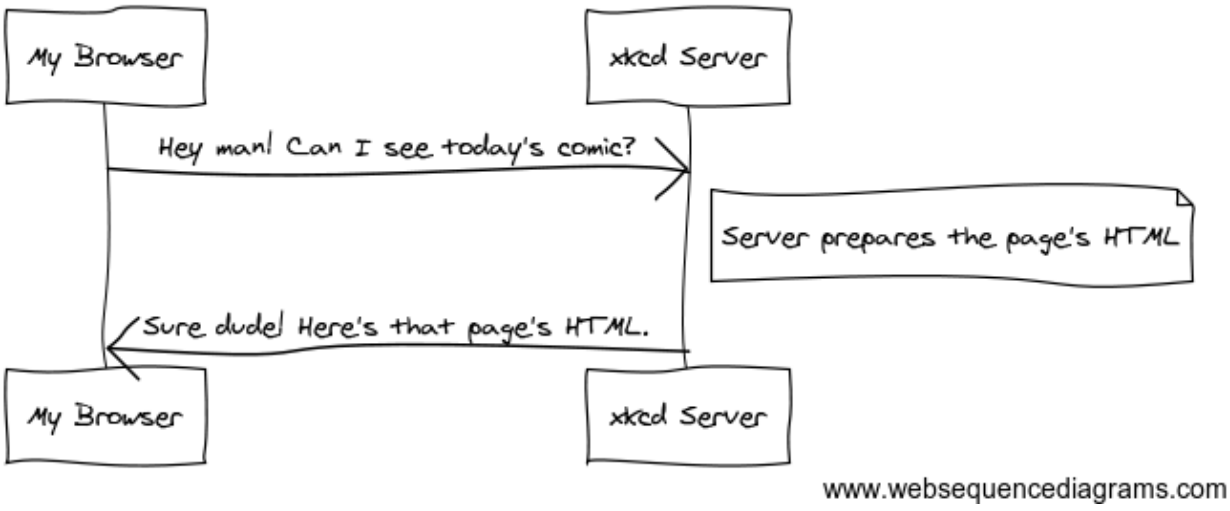
```
DELETE /blog/15 HTTP/1.1
```

Note: There are actually nine HTTP methods defined by the HTTP specification, but many of them are not widely used or supported. In reality, many modern browsers don't support the `PUT` and `DELETE` methods.

In addition to the first line, an HTTP request invariably contains other lines of information called request headers. The headers can supply a wide range of information such as the requested `Host`, the response formats the client accepts (`Accept`) and the application the client is using to make the request (`User-Agent`). Many other headers exist and can be found on Wikipedia's List of HTTP header fields article.

Step 2: The Server returns a Response

Once a server has received the request, it knows exactly which resource the client needs (via the URI) and what the client wants to do with that resource (via the method). For example, in the case of a GET request, the server prepares the resource and returns it in an HTTP response. Consider the response from the xkcd web server:



Translated into HTTP, the response sent back to the browser will look something like this:

```
HTTP/1.1 200 OK
Date: Sat, 02 Apr 2011 21:05:05 GMT
Server: lighttpd/1.4.19
Content-Type: text/html

<html>
```

```
  <!-- HTML for the xkcd comic -->
</html>
```

The HTTP response contains the requested resource (the HTML content in this case), as well as other information about the response. The first line is especially important and contains the HTTP response status code (200 in this case). The status code communicates the overall outcome of the request back to the client. Was the request successful? Was there an error? Different status codes exist that indicate success, an error, or that the client needs to do something (e.g. redirect to another page). A full list can be found on Wikipedia's List of HTTP status codes article.

Like the request, an HTTP response contains additional pieces of information known as HTTP headers. For example, one important HTTP response header is `Content-Type`. The body of the same resource could be returned in multiple different formats like HTML, XML, or JSON and the `Content-Type` header uses Internet Media Types like `text/html` to tell the client which format is being returned. A list of common media types can be found on Wikipedia's List of common media types article.

Many other headers exist, some of which are very powerful. For example, certain headers can be used to create a powerful caching system.

Requests, Responses and Web Development

This request-response conversation is the fundamental process that drives all communication on the web. And as important and powerful as this process is, it's inescapably simple.

The most important fact is this: regardless of the language you use, the type of application you build (web, mobile, JSON API), or the development philosophy you follow, the end goal of an application is **always** to understand each request and create and return the appropriate response.

Symfony is architected to match this reality.

Tip: To learn more about the HTTP specification, read the original HTTP 1.1 RFC or the HTTP Bis, which is an active effort to clarify the original specification. A great tool to check both the request and response headers while browsing is the Live HTTP Headers extension for Firefox.

Requests and Responses in PHP

So how do you interact with the "request" and create a "response" when using PHP? In reality, PHP abstracts you a bit from the whole process:

```
<?php
$uri = $_SERVER['REQUEST_URI'];
$foo = $_GET['foo'];

header('Content-type: text/html');
echo 'The URI requested is: '.$uri;
echo 'The value of the "foo" parameter is: '.$foo;
```

As strange as it sounds, this small application is in fact taking information from the HTTP request and using it to create an HTTP response. Instead of parsing the raw HTTP request message, PHP prepares superglobal variables such as `$_SERVER` and `$_GET` that contain all the information from the request. Similarly, instead of returning the HTTP-formatted text response, you can use the `header()` function to create response headers and simply print out the actual content that will be the content portion of the response message. PHP will create a true HTTP response and return it to the client:

```
HTTP/1.1 200 OK
Date: Sat, 03 Apr 2011 02:14:33 GMT
```

```
Server: Apache/2.2.17 (Unix)
Content-Type: text/html

The URI requested is: /testing?foo=symfony
The value of the "foo" parameter is: symfony
```

Requests and Responses in Symfony

Symfony provides an alternative to the raw PHP approach via two classes that allow you to interact with the HTTP request and response in an easier way. The `Symfony\Component\HttpFoundation\Request` class is a simple object-oriented representation of the HTTP request message. With it, you have all the request information at your fingertips:

```
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();

// the URI being requested (e.g. /about) minus any query parameters
$request->getPathInfo();

// retrieve GET and POST variables respectively
$request->query->get('foo');
$request->request->get('bar', 'default value if bar does not exist');

// retrieve SERVER variables
$request->server->get('HTTP_HOST');

// retrieves an instance of UploadedFile identified by foo
$request->files->get('foo');

// retrieve a COOKIE value
$request->cookies->get('PHPSESSID');

// retrieve an HTTP request header, with normalized, lowercase keys
$request->headers->get('host');
$request->headers->get('content_type');

$request->getMethod();          // GET, POST, PUT, DELETE, HEAD
$request->getLanguages();       // an array of languages the client accepts
```

As a bonus, the `Request` class does a lot of work in the background that you'll never need to worry about. For example, the `isSecure()` method checks the *three* different values in PHP that can indicate whether or not the user is connecting via a secured connection (i.e. `https`).

ParameterBags and Request attributes

As seen above, the `$_GET` and `$_POST` variables are accessible via the public `query` and `request` properties respectively. Each of these objects is a `Symfony\Component\HttpFoundation\ParameterBag` object, which has methods like **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::get'**, **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::has'**, **:method:'Symfony\\Component\\HttpFoundation\\Par** and more. In fact, every public property used in the previous example is some instance of the ParameterBag.

The Request class also has a public `attributes` property, which holds special data related to how the application works internally. For the Symfony2 framework, the `attributes` holds the values returned by the matched route, like `_controller`, `id` (if you have an `{id}` wildcard), and even the name of the matched route (`_route`). The `attributes` property exists entirely to be a place where you can prepare and store context-specific information about the request.

Symfony also provides a `Response` class: a simple PHP representation of an HTTP response message. This allows your application to use an object-oriented interface to construct the response that needs to be returned to the client:

```
use Symfony\Component\HttpFoundation\Response;
$response = new Response();

$response->setContent('<html><body><h1>Hello world!</h1></body></html>');
$response->setStatusCode(200);
$response->headers->set('Content-Type', 'text/html');

// prints the HTTP headers followed by the content
$response->send();
```

If Symfony offered nothing else, you would already have a toolkit for easily accessing request information and an object-oriented interface for creating the response. Even as you learn the many powerful features in Symfony, keep in mind that the goal of your application is always *to interpret a request and create the appropriate response based on your application logic*.

Tip: The `Request` and `Response` classes are part of a standalone component included with Symfony called `HttpFoundation`. This component can be used entirely independent of Symfony and also provides classes for handling sessions and file uploads.

The Journey from the Request to the Response

Like HTTP itself, the `Request` and `Response` objects are pretty simple. The hard part of building an application is writing what comes in between. In other words, the real work comes in writing the code that interprets the request information and creates the response.

Your application probably does many things, like sending emails, handling form submissions, saving things to a database, rendering HTML pages and protecting content with security. How can you manage all of this and still keep your code organized and maintainable?

Symfony was created to solve these problems so that you don't have to.

The Front Controller

Traditionally, applications were built so that each "page" of a site was its own physical file:

```
index.php
contact.php
blog.php
```

There are several problems with this approach, including the inflexibility of the URLs (what if you wanted to change `blog.php` to `news.php` without breaking all of your links?) and the fact that each file *must* manually include some set of core files so that security, database connections and the "look" of the site can remain consistent.

A much better solution is to use a front controller: a single PHP file that handles every request coming into your application. For example:

`/index.php`	executes `index.php`
`/index.php/contact`	executes `index.php`
`/index.php/blog`	executes `index.php`

Tip: Using Apache's `mod_rewrite` (or equivalent with other web servers), the URLs can easily be cleaned up to be just `/`, `/contact` and `/blog`.

Now, every request is handled exactly the same. Instead of individual URLs executing different PHP files, the front controller is *always* executed, and the routing of different URLs to different parts of your application is done internally. This solves both problems with the original approach. Almost all modern web apps do this - including apps like WordPress.

Stay Organized

But inside your front controller, how do you know which page should be rendered and how can you render each in a sane way? One way or another, you'll need to check the incoming URI and execute different parts of your code depending on that value. This can get ugly quickly:

```
// index.php

$request = Request::createFromGlobals();
$path = $request->getPathInfo(); // the URI path being requested

if (in_array($path, array('', '/')) {
    $response = new Response('Welcome to the homepage.');
} elseif ($path == '/contact') {
    $response = new Response('Contact us');
} else {
    $response = new Response('Page not found.', 404);
}
$response->send();
```

Solving this problem can be difficult. Fortunately it's *exactly* what Symfony is designed to do.

The Symfony Application Flow

When you let Symfony handle each request, life is much easier. Symfony follows the same simple pattern for every request:

Each "page" of your site is defined in a routing configuration file that maps different URLs to different PHP functions. The job of each PHP function, called a controller, is to use information from the request - along with many other tools Symfony makes available - to create and return a `Response` object. In other words, the controller is where *your* code goes: it's where you interpret the request and create a response.

It's that easy! Let's review:

- Each request executes a front controller file;
- The routing system determines which PHP function should be executed based on information from the request and routing configuration you've created;

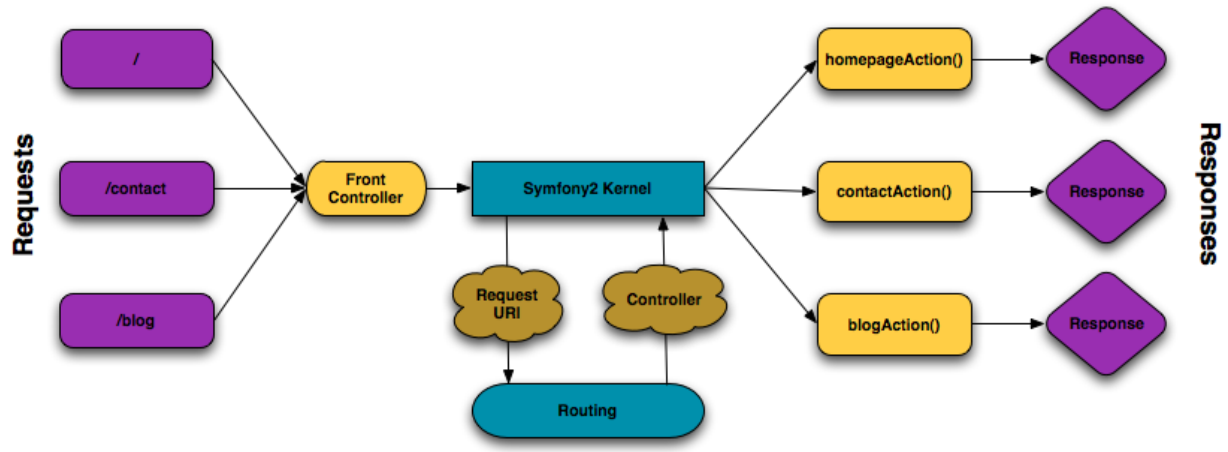


Fig. 2.1: Incoming requests are interpreted by the routing and passed to controller functions that return `Response` objects.

- The correct PHP function is executed, where your code creates and returns the appropriate `Response` object.

A Symfony Request in Action

Without diving into too much detail, let's see this process in action. Suppose you want to add a `/contact` page to your Symfony application. First, start by adding an entry for `/contact` to your routing configuration file:

```
contact:
    pattern:  /contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
```

Note: This example uses YAML to define the routing configuration. Routing configuration can also be written in other formats such as XML or PHP.

When someone visits the `/contact` page, this route is matched, and the specified controller is executed. As you'll learn in the routing chapter, the `AcmeDemoBundle:Main:contact` string is a short syntax that points to a specific PHP method `contactAction` inside a class called `MainController`:

```
class MainController
{
    public function contactAction()
    {
        return new Response('<h1>Contact us!</h1>');
    }
}
```

In this very simple example, the controller simply creates a `Response` object with the HTML "<h1>Contact us!</h1>". In the controller chapter, you'll learn how a controller can render templates, allowing your "presentation" code (i.e. anything that actually writes out HTML) to live in a separate template file. This frees up the controller to worry only about the hard stuff: interacting with the database, handling submitted data, or sending email messages.

Symfony2: Build your App, not your Tools.

You now know that the goal of any app is to interpret each incoming request and create an appropriate response. As an application grows, it becomes more difficult to keep your code organized and maintainable. Invariably, the same complex tasks keep coming up over and over again: persisting things to the database, rendering and reusing templates, handling form submissions, sending emails, validating user input and handling security.

The good news is that none of these problems is unique. Symfony provides a framework full of tools that allow you to build your application, not your tools. With Symfony2, nothing is imposed on you: you're free to use the full Symfony framework, or just one piece of Symfony all by itself.

Standalone Tools: The Symfony2 *Components*

So what *is* Symfony2? First, Symfony2 is a collection of over twenty independent libraries that can be used inside *any* PHP project. These libraries, called the *Symfony2 Components*, contain something useful for almost any situation, regardless of how your project is developed. To name a few:

- HttpFoundation - Contains the `Request` and `Response` classes, as well as other classes for handling sessions and file uploads;
- Routing - Powerful and fast routing system that allows you to map a specific URI (e.g. `/contact`) to some information about how that request should be handled (e.g. execute the `contactAction()` method);
- Form - A full-featured and flexible framework for creating forms and handing form submissions;
- Validator A system for creating rules about data and then validating whether or not user-submitted data follows those rules;
- ClassLoader An autoloading library that allows PHP classes to be used without needing to manually `require` the files containing those classes;
- Templating A toolkit for rendering templates, handling template inheritance (i.e. a template is decorated with a layout) and performing other common template tasks;
- Security - A powerful library for handling all types of security inside an application;
- Translation A framework for translating strings in your application.

Each and every one of these components is decoupled and can be used in *any* PHP project, regardless of whether or not you use the Symfony2 framework. Every part is made to be used if needed and replaced when necessary.

The Full Solution: The Symfony2 *Framework*

So then, what *is* the Symfony2 *Framework*? The *Symfony2 Framework* is a PHP library that accomplishes two distinct tasks:

1. Provides a selection of components (i.e. the Symfony2 Components) and third-party libraries (e.g. `Swiftmailer` for sending emails);
2. Provides sensible configuration and a "glue" library that ties all of these pieces together.

The goal of the framework is to integrate many independent tools in order to provide a consistent experience for the developer. Even the framework itself is a Symfony2 bundle (i.e. a plugin) that can be configured or replaced entirely.

Symfony2 provides a powerful set of tools for rapidly developing web applications without imposing on your application. Normal users can quickly start development by using a Symfony2 distribution, which provides a project skeleton with sensible defaults. For more advanced users, the sky is the limit.

2.1.2 Symfony2 versus Flat PHP

Why is Symfony2 better than just opening up a file and writing flat PHP?

If you've never used a PHP framework, aren't familiar with the MVC philosophy, or just wonder what all the *hype* is around Symfony2, this chapter is for you. Instead of *telling* you that Symfony2 allows you to develop faster and better software than with flat PHP, you'll see for yourself.

In this chapter, you'll write a simple application in flat PHP, and then refactor it to be more organized. You'll travel through time, seeing the decisions behind why web development has evolved over the past several years to where it is now.

By the end, you'll see how Symfony2 can rescue you from mundane tasks and let you take back control of your code.

A simple Blog in flat PHP

In this chapter, you'll build the token blog application using only flat PHP. To begin, create a single page that displays blog entries that have been persisted to the database. Writing in flat PHP is quick and dirty:

```
<?php
// index.php

$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);
?>

<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php while ($row = mysql_fetch_assoc($result)): ?>
            <li>
                <a href="/show.php?id=<?php echo $row['id'] ?>">
                    <?php echo $row['title'] ?>
                </a>
            </li>
            <?php endwhile; ?>
        </ul>
    </body>
</html>

<?php
mysql_close($link);
```

That's quick to write, fast to execute, and, as your app grows, impossible to maintain. There are several problems that need to be addressed:

- **No error-checking**: What if the connection to the database fails?
- **Poor organization**: If the application grows, this single file will become increasingly unmaintainable. Where should you put code to handle a form submission? How can you validate data? Where should code go for sending emails?

- **Difficult to reuse code**: Since everything is in one file, there's no way to reuse any part of the application for other "pages" of the blog.

Note: Another problem not mentioned here is the fact that the database is tied to MySQL. Though not covered here, Symfony2 fully integrates Doctrine, a library dedicated to database abstraction and mapping.

Let's get to work on solving these problems and more.

Isolating the Presentation

The code can immediately gain from separating the application "logic" from the code that prepares the HTML "presentation":

```
<?php
// index.php

$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);

$posts = array();
while ($row = mysql_fetch_assoc($result)) {
    $posts[] = $row;
}

mysql_close($link);

// include the HTML presentation code
require 'templates/list.php';
```

The HTML code is now stored in a separate file (`templates/list.php`), which is primarily an HTML file that uses a template-like PHP syntax:

```
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php foreach ($posts as $post): ?>
            <li>
                <a href="/read?id=<?php echo $post['id'] ?>">
                    <?php echo $post['title'] ?>
                </a>
            </li>
            <?php endforeach; ?>
        </ul>
    </body>
</html>
```

By convention, the file that contains all of the application logic - `index.php` - is known as a "controller". The term controller is a word you'll hear a lot, regardless of the language or framework you use. It refers simply to the area of *your* code that processes user input and prepares the response.

In this case, our controller prepares data from the database and then includes a template to present that data. With the

controller isolated, you could easily change *just* the template file if you needed to render the blog entries in some other format (e.g. `list.json.php` for JSON format).

Isolating the Application (Domain) Logic

So far the application contains only one page. But what if a second page needed to use the same database connection, or even the same array of blog posts? Refactor the code so that the core behavior and data-access functions of the application are isolated in a new file called `model.php`:

```
<?php
// model.php

function open_database_connection()
{
    $link = mysql_connect('localhost', 'myuser', 'mypassword');
    mysql_select_db('blog_db', $link);

    return $link;
}

function close_database_connection($link)
{
    mysql_close($link);
}

function get_all_posts()
{
    $link = open_database_connection();

    $result = mysql_query('SELECT id, title FROM post', $link);
    $posts = array();
    while ($row = mysql_fetch_assoc($result)) {
        $posts[] = $row;
    }
    close_database_connection($link);

    return $posts;
}
```

Tip: The filename `model.php` is used because the logic and data access of an application is traditionally known as the "model" layer. In a well-organized application, the majority of the code representing your "business logic" should live in the model (as opposed to living in a controller). And unlike in this example, only a portion (or none) of the model is actually concerned with accessing a database.

The controller (`index.php`) is now very simple:

```
<?php
require_once 'model.php';

$posts = get_all_posts();

require 'templates/list.php';
```

Now, the sole task of the controller is to get data from the model layer of the application (the model) and to call a template to render that data. This is a very simple example of the model-view-controller pattern.

Isolating the Layout

At this point, the application has been refactored into three distinct pieces offering various advantages and the opportunity to reuse almost everything on different pages.

The only part of the code that *can't* be reused is the page layout. Fix that by creating a new `layout.php` file:

```
<!-- templates/layout.php -->
<html>
    <head>
        <title><?php echo $title ?></title>
    </head>
    <body>
        <?php echo $content ?>
    </body>
</html>
```

The template (`templates/list.php`) can now be simplified to "extend" the layout:

```
<?php $title = 'List of Posts' ?>

<?php ob_start() ?>
    <h1>List of Posts</h1>
    <ul>
        <?php foreach ($posts as $post): ?>
        <li>
            <a href="/read?id=<?php echo $post['id'] ?>">
                <?php echo $post['title'] ?>
            </a>
        </li>
        <?php endforeach; ?>
    </ul>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

You've now introduced a methodology that allows for the reuse of the layout. Unfortunately, to accomplish this, you're forced to use a few ugly PHP functions (`ob_start()`, `ob_get_clean()`) in the template. Symfony2 uses a `Templating` component that allows this to be accomplished cleanly and easily. You'll see it in action shortly.

Adding a Blog "show" Page

The blog "list" page has now been refactored so that the code is better-organized and reusable. To prove it, add a blog "show" page, which displays an individual blog post identified by an `id` query parameter.

To begin, create a new function in the `model.php` file that retrieves an individual blog result based on a given id:

```
// model.php
function get_post_by_id($id)
{
    $link = open_database_connection();

    $id = mysql_real_escape_string($id);
    $query = 'SELECT date, title, body FROM post WHERE id = '.$id;
    $result = mysql_query($query);
    $row = mysql_fetch_assoc($result);

    close_database_connection($link);
```

```
    return $row;
}
```

Next, create a new file called `show.php` - the controller for this new page:

```
<?php
require_once 'model.php';

$post = get_post_by_id($_GET['id']);

require 'templates/show.php';
```

Finally, create the new template file - `templates/show.php` - to render the individual blog post:

```
<?php $title = $post['title'] ?>

<?php ob_start() ?>
    <h1><?php echo $post['title'] ?></h1>

    <div class="date"><?php echo $post['date'] ?></div>
    <div class="body">
        <?php echo $post['body'] ?>
    </div>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Creating the second page is now very easy and no code is duplicated. Still, this page introduces even more lingering problems that a framework can solve for you. For example, a missing or invalid `id` query parameter will cause the page to crash. It would be better if this caused a 404 page to be rendered, but this can't really be done easily yet. Worse, had you forgotten to clean the `id` parameter via the `mysql_real_escape_string()` function, your entire database would be at risk for an SQL injection attack.

Another major problem is that each individual controller file must include the `model.php` file. What if each controller file suddenly needed to include an additional file or perform some other global task (e.g. enforce security)? As it stands now, that code would need to be added to every controller file. If you forget to include something in one file, hopefully it doesn't relate to security...

A "Front Controller" to the Rescue

The solution is to use a front controller: a single PHP file through which *all* requests are processed. With a front controller, the URIs for the application change slightly, but start to become more flexible:

```
Without a front controller
/index.php          => Blog post list page (index.php executed)
/show.php           => Blog post show page (show.php executed)

With index.php as the front controller
/index.php          => Blog post list page (index.php executed)
/index.php/show     => Blog post show page (index.php executed)
```

Tip: The `index.php` portion of the URI can be removed if using Apache rewrite rules (or equivalent). In that case, the resulting URI of the blog show page would be simply `/show`.

When using a front controller, a single PHP file (`index.php` in this case) renders *every* request. For the blog post show page, `/index.php/show` will actually execute the `index.php` file, which is now responsible for routing

requests internally based on the full URI. As you'll see, a front controller is a very powerful tool.

Creating the Front Controller

You're about to take a **big** step with the application. With one file handling all requests, you can centralize things such as security handling, configuration loading, and routing. In this application, `index.php` must now be smart enough to render the blog post list page *or* the blog post show page based on the requested URI:

```
<?php
// index.php

// load and initialize any global libraries
require_once 'model.php';
require_once 'controllers.php';

// route the request internally
$uri = $_SERVER['REQUEST_URI'];
if ($uri == '/index.php') {
    list_action();
} elseif ($uri == '/index.php/show' && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Page Not Found</h1></body></html>';
}
```

For organization, both controllers (formerly `index.php` and `show.php`) are now PHP functions and each has been moved into a separate file, `controllers.php`:

```
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

As a front controller, `index.php` has taken on an entirely new role, one that includes loading the core libraries and routing the application so that one of the two controllers (the `list_action()` and `show_action()` functions) is called. In reality, the front controller is beginning to look and act a lot like Symfony2's mechanism for handling and routing requests.

Tip: Another advantage of a front controller is flexible URLs. Notice that the URL to the blog post show page could be changed from `/show` to `/read` by changing code in only one location. Before, an entire file needed to be renamed. In Symfony2, URLs are even more flexible.

By now, the application has evolved from a single PHP file into a structure that is organized and allows for code reuse. You should be happier, but far from satisfied. For example, the "routing" system is fickle, and wouldn't recognize that the list page (`/index.php`) should be accessible also via `/` (if Apache rewrite rules were added). Also, instead of developing the blog, a lot of time is being spent working on the "architecture" of the code (e.g. routing, calling controllers, templates, etc.). More time will need to be spent to handle form submissions, input validation, logging and security. Why should you have to reinvent solutions to all these routine problems?

Add a Touch of Symfony2

Symfony2 to the rescue. Before actually using Symfony2, you need to make sure PHP knows how to find the Symfony2 classes. This is accomplished via an autoloader that Symfony provides. An autoloader is a tool that makes it possible to start using PHP classes without explicitly including the file containing the class.

First, download symfony and place it into a `vendor/symfony/` directory. Next, create an `app/bootstrap.php` file. Use it to `require` the two files in the application and to configure the autoloader:

```
<?php
// bootstrap.php
require_once 'model.php';
require_once 'controllers.php';
require_once 'vendor/symfony/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';

$loader = new Symfony\Component\ClassLoader\UniversalClassLoader();
$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/../vendor/symfony/src',
));

$loader->register();
```

This tells the autoloader where the `Symfony` classes are. With this, you can start using Symfony classes without using the `require` statement for the files that contain them.

Core to Symfony's philosophy is the idea that an application's main job is to interpret each request and return a response. To this end, Symfony2 provides both a `Symfony\Component\HttpFoundation\Request` and a `Symfony\Component\HttpFoundation\Response` class. These classes are object-oriented representations of the raw HTTP request being processed and the HTTP response being returned. Use them to improve the blog:

```
<?php
// index.php
require_once 'app/bootstrap.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();

$uri = $request->getPathInfo();
if ($uri == '/') {
    $response = list_action();
} elseif ($uri == '/show' && $request->query->has('id')) {
    $response = show_action($request->query->get('id'));
} else {
    $html = '<html><body><h1>Page Not Found</h1></body></html>';
    $response = new Response($html, 404);
}

// echo the headers and send the response
$response->send();
```

The controllers are now responsible for returning a `Response` object. To make this easier, you can add a new `render_template()` function, which, incidentally, acts quite a bit like the Symfony2 templating engine:

```
// controllers.php
use Symfony\Component\HttpFoundation\Response;

function list_action()
```

```
{
    $posts = get_all_posts();
    $html = render_template('templates/list.php', array('posts' => $posts));

    return new Response($html);
}

function show_action($id)
{
    $post = get_post_by_id($id);
    $html = render_template('templates/show.php', array('post' => $post));

    return new Response($html);
}

// helper function to render templates
function render_template($path, array $args)
{
    extract($args);
    ob_start();
    require $path;
    $html = ob_get_clean();

    return $html;
}
```

By bringing in a small part of Symfony2, the application is more flexible and reliable. The `Request` provides a dependable way to access information about the HTTP request. Specifically, the `getPathInfo()` method returns a cleaned URI (always returning `/show` and never `/index.php/show`). So, even if the user goes to `/index.php/show`, the application is intelligent enough to route the request through `show_action()`.

The `Response` object gives flexibility when constructing the HTTP response, allowing HTTP headers and content to be added via an object-oriented interface. And while the responses in this application are simple, this flexibility will pay dividends as your application grows.

The Sample Application in Symfony2

The blog has come a *long* way, but it still contains a lot of code for such a simple application. Along the way, we've also invented a simple routing system and a method using `ob_start()` and `ob_get_clean()` to render templates. If, for some reason, you needed to continue building this "framework" from scratch, you could at least use Symfony's standalone Routing and Templating components, which already solve these problems.

Instead of re-solving common problems, you can let Symfony2 take care of them for you. Here's the same sample application, now built in Symfony2:

```
<?php
// src/Acme/BlogBundle/Controller/BlogController.php

namespace Acme\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function listAction()
    {
        $posts = $this->get('doctrine')->getEntityManager()
            ->createQuery('SELECT p FROM AcmeBlogBundle:Post p')
```

```
            ->execute();

        return $this->render('AcmeBlogBundle:Blog:list.html.php', array('posts' => $posts));
    }

    public function showAction($id)
    {
        $post = $this->get('doctrine')
            ->getEntityManager()
            ->getRepository('AcmeBlogBundle:Post')
            ->find($id);

        if (!$post) {
            // cause the 404 page not found to be displayed
            throw $this->createNotFoundException();
        }

        return $this->render('AcmeBlogBundle:Blog:show.html.php', array('post' => $post));
    }
}
```

The two controllers are still lightweight. Each uses the Doctrine ORM library to retrieve objects from the database and the `Templating` component to render a template and return a `Response` object. The list template is now quite a bit simpler:

```
<!-- src/Acme/BlogBundle/Resources/views/Blog/list.html.php -->
<?php $view->extend('::layout.html.php') ?>

<?php $view['slots']->set('title', 'List of Posts') ?>

<h1>List of Posts</h1>
<ul>
    <?php foreach ($posts as $post): ?>
    <li>
        <a href="<?php echo $view['router']->generate('blog_show', array('id' => $post->getId())) ?>">
            <?php echo $post->getTitle() ?>
        </a>
    </li>
    <?php endforeach; ?>
</ul>
```

The layout is nearly identical:

```
<!-- app/Resources/views/layout.html.php -->
<html>
    <head>
        <title><?php echo $view['slots']->output('title', 'Default title') ?></title>
    </head>
    <body>
        <?php echo $view['slots']->output('_content') ?>
    </body>
</html>
```

Note: We'll leave the show template as an exercise, as it should be trivial to create based on the list template.

When Symfony2's engine (called the `Kernel`) boots up, it needs a map so that it knows which controllers to execute based on the request information. A routing configuration map provides this information in a readable format:

```
# app/config/routing.yml
blog_list:
    pattern:  /blog
    defaults: { _controller: AcmeBlogBundle:Blog:list }

blog_show:
    pattern:  /blog/show/{id}
    defaults: { _controller: AcmeBlogBundle:Blog:show }
```

Now that Symfony2 is handling all the mundane tasks, the front controller is dead simple. And since it does so little, you'll never have to touch it once it's created (and if you use a Symfony2 distribution, you won't even need to create it!):

```
<?php
// web/app.php
require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->handle(Request::createFromGlobals())->send();
```

The front controller's only job is to initialize Symfony2's engine (`Kernel`) and pass it a `Request` object to handle. Symfony2's core then uses the routing map to determine which controller to call. Just like before, the controller method is responsible for returning the final `Response` object. There's really not much else to it.

For a visual representation of how Symfony2 handles each request, see the *request flow diagram*.

Where Symfony2 Delivers

In the upcoming chapters, you'll learn more about how each piece of Symfony works and the recommended organization of a project. For now, let's see how migrating the blog from flat PHP to Symfony2 has improved life:

- Your application now has **clear and consistently organized code** (though Symfony doesn't force you into this). This promotes **reusability** and allows for new developers to be productive in your project more quickly.
- 100% of the code you write is for *your* application. You **don't need to develop or maintain low-level utilities** such as *autoloading*, routing, or rendering controllers.
- Symfony2 gives you **access to open source tools** such as Doctrine and the Templating, Security, Form, Validation and Translation components (to name a few).
- The application now enjoys **fully-flexible URLs** thanks to the `Routing` component.
- Symfony2's HTTP-centric architecture gives you access to powerful tools such as **HTTP caching** powered by **Symfony2's internal HTTP cache** or more powerful tools such as Varnish. This is covered in a later chapter all about caching.

And perhaps best of all, by using Symfony2, you now have access to a whole set of **high-quality open source tools developed by the Symfony2 community**! A good selection of Symfony2 community tools can be found on KnpBundles.com.

Better templates

If you choose to use it, Symfony2 comes standard with a templating engine called Twig that makes templates faster to write and easier to read. It means that the sample application could contain even less code! Take, for example, the list template written in Twig:

```
{# src/Acme/BlogBundle/Resources/views/Blog/list.html.twig #}

{% extends "::layout.html.twig" %}
{% block title %}List of Posts{% endblock %}

{% block body %}
    <h1>List of Posts</h1>
    <ul>
        {% for post in posts %}
        <li>
            <a href="{{ path('blog_show', { 'id': post.id }) }}">
                {{ post.title }}
            </a>
        </li>
        {% endfor %}
    </ul>
{% endblock %}
```

The corresponding `layout.html.twig` template is also easier to write:

```
{# app/Resources/views/layout.html.twig #}

<html>
    <head>
        <title>{% block title %}Default title{% endblock %}</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Twig is well-supported in Symfony2. And while PHP templates will always be supported in Symfony2, we'll continue to discuss the many advantages of Twig. For more information, see the templating chapter.

Learn more from the Cookbook

- How to use PHP instead of Twig for Templates
- How to define Controllers as Services

2.1.3 Installing and Configuring Symfony

The goal of this chapter is to get you up and running with a working application built on top of Symfony. Fortunately, Symfony offers "distributions", which are functional Symfony "starter" projects that you can download and begin developing in immediately.

Tip: If you're looking for instructions on how best to create a new project and store it via source control, see *Using Source Control*.

Downloading a Symfony2 Distribution

Tip: First, check that you have installed and configured a Web server (such as Apache) with PHP 5.3.2 or higher. For more information on Symfony2 requirements, see the requirements reference.

Symfony2 packages "distributions", which are fully-functional applications that include the Symfony2 core libraries, a selection of useful bundles, a sensible directory structure and some default configuration. When you download a Symfony2 distribution, you're downloading a functional application skeleton that can be used immediately to begin developing your application.

Start by visiting the Symfony2 download page at http://symfony.com/download. On this page, you'll see the *Symfony Standard Edition*, which is the main Symfony2 distribution. Here, you'll need to make two choices:

- Download either a `.tgz` or `.zip` archive - both are equivalent, download whatever you're more comfortable using;
- Download the distribution with or without vendors. If you have Git installed on your computer, you should download Symfony2 "without vendors", as it adds a bit more flexibility when including third-party/vendor libraries.

Download one of the archives somewhere under your local web server's root directory and unpack it. From a UNIX command line, this can be done with one of the following commands (replacing `###` with your actual filename):

```
# for .tgz file
tar zxvf Symfony_Standard_Vendors_2.0.###.tgz

# for a .zip file
unzip Symfony_Standard_Vendors_2.0.###.zip
```

When you're finished, you should have a `Symfony/` directory that looks something like this:

```
www/ <- your web root directory
    Symfony/ <- the unpacked archive
        app/
            cache/
            config/
            logs/
        src/
            ...
        vendor/
            ...
        web/
            app.php
            ...
```

Updating Vendors

Finally, if you downloaded the archive "without vendors", install the vendors by running the following command from the command line:

```
php bin/vendors install
```

This command downloads all of the necessary vendor libraries - including Symfony itself - into the `vendor/` directory. For more information on how third-party vendor libraries are managed inside Symfony2, see "cookbook-managing-vendor-libraries".

Configuration and Setup

At this point, all of the needed third-party libraries now live in the `vendor/` directory. You also have a default application setup in `app/` and some sample code inside the `src/` directory.

Symfony2 comes with a visual server configuration tester to help make sure your Web server and PHP are configured to use Symfony. Use the following URL to check your configuration:

```
http://localhost/Symfony/web/config.php
```

If there are any issues, correct them now before moving on.

Setting up Permissions

One common issue is that the `app/cache` and `app/logs` directories must be writable both by the web server and the command line user. On a UNIX system, if your web server user is different from your command line user, you can run the following commands just once in your project to ensure that permissions will be setup properly. Change `www-data` to your web server user:

1. Using ACL on a system that supports chmod +a

Many systems allow you to use the `chmod +a` command. Try this first, and if you get an error - try the next method:

```
rm -rf app/cache/*
rm -rf app/logs/*

sudo chmod +a "www-data allow delete,write,append,file_inherit,directory_inherit" app/cache app/logs
sudo chmod +a "`whoami` allow delete,write,append,file_inherit,directory_inherit" app/cache app/logs
```

2. Using Acl on a system that does not support chmod +a

Some systems don't support `chmod +a`, but do support another utility called `setfacl`. You may need to enable ACL support on your partition and install setfacl before using it (as is the case with Ubuntu), like so:

```
sudo setfacl -R -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
sudo setfacl -dR -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
```

3. Without using ACL

If you don't have access to changing the ACL of the directories, you will need to change the umask so that the cache and log directories will be group-writable or world-writable (depending if the web server user and the command line user are in the same group or not). To achieve this, put the following line at the beginning of the `app/console`, `web/app.php` and `web/app_dev.php` files:

```
umask(0002); // This will let the permissions be 0775

// or

umask(0000); // This will let the permissions be 0777
```

Note that using the ACL is recommended when you have access to them on your server because changing the umask is not thread-safe.

When everything is fine, click on "Go to the Welcome page" to request your first "real" Symfony2 webpage:

```
http://localhost/Symfony/web/app_dev.php/
```

Symfony2 should welcome and congratulate you for your hard work so far!

Beginning Development

Now that you have a fully-functional Symfony2 application, you can begin development! Your distribution may contain some sample code - check the `README.rst` file included with the distribution (open it as a text file) to learn about what sample code was included with your distribution and how you can remove it later.

If you're new to Symfony, join us in the "Creating Pages in Symfony2", where you'll learn how to create pages, change configuration, and do everything else you'll need in your new application.

Using Source Control

If you're using a version control system like `Git` or `Subversion`, you can setup your version control system and begin committing your project to it as normal. The Symfony Standard edition *is* the starting point for your new project.

For specific instructions on how best to setup your project to be stored in git, see How to Create and store a Symfony2 Project in git.

Ignoring the `vendor/` Directory

If you've downloaded the archive *without vendors*, you can safely ignore the entire `vendor/` directory and not commit it to source control. With `Git`, this is done by creating and adding the following to a `.gitignore` file:

```
vendor/
```

Now, the vendor directory won't be committed to source control. This is fine (actually, it's great!) because when someone else clones or checks out the project, he/she can simply run the `php bin/vendors install` script to download all the necessary vendor libraries.

2.1.4 Creating Pages in Symfony2

Creating a new page in Symfony2 is a simple two-step process:

- *Create a route*: A route defines the URL (e.g. `/about`) to your page and specifies a controller (which is a PHP function) that Symfony2 should execute when the URL of an incoming request matches the route pattern;
- *Create a controller*: A controller is a PHP function that takes the incoming request and transforms it into the Symfony2 `Response` object that's returned to the user.

This simple approach is beautiful because it matches the way that the Web works. Every interaction on the Web is initiated by an HTTP request. The job of your application is simply to interpret the request and return the appropriate HTTP response.

Symfony2 follows this philosophy and provides you with tools and conventions to keep your application organized as it grows in users and complexity.

Sounds simple enough? Let's dive in!

The "Hello Symfony!" Page

Let's start with a spin off of the classic "Hello World!" application. When you're finished, the user will be able to get a personal greeting (e.g. "Hello Symfony") by going to the following URL:

```
http://localhost/app_dev.php/hello/Symfony
```

Actually, you'll be able to replace `Symfony` with any other name to be greeted. To create the page, follow the simple two-step process.

Note: The tutorial assumes that you've already downloaded Symfony2 and configured your webserver. The above URL assumes that `localhost` points to the `web` directory of your new Symfony2 project. For detailed information on this process, see the Installing Symfony2.

Before you begin: Create the Bundle

Before you begin, you'll need to create a *bundle*. In Symfony2, a bundle is like a plugin, except that all of the code in your application will live inside a bundle.

A bundle is nothing more than a directory that houses everything related to a specific feature, including PHP classes, configuration, and even stylesheets and Javascript files (see *The Bundle System*).

To create a bundle called `AcmeHelloBundle` (a play bundle that you'll build in this chapter), run the following command and follow the on-screen instructions (use all of the default options):

```
php app/console generate:bundle --namespace=Acme/HelloBundle --format=yml
```

Behind the scenes, a directory is created for the bundle at `src/Acme/HelloBundle`. A line is also automatically added to the `app/AppKernel.php` file so that the bundle is registered with the kernel:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Acme\HelloBundle\AcmeHelloBundle(),
    );
    // ...
```

```
    return $bundles;
}
```

Now that you have a bundle setup, you can begin building your application inside the bundle.

Step 1: Create the Route

By default, the routing configuration file in a Symfony2 application is located at `app/config/routing.yml`. Like all configuration in Symfony2, you can also choose to use XML or PHP out of the box to configure routes.

If you look at the main routing file, you'll see that Symfony already added an entry when you generated the `AcmeHelloBundle`:

- *YAML*

```
# app/config/routing.yml
AcmeHelloBundle:
    resource: "@AcmeHelloBundle/Resources/config/routing.yml"
    prefix:   /
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@AcmeHelloBundle/Resources/config/routing.xml" prefix="/" />
</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->addCollection(
    $loader->import('@AcmeHelloBundle/Resources/config/routing.php'),
    '/',
);

return $collection;
```

This entry is pretty basic: it tells Symfony to load routing configuration from the `Resources/config/routing.yml` file that lives inside the `AcmeHelloBundle`. This means that you place routing configuration directly in `app/config/routing.yml` or organize your routes throughout your application, and import them from here.

Now that the `routing.yml` file from the bundle is being imported, add the new route that defines the URL of the page that you're about to create:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/routing.yml
hello:
```

```
        pattern:  /hello/{name}
        defaults: { _controller: AcmeHelloBundle:Hello:index }
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="hello" pattern="/hello/{name}">
        <default key="_controller">AcmeHelloBundle:Hello:index</default>
    </route>
</routes>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('hello', new Route('/hello/{name}', array(
    '_controller' => 'AcmeHelloBundle:Hello:index',
)));

return $collection;
```

The routing consists of two basic pieces: the `pattern`, which is the URL that this route will match, and a `defaults` array, which specifies the controller that should be executed. The placeholder syntax in the pattern (`{name}`) is a wildcard. It means that `/hello/Ryan`, `/hello/Fabien` or any other similar URL will match this route. The `{name}` placeholder parameter will also be passed to the controller so that you can use its value to personally greet the user.

Note: The routing system has many more great features for creating flexible and powerful URL structures in your application. For more details, see the chapter all about Routing.

Step 2: Create the Controller

When a URL such as `/hello/Ryan` is handled by the application, the `hello` route is matched and the `AcmeHelloBundle:Hello:index` controller is executed by the framework. The second step of the page-creation process is to create that controller.

The controller - `AcmeHelloBundle:Hello:index` is the *logical* name of the controller, and it maps to the `indexAction` method of a PHP class called `Acme\HelloBundle\Controller\Hello`. Start by creating this file inside your `AcmeHelloBundle`:

```
// src/Acme/HelloBundle/Controller/HelloController.php
namespace Acme\HelloBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class HelloController
{
}
```

In reality, the controller is nothing more than a PHP method that you create and Symfony executes. This is where your code uses information from the request to build and prepare the resource being requested. Except in some advanced cases, the end product of a controller is always the same: a Symfony2 `Response` object.

Create the `indexAction` method that Symfony will execute when the `hello` route is matched:

```
// src/Acme/HelloBundle/Controller/HelloController.php

// ...
class HelloController
{
    public function indexAction($name)
    {
        return new Response('<html><body>Hello '.$name.'!</body></html>');
    }
}
```

The controller is simple: it creates a new `Response` object, whose first argument is the content that should be used in the response (a small HTML page in this example).

Congratulations! After creating only a route and a controller, you already have a fully-functional page! If you've setup everything correctly, your application should greet you:

```
http://localhost/app_dev.php/hello/Ryan
```

Tip: You can also view your app in the "prod" *environment* by visiting:

```
http://localhost/app.php/hello/Ryan
```

If you get an error, it's likely because you need to clear your cache by running:

```
php app/console cache:clear --env=prod --no-debug
```

An optional, but common, third step in the process is to create a template.

Note: Controllers are the main entry point for your code and a key ingredient when creating pages. Much more information can be found in the Controller Chapter.

Optional Step 3: Create the Template

Templates allows you to move all of the presentation (e.g. HTML code) into a separate file and reuse different portions of the page layout. Instead of writing the HTML inside the controller, render a template instead:

```
1  // src/Acme/HelloBundle/Controller/HelloController.php
2  namespace Acme\HelloBundle\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6  class HelloController extends Controller
7  {
8      public function indexAction($name)
9      {
10         return $this->render('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));
11
12         // render a PHP template instead
13         // return $this->render('AcmeHelloBundle:Hello:index.html.php', array('name' => $name));
```

```
14      }
15  }
```

Note: In order to use the `render()` method, your controller must extend the `Symfony\Bundle\FrameworkBundle\Controller\Controller` class (API docs: `Symfony\Bundle\FrameworkBundle\Controller\Controller`), which adds shortcuts for tasks that are common inside controllers. This is done in the above example by adding the `use` statement on line 4 and then extending `Controller` on line 6.

The `render()` method creates a `Response` object filled with the content of the given, rendered template. Like any other controller, you will ultimately return that `Response` object.

Notice that there are two different examples for rendering the template. By default, Symfony2 supports two different templating languages: classic PHP templates and the succinct but powerful Twig templates. Don't be alarmed - you're free to choose either or even both in the same project.

The controller renders the `AcmeHelloBundle:Hello:index.html.twig` template, which uses the following naming convention:

BundleName:**ControllerName**:**TemplateName**

This is the *logical* name of the template, which is mapped to a physical location using the following convention.

/path/to/BundleName/Resources/views/**ControllerName/TemplateName**

In this case, `AcmeHelloBundle` is the bundle name, `Hello` is the controller, and `index.html.twig` the template:

- *Twig*

```
1  {# src/Acme/HelloBundle/Resources/views/Hello/index.html.twig #}
2  {% extends '::base.html.twig' %}
3
4  {% block body %}
5      Hello {{ name }}!
6  {% endblock %}
```

- *PHP*

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php $view->extend('::base.html.php') ?>

Hello <?php echo $view->escape($name) ?>!
```

Let's step through the Twig template line-by-line:

- *line 2*: The `extends` token defines a parent template. The template explicitly defines a layout file inside of which it will be placed.
- *line 4*: The `block` token says that everything inside should be placed inside a block called `body`. As you'll see, it's the responsibility of the parent template (`base.html.twig`) to ultimately render the block called `body`.

The parent template, `::base.html.twig`, is missing both the **BundleName** and **ControllerName** portions of its name (hence the double colon (`::`) at the beginning). This means that the template lives outside of the bundles and in the `app` directory:

- *Twig*

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
```

```
        <head>
            <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
            <title>{% block title %}Welcome!{% endblock %}</title>
            {% block stylesheets %}{% endblock %}
            <link rel="shortcut icon" href="{{ asset('favicon.ico') }}" />
        </head>
        <body>
            {% block body %}{% endblock %}
            {% block javascripts %}{% endblock %}
        </body>
    </html>
```

- *PHP*

```
    <!-- app/Resources/views/base.html.php -->
    <!DOCTYPE html>
    <html>
        <head>
            <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
            <title><?php $view['slots']->output('title', 'Welcome!') ?></title>
            <?php $view['slots']->output('stylesheets') ?>
            <link rel="shortcut icon" href="<?php echo $view['assets']->getUrl('favicon.ico') ?>" />
        </head>
        <body>
            <?php $view['slots']->output('_content') ?>
            <?php $view['slots']->output('stylesheets') ?>
        </body>
    </html>
```

The base template file defines the HTML layout and renders the `body` block that you defined in the `index.html.twig` template. It also renders a `title` block, which you could choose to define in the `index.html.twig` template. Since you did not define the `title` block in the child template, it defaults to "Welcome!".

Templates are a powerful way to render and organize the content for your page. A template can render anything, from HTML markup, to CSS code, or anything else that the controller may need to return.

In the lifecycle of handling a request, the templating engine is simply an optional tool. Recall that the goal of each controller is to return a `Response` object. Templates are a powerful, but optional, tool for creating the content for that `Response` object.

The Directory Structure

After just a few short sections, you already understand the philosophy behind creating and rendering pages in Symfony2. You've also already begun to see how Symfony2 projects are structured and organized. By the end of this section, you'll know where to find and put different types of files and why.

Though entirely flexible, by default, each Symfony application has the same basic and recommended directory structure:

- `app/`: This directory contains the application configuration;
- `src/`: All the project PHP code is stored under this directory;
- `vendor/`: Any vendor libraries are placed here by convention;
- `web/`: This is the web root directory and contains any publicly accessible files;

The Web Directory

The web root directory is the home of all public and static files including images, stylesheets, and JavaScript files. It is also where each front controller lives:

```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

The front controller file (`app.php` in this example) is the actual PHP file that's executed when using a Symfony2 application and its job is to use a Kernel class, `AppKernel`, to bootstrap the application.

Tip: Having a front controller means different and more flexible URLs than are used in a typical flat PHP application. When using a front controller, URLs are formatted in the following way:

```
http://localhost/app.php/hello/Ryan
```

The front controller, `app.php`, is executed and the "internal:" URL `/hello/Ryan` is routed internally using the routing configuration. By using Apache `mod_rewrite` rules, you can force the `app.php` file to be executed without needing to specify it in the URL:

```
http://localhost/hello/Ryan
```

Though front controllers are essential in handling every request, you'll rarely need to modify or even think about them. We'll mention them again briefly in the *Environments* section.

The Application (`app`) Directory

As you saw in the front controller, the `AppKernel` class is the main entry point of the application and is responsible for all configuration. As such, it is stored in the `app/` directory.

This class must implement two methods that define everything that Symfony needs to know about your application. You don't even need to worry about these methods when starting - Symfony fills them in for you with sensible defaults.

- `registerBundles()`: Returns an array of all bundles needed to run the application (see *The Bundle System*);
- `registerContainerConfiguration()`: Loads the main application configuration resource file (see the *Application Configuration* section).

In day-to-day development, you'll mostly use the `app/` directory to modify configuration and routing files in the `app/config/` directory (see *Application Configuration*). It also contains the application cache directory (`app/cache`), a log directory (`app/logs`) and a directory for application-level resource files, such as templates (`app/Resources`). You'll learn more about each of these directories in later chapters.

Autoloading

When Symfony is loading, a special file - `app/autoload.php` - is included. This file is responsible for configuring the autoloader, which will autoload your application files from the `src/` directory and third-party libraries from the `vendor/` directory.
Because of the autoloader, you never need to worry about using `include` or `require` statements. Instead, Symfony2 uses the namespace of a class to determine its location and automatically includes the file on your behalf the instant you need a class.
The autoloader is already configured to look in the `src/` directory for any of your PHP classes. For autoloading to work, the class name and path to the file have to follow the same pattern:

```
Class Name:
    Acme\HelloBundle\Controller\HelloController
Path:
    src/Acme/HelloBundle/Controller/HelloController.php
```

Typically, the only time you'll need to worry about the `app/autoload.php` file is when you're including a new third-party library in the `vendor/` directory. For more information on autoloading, see How to autoload Classes.

The Source (`src`) Directory

Put simply, the `src/` directory contains all of the actual code (PHP code, templates, configuration files, stylesheets, etc) that drives *your* application. When developing, the vast majority of your work will be done inside one or more bundles that you create in this directory.

But what exactly is a bundle?

The Bundle System

A bundle is similar to a plugin in other software, but even better. The key difference is that *everything* is a bundle in Symfony2, including both the core framework functionality and the code written for your application. Bundles are first-class citizens in Symfony2. This gives you the flexibility to use pre-built features packaged in third-party bundles or to distribute your own bundles. It makes it easy to pick and choose which features to enable in your application and to optimize them the way you want.

Note: While you'll learn the basics here, an entire cookbook entry is devoted to the organization and best practices of bundles.

A bundle is simply a structured set of files within a directory that implement a single feature. You might create a `BlogBundle`, a `ForumBundle` or a bundle for user management (many of these exist already as open source bundles). Each directory contains everything related to that feature, including PHP files, templates, stylesheets, JavaScripts, tests and anything else. Every aspect of a feature exists in a bundle and every feature lives in a bundle.

An application is made up of bundles as defined in the `registerBundles()` method of the `AppKernel` class:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
```

```
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
    );

    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
        $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }

    return $bundles;
}
```

With the `registerBundles()` method, you have total control over which bundles are used by your application (including the core Symfony bundles).

Tip: A bundle can live *anywhere* as long as it can be autoloaded (via the autoloader configured at `app/autoload.php`).

Creating a Bundle

The Symfony Standard Edition comes with a handy task that creates a fully-functional bundle for you. Of course, creating a bundle by hand is pretty easy as well.

To show you how simple the bundle system is, create a new bundle called `AcmeTestBundle` and enable it.

Tip: The `Acme` portion is just a dummy name that should be replaced by some "vendor" name that represents you or your organization (e.g. `ABCTestBundle` for some company named `ABC`).

Start by creating a `src/Acme/TestBundle/` directory and adding a new file called `AcmeTestBundle.php`:

```
// src/Acme/TestBundle/AcmeTestBundle.php
namespace Acme\TestBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class AcmeTestBundle extends Bundle
{
}
```

Tip: The name `AcmeTestBundle` follows the standard *Bundle naming conventions*. You could also choose to shorten the name of the bundle to simply `TestBundle` by naming this class `TestBundle` (and naming the file `TestBundle.php`).

This empty class is the only piece you need to create the new bundle. Though commonly empty, this class is powerful and can be used to customize the behavior of the bundle.

Now that you've created the bundle, enable it via the `AppKernel` class:

```
// app/AppKernel.php
public function registerBundles()
```

```
{
    $bundles = array(
        // ...

        // register your bundles
        new Acme\TestBundle\AcmeTestBundle(),
    );
    // ...

    return $bundles;
}
```

And while it doesn't do anything yet, `AcmeTestBundle` is now ready to be used.

And as easy as this is, Symfony also provides a command-line interface for generating a basic bundle skeleton:

```
php app/console generate:bundle --namespace=Acme/TestBundle
```

The bundle skeleton generates with a basic controller, template and routing resource that can be customized. You'll learn more about Symfony2's command-line tools later.

Tip: Whenever creating a new bundle or using a third-party bundle, always make sure the bundle has been enabled in `registerBundles()`. When using the `generate:bundle` command, this is done for you.

Bundle Directory Structure

The directory structure of a bundle is simple and flexible. By default, the bundle system follows a set of conventions that help to keep code consistent between all Symfony2 bundles. Take a look at `AcmeHelloBundle`, as it contains some of the most common elements of a bundle:

- `Controller/` contains the controllers of the bundle (e.g. `HelloController.php`);
- `Resources/config/` houses configuration, including routing configuration (e.g. `routing.yml`);
- `Resources/views/` holds templates organized by controller name (e.g. `Hello/index.html.twig`);
- `Resources/public/` contains web assets (images, stylesheets, etc) and is copied or symbolically linked into the project `web/` directory via the `assets:install` console command;
- `Tests/` holds all tests for the bundle.

A bundle can be as small or large as the feature it implements. It contains only the files you need and nothing else.

As you move through the book, you'll learn how to persist objects to a database, create and validate forms, create translations for your application, write tests and much more. Each of these has their own place and role within the bundle.

Application Configuration

An application consists of a collection of bundles representing all of the features and capabilities of your application. Each bundle can be customized via configuration files written in YAML, XML or PHP. By default, the main configuration file lives in the `app/config/` directory and is called either `config.yml`, `config.xml` or `config.php` depending on which format you prefer:

- *YAML*

```yaml
# app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }

framework:
    secret:          %secret%
    charset:         UTF-8
    router:          { resource: "%kernel.root_dir%/config/routing.yml" }
    # ...

# Twig Configuration
twig:
    debug:            %kernel.debug%
    strict_variables: %kernel.debug%

# ...
```

- *XML*

```xml
<!-- app/config/config.xml -->
<imports>
    <import resource="parameters.yml" />
    <import resource="security.yml" />
</imports>

<framework:config charset="UTF-8" secret="%secret%">
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
    <!-- ... -->
</framework:config>

<!-- Twig Configuration -->
<twig:config debug="%kernel.debug%" strict-variables="%kernel.debug%" />

<!-- ... -->
```

- *PHP*

```php
$this->import('parameters.yml');
$this->import('security.yml');

$container->loadFromExtension('framework', array(
    'secret'          => '%secret%',
    'charset'         => 'UTF-8',
    'router'          => array('resource' => '%kernel.root_dir%/config/routing.php'),
    // ...
    ),
));

// Twig Configuration
$container->loadFromExtension('twig', array(
    'debug'            => '%kernel.debug%',
    'strict_variables' => '%kernel.debug%',
));

// ...
```

Note: You'll learn exactly how to load each file/format in the next section *Environments*.

Each top-level entry like `framework` or `twig` defines the configuration for a particular bundle. For example, the `framework` key defines the configuration for the core Symfony `FrameworkBundle` and includes configuration for the routing, templating, and other core systems.

For now, don't worry about the specific configuration options in each section. The configuration file ships with sensible defaults. As you read more and explore each part of Symfony2, you'll learn about the specific configuration options of each feature.

Configuration Formats

Throughout the chapters, all configuration examples will be shown in all three formats (YAML, XML and PHP). Each has its own advantages and disadvantages. The choice of which to use is up to you:

- *YAML*: Simple, clean and readable;
- *XML*: More powerful than YAML at times and supports IDE autocompletion;
- *PHP*: Very powerful but less readable than standard configuration formats.

Default Configuration Dump

New in version 2.1: The `config:dump-reference` command was added in Symfony 2.1

You can dump the default configuration for a bundle in yaml to the console using the `config:dump-reference` command. Here is an example of dumping the default FrameworkBundle configuration:

```
app/console config:dump-reference FrameworkBundle
```

Note: See the cookbook article: How to expose a Semantic Configuration for a Bundle for information on adding configuration for your own bundle.

Environments

An application can run in various environments. The different environments share the same PHP code (apart from the front controller), but use different configuration. For instance, a `dev` environment will log warnings and errors, while a `prod` environment will only log errors. Some files are rebuilt on each request in the `dev` environment (for the developer's convenience), but cached in the `prod` environment. All environments live together on the same machine and execute the same application.

A Symfony2 project generally begins with three environments (`dev`, `test` and `prod`), though creating new environments is easy. You can view your application in different environments simply by changing the front controller in your browser. To see the application in the `dev` environment, access the application via the development front controller:

```
http://localhost/app_dev.php/hello/Ryan
```

If you'd like to see how your application will behave in the production environment, call the `prod` front controller instead:

```
http://localhost/app.php/hello/Ryan
```

Since the `prod` environment is optimized for speed; the configuration, routing and Twig templates are compiled into flat PHP classes and cached. When viewing changes in the `prod` environment, you'll need to clear these cached files and allow them to rebuild:

```
php app/console cache:clear --env=prod --no-debug
```

Note: If you open the `web/app.php` file, you'll find that it's configured explicitly to use the `prod` environment:

```
$kernel = new AppKernel('prod', false);
```

You can create a new front controller for a new environment by copying this file and changing `prod` to some other value.

Note: The `test` environment is used when running automated tests and cannot be accessed directly through the browser. See the testing chapter for more details.

Environment Configuration

The `AppKernel` class is responsible for actually loading the configuration file of your choice:

```
// app/AppKernel.php
public function registerContainerConfiguration(LoaderInterface $loader)
{
    $loader->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
}
```

You already know that the `.yml` extension can be changed to `.xml` or `.php` if you prefer to use either XML or PHP to write your configuration. Notice also that each environment loads its own configuration file. Consider the configuration file for the `dev` environment.

- *YAML*

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

framework:
    router:   { resource: "%kernel.root_dir%/config/routing_dev.yml" }
    profiler: { only_exceptions: false }

# ...
```

- *XML*

```
<!-- app/config/config_dev.xml -->
<imports>
    <import resource="config.xml" />
</imports>

<framework:config>
    <framework:router resource="%kernel.root_dir%/config/routing_dev.xml" />
    <framework:profiler only-exceptions="false" />
</framework:config>

<!-- ... -->
```

- *PHP*

```
// app/config/config_dev.php
$loader->import('config.php');

$container->loadFromExtension('framework', array(
    'router'   => array('resource' => '%kernel.root_dir%/config/routing_dev.php'),
    'profiler' => array('only-exceptions' => false),
```

```
    ));

    // ...
```

The `imports` key is similar to a PHP `include` statement and guarantees that the main configuration file (`config.yml`) is loaded first. The rest of the file tweaks the default configuration for increased logging and other settings conducive to a development environment.

Both the `prod` and `test` environments follow the same model: each environment imports the base configuration file and then modifies its configuration values to fit the needs of the specific environment. This is just a convention, but one that allows you to reuse most of your configuration and customize just pieces of it between environments.

Summary

Congratulations! You've now seen every fundamental aspect of Symfony2 and have hopefully discovered how easy and flexible it can be. And while there are *a lot* of features still to come, be sure to keep the following basic points in mind:

- creating a page is a three-step process involving a **route**, a **controller** and (optionally) a **template**.
- each project contains just a few main directories: `web/` (web assets and the front controllers), `app/` (configuration), `src/` (your bundles), and `vendor/` (third-party code) (there's also a `bin/` directory that's used to help updated vendor libraries);
- each feature in Symfony2 (including the Symfony2 framework core) is organized into a *bundle*, which is a structured set of files for that feature;
- the **configuration** for each bundle lives in the `app/config` directory and can be specified in YAML, XML or PHP;
- each **environment** is accessible via a different front controller (e.g. `app.php` and `app_dev.php`) and loads a different configuration file.

From here, each chapter will introduce you to more and more powerful tools and advanced concepts. The more you know about Symfony2, the more you'll appreciate the flexibility of its architecture and the power it gives you to rapidly develop applications.

2.1.5 Controller

A controller is a PHP function you create that takes information from the HTTP request and constructs and returns an HTTP response (as a Symfony2 `Response` object). The response could be an HTML page, an XML document, a serialized JSON array, an image, a redirect, a 404 error or anything else you can dream up. The controller contains whatever arbitrary logic *your application* needs to render the content of a page.

To see how simple this is, let's look at a Symfony2 controller in action. The following controller would render a page that simply prints `Hello world!`:

```
use Symfony\Component\HttpFoundation\Response;

public function helloAction()
{
    return new Response('Hello world!');
}
```

The goal of a controller is always the same: create and return a `Response` object. Along the way, it might read information from the request, load a database resource, send an email, or set information on the user's session. But in all cases, the controller will eventually return the `Response` object that will be delivered back to the client.

There's no magic and no other requirements to worry about! Here are a few common examples:

- *Controller A* prepares a `Response` object representing the content for the homepage of the site.
- *Controller B* reads the `slug` parameter from the request to load a blog entry from the database and create a `Response` object displaying that blog. If the `slug` can't be found in the database, it creates and returns a `Response` object with a 404 status code.
- *Controller C* handles the form submission of a contact form. It reads the form information from the request, saves the contact information to the database and emails the contact information to the webmaster. Finally, it creates a `Response` object that redirects the client's browser to the contact form "thank you" page.

Requests, Controller, Response Lifecycle

Every request handled by a Symfony2 project goes through the same simple lifecycle. The framework takes care of the repetitive tasks and ultimately executes a controller, which houses your custom application code:

1. Each request is handled by a single front controller file (e.g. `app.php` or `app_dev.php`) that bootstraps the application;
2. The `Router` reads information from the request (e.g. the URI), finds a route that matches that information, and reads the `_controller` parameter from the route;
3. The controller from the matched route is executed and the code inside the controller creates and returns a `Response` object;
4. The HTTP headers and content of the `Response` object are sent back to the client.

Creating a page is as easy as creating a controller (#3) and making a route that maps a URL to that controller (#2).

Note: Though similarly named, a "front controller" is different from the "controllers" we'll talk about in this chapter. A front controller is a short PHP file that lives in your web directory and through which all requests are directed. A typical application will have a production front controller (e.g. `app.php`) and a development front controller (e.g. `app_dev.php`). You'll likely never need to edit, view or worry about the front controllers in your application.

A Simple Controller

While a controller can be any PHP callable (a function, method on an object, or a `Closure`), in Symfony2, a controller is usually a single method inside a controller object. Controllers are also called *actions*.

```
1  // src/Acme/HelloBundle/Controller/HelloController.php
2
3  namespace Acme\HelloBundle\Controller;
4  use Symfony\Component\HttpFoundation\Response;
5
6  class HelloController
7  {
8      public function indexAction($name)
9      {
10       return new Response('<html><body>Hello '.$name.'!</body></html>');
11     }
12 }
```

Tip: Note that the *controller* is the `indexAction` method, which lives inside a *controller class* (`HelloController`). Don't be confused by the naming: a *controller class* is simply a convenient way to group several controllers/actions together. Typically, the controller class will house several controllers/actions (e.g. `updateAction`, `deleteAction`, etc).

This controller is pretty straightforward, but let's walk through it:

- *line 3*: Symfony2 takes advantage of PHP 5.3 namespace functionality to namespace the entire controller class. The `use` keyword imports the `Response` class, which our controller must return.
- *line 6*: The class name is the concatenation of a name for the controller class (i.e. `Hello`) and the word `Controller`. This is a convention that provides consistency to controllers and allows them to be referenced only by the first part of the name (i.e. `Hello`) in the routing configuration.
- *line 8*: Each action in a controller class is suffixed with `Action` and is referenced in the routing configuration by the action's name (`index`). In the next section, you'll create a route that maps a URI to this action. You'll learn how the route's placeholders (`{name}`) become arguments to the action method (`$name`).
- *line 10*: The controller creates and returns a `Response` object.

Mapping a URL to a Controller

The new controller returns a simple HTML page. To actually view this page in your browser, you need to create a route, which maps a specific URL pattern to the controller:

- *YAML*

```
# app/config/routing.yml
hello:
    pattern:      /hello/{name}
    defaults:     { _controller: AcmeHelloBundle:Hello:index }
```

- *XML*

```
<!-- app/config/routing.xml -->
<route id="hello" pattern="/hello/{name}">
    <default key="_controller">AcmeHelloBundle:Hello:index</default>
</route>
```

- *PHP*

```
// app/config/routing.php
$collection->add('hello', new Route('/hello/{name}', array(
    '_controller' => 'AcmeHelloBundle:Hello:index',
)));
```

Going to `/hello/ryan` now executes the `HelloController::indexAction()` controller and passes in `ryan` for the `$name` variable. Creating a "page" means simply creating a controller method and associated route.

Notice the syntax used to refer to the controller: `AcmeHelloBundle:Hello:index`. Symfony2 uses a flexible string notation to refer to different controllers. This is the most common syntax and tells Symfony2 to look for a controller class called `HelloController` inside a bundle named `AcmeHelloBundle`. The method `indexAction()` is then executed.

For more details on the string format used to reference different controllers, see *Controller Naming Pattern*.

Note: This example places the routing configuration directly in the `app/config/` directory. A better way to organize your routes is to place each route in the bundle it belongs to. For more information on this, see *Including External Routing Resources*.

Tip: You can learn much more about the routing system in the Routing chapter.

Route Parameters as Controller Arguments

You already know that the `_controller` parameter `AcmeHelloBundle:Hello:index` refers to a `HelloController::indexAction()` method that lives inside the `AcmeHelloBundle` bundle. What's more interesting is the arguments that are passed to that method:

```
<?php
// src/Acme/HelloBundle/Controller/HelloController.php

namespace Acme\HelloBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HelloController extends Controller
{
    public function indexAction($name)
    {
      // ...
    }
}
```

The controller has a single argument, `$name`, which corresponds to the `{name}` parameter from the matched route (`ryan` in our example). In fact, when executing your controller, Symfony2 matches each argument of the controller with a parameter from the matched route. Take the following example:

- *YAML*

```
# app/config/routing.yml
hello:
    pattern:      /hello/{first_name}/{last_name}
    defaults:     { _controller: AcmeHelloBundle:Hello:index, color: green }
```

- *XML*

```
<!-- app/config/routing.xml -->
<route id="hello" pattern="/hello/{first_name}/{last_name}">
    <default key="_controller">AcmeHelloBundle:Hello:index</default>
    <default key="color">green</default>
</route>
```

- *PHP*

```
// app/config/routing.php
$collection->add('hello', new Route('/hello/{first_name}/{last_name}', array(
    '_controller' => 'AcmeHelloBundle:Hello:index',
    'color'       => 'green',
)));
```

The controller for this can take several arguments:

```
public function indexAction($first_name, $last_name, $color)
{
    // ...
}
```

Notice that both placeholder variables (`{first_name}`, `{last_name}`) as well as the default `color` variable are available as arguments in the controller. When a route is matched, the placeholder variables are merged with the `defaults` to make one array that's available to your controller.

Mapping route parameters to controller arguments is easy and flexible. Keep the following guidelines in mind while you develop.

- **The order of the controller arguments does not matter**

  Symfony is able to match the parameter names from the route to the variable names in the controller method's signature. In other words, it realizes that the `{last_name}` parameter matches up with the `$last_name` argument. The arguments of the controller could be totally reordered and still work perfectly:

```
public function indexAction($last_name, $color, $first_name)
{
    // ..
}
```

- **Each required controller argument must match up with a routing parameter**

  The following would throw a `RuntimeException` because there is no `foo` parameter defined in the route:

```
public function indexAction($first_name, $last_name, $color, $foo)
{
    // ..
}
```

  Making the argument optional, however, is perfectly ok. The following example would not throw an exception:

```
public function indexAction($first_name, $last_name, $color, $foo = 'bar')
{
    // ..
}
```

- **Not all routing parameters need to be arguments on your controller**

  If, for example, the `last_name` weren't important for your controller, you could omit it entirely:

```
public function indexAction($first_name, $color)
{
    // ..
}
```

Tip: Every route also has a special `_route` parameter, which is equal to the name of the route that was matched (e.g. `hello`). Though not usually useful, this is equally available as a controller argument.

The `Request` as a Controller Argument

For convenience, you can also have Symfony pass you the `Request` object as an argument to your controller. This is especially convenient when you're working with forms, for example:

```
use Symfony\Component\HttpFoundation\Request;

public function updateAction(Request $request)
{
    $form = $this->createForm(...);

    $form->bindRequest($request);
    // ...
}
```

The Base Controller Class

For convenience, Symfony2 comes with a base `Controller` class that assists with some of the most common controller tasks and gives your controller class access to any resource it might need. By extending this `Controller` class, you can take advantage of several helper methods.

Add the `use` statement atop the `Controller` class and then modify the `HelloController` to extend it:

```
// src/Acme/HelloBundle/Controller/HelloController.php

namespace Acme\HelloBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class HelloController extends Controller
{
    public function indexAction($name)
    {
      return new Response('<html><body>Hello '.$name.'!</body></html>');
    }
}
```

This doesn't actually change anything about how your controller works. In the next section, you'll learn about the helper methods that the base controller class makes available. These methods are just shortcuts to using core Symfony2 functionality that's available to you with or without the use of the base `Controller` class. A great way to see the core functionality in action is to look in the `Symfony\Bundle\FrameworkBundle\Controller\Controller` class itself.

Tip: Extending the base class is *optional* in Symfony; it contains useful shortcuts but nothing mandatory. You can also extend `Symfony\Component\DependencyInjection\ContainerAware`. The service container object will then be accessible via the `container` property.

Note: You can also define your Controllers as Services.

Common Controller Tasks

Though a controller can do virtually anything, most controllers will perform the same basic tasks over and over again. These tasks, such as redirecting, forwarding, rendering templates and accessing core services, are very easy to manage in Symfony2.

Redirecting

If you want to redirect the user to another page, use the `redirect()` method:

```
public function indexAction()
{
    return $this->redirect($this->generateUrl('homepage'));
}
```

The `generateUrl()` method is just a helper function that generates the URL for a given route. For more information, see the Routing chapter.

By default, the `redirect()` method performs a 302 (temporary) redirect. To perform a 301 (permanent) redirect, modify the second argument:

```
public function indexAction()
{
    return $this->redirect($this->generateUrl('homepage'), 301);
}
```

Tip: The `redirect()` method is simply a shortcut that creates a `Response` object that specializes in redirecting the user. It's equivalent to:

```
use Symfony\Component\HttpFoundation\RedirectResponse;

return new RedirectResponse($this->generateUrl('homepage'));
```

Forwarding

You can also easily forward to another controller internally with the `forward()` method. Instead of redirecting the user's browser, it makes an internal sub-request, and calls the specified controller. The `forward()` method returns the `Response` object that's returned from that controller:

```
public function indexAction($name)
{
    $response = $this->forward('AcmeHelloBundle:Hello:fancy', array(
        'name'  => $name,
        'color' => 'green'
    ));

    // further modify the response or return it directly

    return $response;
}
```

Notice that the *forward()* method uses the same string representation of the controller used in the routing configuration. In this case, the target controller class will be `HelloController` inside some `AcmeHelloBundle`. The array passed to the method becomes the arguments on the resulting controller. This same interface is used when embedding controllers into templates (see *Embedding Controllers*). The target controller method should look something like the following:

```
public function fancyAction($name, $color)
{
    // ... create and return a Response object
}
```

And just like when creating a controller for a route, the order of the arguments to `fancyAction` doesn't matter. Symfony2 matches the index key names (e.g. `name`) with the method argument names (e.g. `$name`). If you change the order of the arguments, Symfony2 will still pass the correct value to each variable.

Tip: Like other base `Controller` methods, the `forward` method is just a shortcut for core Symfony2 functionality. A forward can be accomplished directly via the `http_kernel` service. A forward returns a `Response` object:

```
$httpKernel = $this->container->get('http_kernel');
$response = $httpKernel->forward('AcmeHelloBundle:Hello:fancy', array(
    'name'  => $name,
    'color' => 'green',
));
```

Rendering Templates

Though not a requirement, most controllers will ultimately render a template that's responsible for generating the HTML (or other format) for the controller. The `renderView()` method renders a template and returns its content. The content from the template can be used to create a `Response` object:

```
$content = $this->renderView('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));

return new Response($content);
```

This can even be done in just one step with the `render()` method, which returns a `Response` object containing the content from the template:

```
return $this->render('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));
```

In both cases, the `Resources/views/Hello/index.html.twig` template inside the `AcmeHelloBundle` will be rendered.

The Symfony templating engine is explained in great detail in the Templating chapter.

Tip: The `renderView` method is a shortcut to direct use of the `templating` service. The `templating` service can also be used directly:

```
$templating = $this->get('templating');
$content = $templating->render('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));
```

Accessing other Services

When extending the base controller class, you can access any Symfony2 service via the `get()` method. Here are several common services you might need:

```
$request = $this->getRequest();

$templating = $this->get('templating');

$router = $this->get('router');

$mailer = $this->get('mailer');
```

There are countless other services available and you are encouraged to define your own. To list all available services, use the `container:debug` console command:

```
php app/console container:debug
```

For more information, see the Service Container chapter.

Managing Errors and 404 Pages

When things are not found, you should play well with the HTTP protocol and return a 404 response. To do this, you'll throw a special type of exception. If you're extending the base controller class, do the following:

```
public function indexAction()
{
    $product = // retrieve the object from database
    if (!$product) {
        throw $this->createNotFoundException('The product does not exist');
```

```
    }

    return $this->render(...);
}
```

The `createNotFoundException()` method creates a special `NotFoundHttpException` object, which ultimately triggers a 404 HTTP response inside Symfony.

Of course, you're free to throw any `Exception` class in your controller - Symfony2 will automatically return a 500 HTTP response code.

```
throw new \Exception('Something went wrong!');
```

In every case, a styled error page is shown to the end user and a full debug error page is shown to the developer (when viewing the page in debug mode). Both of these error pages can be customized. For details, read the "How to customize Error Pages" cookbook recipe.

Managing the Session

Symfony2 provides a nice session object that you can use to store information about the user (be it a real person using a browser, a bot, or a web service) between requests. By default, Symfony2 stores the attributes in a cookie by using the native PHP sessions.

Storing and retrieving information from the session can be easily achieved from any controller:

```
$session = $this->getRequest()->getSession();

// store an attribute for reuse during a later user request
$session->set('foo', 'bar');

// in another controller for another request
$foo = $session->get('foo');

// use a default value of the key doesn't exist
$filters = $session->set('filters', array());
```

These attributes will remain on the user for the remainder of that user's session.

Flash Messages

You can also store small messages that will be stored on the user's session for exactly one additional request. This is useful when processing a form: you want to redirect and have a special message shown on the *next* request. These types of messages are called "flash" messages.

For example, imagine you're processing a form submit:

```
public function updateAction()
{
    $form = $this->createForm(...);

    $form->bindRequest($this->getRequest());
    if ($form->isValid()) {
        // do some sort of processing

        $this->get('session')->setFlash('notice', 'Your changes were saved!');

        return $this->redirect($this->generateUrl(...));
```

```
    }

    return $this->render(...);
}
```

After processing the request, the controller sets a `notice` flash message and then redirects. The name (`notice`) isn't significant - it's just what you're using to identify the type of the message.

In the template of the next action, the following code could be used to render the `notice` message:

- *Twig*

```
{% if app.session.hasFlash('notice') %}
    <div class="flash-notice">
        {{ app.session.flash('notice') }}
    </div>
{% endif %}
```

- *PHP*

```
<?php if ($view['session']->hasFlash('notice')): ?>
    <div class="flash-notice">
        <?php echo $view['session']->getFlash('notice') ?>
    </div>
<?php endif; ?>
```

By design, flash messages are meant to live for exactly one request (they're "gone in a flash"). They're designed to be used across redirects exactly as you've done in this example.

The Response Object

The only requirement for a controller is to return a `Response` object. The `Symfony\Component\HttpFoundation\Response` class is a PHP abstraction around the HTTP response - the text-based message filled with HTTP headers and content that's sent back to the client:

```
// create a simple Response with a 200 status code (the default)
$response = new Response('Hello '.$name, 200);

// create a JSON-response with a 200 status code
$response = new Response(json_encode(array('name' => $name)));
$response->headers->set('Content-Type', 'application/json');
```

Tip: The `headers` property is a `Symfony\Component\HttpFoundation\HeaderBag` object with several useful methods for reading and mutating the `Response` headers. The header names are normalized so that using `Content-Type` is equivalent to `content-type` or even `content_type`.

The Request Object

Besides the values of the routing placeholders, the controller also has access to the `Request` object when extending the base `Controller` class:

```
$request = $this->getRequest();

$request->isXmlHttpRequest(); // is it an Ajax request?

$request->getPreferredLanguage(array('en', 'fr'));
```

```
$request->query->get('page'); // get a $_GET parameter

$request->request->get('page'); // get a $_POST parameter
```

Like the `Response` object, the request headers are stored in a `HeaderBag` object and are easily accessible.

Final Thoughts

Whenever you create a page, you'll ultimately need to write some code that contains the logic for that page. In Symfony, this is called a controller, and it's a PHP function that can do anything it needs in order to return the final `Response` object that will be returned to the user.

To make life easier, you can choose to extend a base `Controller` class, which contains shortcut methods for many common controller tasks. For example, since you don't want to put HTML code in your controller, you can use the `render()` method to render and return the content from a template.

In other chapters, you'll see how the controller can be used to persist and fetch objects from a database, process form submissions, handle caching and more.

Learn more from the Cookbook

- How to customize Error Pages
- How to define Controllers as Services

2.1.6 Routing

Beautiful URLs are an absolute must for any serious web application. This means leaving behind ugly URLs like `index.php?article_id=57` in favor of something like `/read/intro-to-symfony`.

Having flexibility is even more important. What if you need to change the URL of a page from `/blog` to `/news`? How many links should you need to hunt down and update to make the change? If you're using Symfony's router, the change is simple.

The Symfony2 router lets you define creative URLs that you map to different areas of your application. By the end of this chapter, you'll be able to:

- Create complex routes that map to controllers
- Generate URLs inside templates and controllers
- Load routing resources from bundles (or anywhere else)
- Debug your routes

Routing in Action

A *route* is a map from a URL pattern to a controller. For example, suppose you want to match any URL like `/blog/my-post` or `/blog/all-about-symfony` and send it to a controller that can look up and render that blog entry. The route is simple:

- *YAML*

```
    # app/config/routing.yml
    blog_show:
        pattern:   /blog/{slug}
        defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
    <!-- app/config/routing.xml -->
    <?xml version="1.0" encoding="UTF-8" ?>
    <routes xmlns="http://symfony.com/schema/routing"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

        <route id="blog_show" pattern="/blog/{slug}">
            <default key="_controller">AcmeBlogBundle:Blog:show</default>
        </route>
    </routes>
```

- *PHP*

```
    // app/config/routing.php
    use Symfony\Component\Routing\RouteCollection;
    use Symfony\Component\Routing\Route;

    $collection = new RouteCollection();
    $collection->add('blog_show', new Route('/blog/{slug}', array(
        '_controller' => 'AcmeBlogBundle:Blog:show',
    )));

    return $collection;
```

The pattern defined by the `blog_show` route acts like `/blog/*` where the wildcard is given the name `slug`. For the URL `/blog/my-blog-post`, the `slug` variable gets a value of `my-blog-post`, which is available for you to use in your controller (keep reading).

The `_controller` parameter is a special key that tells Symfony which controller should be executed when a URL matches this route. The `_controller` string is called the *logical name*. It follows a pattern that points to a specific PHP class and method:

```
// src/Acme/BlogBundle/Controller/BlogController.php

namespace Acme\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($slug)
    {
        $blog = // use the $slug varible to query the database

        return $this->render('AcmeBlogBundle:Blog:show.html.twig', array(
            'blog' => $blog,
        ));
    }
}
```

Congratulations! You've just created your first route and connected it to a controller. Now, when you visit `/blog/my-post`, the `showAction` controller will be executed and the `$slug` variable will be equal to `my-post`.

This is the goal of the Symfony2 router: to map the URL of a request to a controller. Along the way, you'll learn all sorts of tricks that make mapping even the most complex URLs easy.

Routing: Under the Hood

When a request is made to your application, it contains an address to the exact "resource" that the client is requesting. This address is called the URL, (or URI), and could be `/contact`, `/blog/read-me`, or anything else. Take the following HTTP request for example:

```
GET /blog/my-blog-post
```

The goal of the Symfony2 routing system is to parse this URL and determine which controller should be executed. The whole process looks like this:

1. The request is handled by the Symfony2 front controller (e.g. `app.php`);
2. The Symfony2 core (i.e. Kernel) asks the router to inspect the request;
3. The router matches the incoming URL to a specific route and returns information about the route, including the controller that should be executed;
4. The Symfony2 Kernel executes the controller, which ultimately returns a `Response` object.

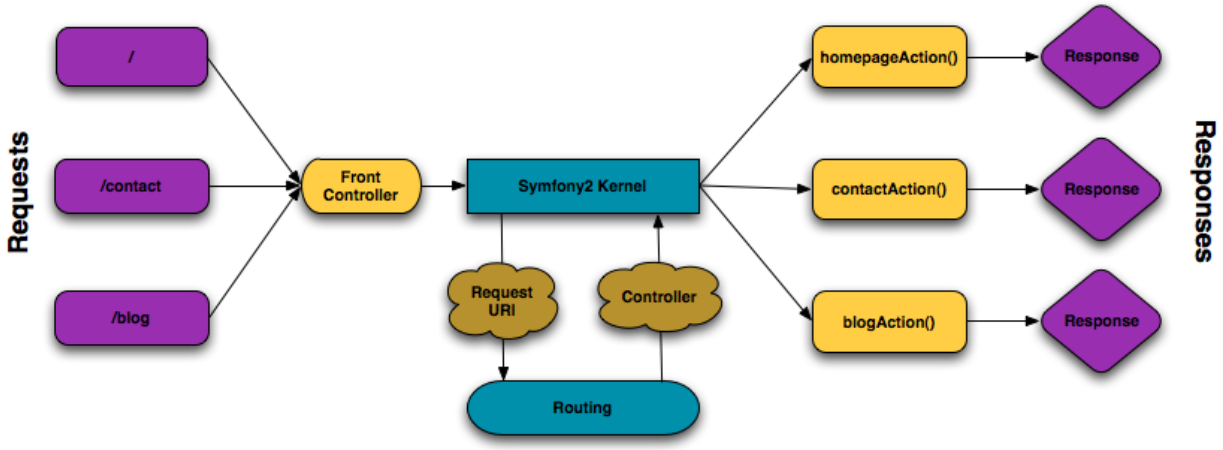


Fig. 2.2: The routing layer is a tool that translates the incoming URL into a specific controller to execute.

Creating Routes

Symfony loads all the routes for your application from a single routing configuration file. The file is usually `app/config/routing.yml`, but can be configured to be anything (including an XML or PHP file) via the application configuration file:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config ...>
    <!-- ... -->
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'router'        => array('resource' => '%kernel.root_dir%/config/routing.php'),
));
```

Tip: Even though all routes are loaded from a single file, it's common practice to include additional routing resources from inside the file. See the *Including External Routing Resources* section for more information.

Basic Route Configuration

Defining a route is easy, and a typical application will have lots of routes. A basic route consists of just two parts: the `pattern` to match and a `defaults` array:

- *YAML*

```
_welcome:
    pattern:   /
    defaults:  { _controller: AcmeDemoBundle:Main:homepage }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="_welcome" pattern="/">
        <default key="_controller">AcmeDemoBundle:Main:homepage</default>
    </route>

</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('_welcome', new Route('/', array(
    '_controller' => 'AcmeDemoBundle:Main:homepage',
)));

return $collection;
```

This route matches the homepage (`/`) and maps it to the `AcmeDemoBundle:Main:homepage` controller. The `_controller` string is translated by Symfony2 into an actual PHP function and executed. That process will be explained shortly in the *Controller Naming Pattern* section.

Routing with Placeholders

Of course the routing system supports much more interesting routes. Many routes will contain one or more named "wildcard" placeholders:

- *YAML*

```
blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog_show" pattern="/blog/{slug}">
        <default key="_controller">AcmeBlogBundle:Blog:show</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog_show', new Route('/blog/{slug}', array(
    '_controller' => 'AcmeBlogBundle:Blog:show',
)));

return $collection;
```

The pattern will match anything that looks like `/blog/*`. Even better, the value matching the `{slug}` placeholder will be available inside your controller. In other words, if the URL is `/blog/hello-world`, a `$slug` variable, with a value of `hello-world`, will be available in the controller. This can be used, for example, to load the blog post matching that string.

The pattern will *not*, however, match simply `/blog`. That's because, by default, all placeholders are required. This can be changed by adding a placeholder value to the `defaults` array.

Required and Optional Placeholders

To make things more exciting, add a new route that displays a list of all the available blog posts for this imaginary blog application:

- *YAML*

```
blog:
    pattern:   /blog
    defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

- *XML*

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
    </route>
</routes>
```

- *PHP*

```php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
)));

return $collection;
```

So far, this route is as simple as possible - it contains no placeholders and will only match the exact URL `/blog`. But what if you need this route to support pagination, where `/blog/2` displays the second page of blog entries? Update the route to have a new `{page}` placeholder:

- *YAML*

```yaml
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

- *XML*

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{page}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
    </route>
</routes>
```

- *PHP*

```php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
)));

return $collection;
```

Like the `{slug}` placeholder before, the value matching `{page}` will be available inside your controller. Its value can be used to determine which set of blog posts to display for the given page.

But hold on! Since placeholders are required by default, this route will no longer match on simply `/blog`. Instead, to see page 1 of the blog, you'd need to use the URL `/blog/1`! Since that's no way for a rich web app to behave, modify the route to make the `{page}` parameter optional. This is done by including it in the `defaults` collection:

- *YAML*

```
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{page}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="page">1</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'page' => 1,
)));

return $collection;
```

By adding `page` to the `defaults` key, the `{page}` placeholder is no longer required. The URL `/blog` will match this route and the value of the `page` parameter will be set to `1`. The URL `/blog/2` will also match, giving the `page` parameter a value of `2`. Perfect.

/blog	{page} = 1
/blog/1	{page} = 1
/blog/2	{page} = 2

Adding Requirements

Take a quick look at the routes that have been created so far:

- *YAML*

```
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }

blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{page}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="page">1</default>
    </route>

    <route id="blog_show" pattern="/blog/{slug}">
        <default key="_controller">AcmeBlogBundle:Blog:show</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'page' => 1,
)));

$collection->add('blog_show', new Route('/blog/{show}', array(
    '_controller' => 'AcmeBlogBundle:Blog:show',
)));

return $collection;
```

Can you spot the problem? Notice that both routes have patterns that match URL's that look like `/blog/*`. The Symfony router will always choose the **first** matching route it finds. In other words, the `blog_show` route will *never* be matched. Instead, a URL like `/blog/my-blog-post` will match the first route (`blog`) and return a nonsense value of `my-blog-post` to the `{page}` parameter.

URL	route	parameters
/blog/2	blog	{page} = 2
/blog/my-blog-post	blog	{page} = my-blog-post

The answer to the problem is to add route *requirements*. The routes in this example would work perfectly if the `/blog/{page}` pattern *only* matched URLs where the `{page}` portion is an integer. Fortunately, regular expression requirements can easily be added for each parameter. For example:

- *YAML*

```
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }
    requirements:
        page:  \d+
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{page}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="page">1</default>
        <requirement key="page">\d+</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'page' => 1,
), array(
    'page' => '\d+',
)));

return $collection;
```

The `\d+` requirement is a regular expression that says that the value of the `{page}` parameter must be a digit (i.e. a number). The `blog` route will still match on a URL like `/blog/2` (because 2 is a number), but it will no longer match a URL like `/blog/my-blog-post` (because `my-blog-post` is *not* a number).

As a result, a URL like `/blog/my-blog-post` will now properly match the `blog_show` route.

URL	route	parameters
/blog/2	blog	{page} = 2
/blog/my-blog-post	blog_show	{slug} = my-blog-post

Earlier Routes always Win

What this all means is that the order of the routes is very important. If the `blog_show` route were placed above the `blog` route, the URL `/blog/2` would match `blog_show` instead of `blog` since the `{slug}` parameter of `blog_show` has no requirements. By using proper ordering and clever requirements, you can accomplish just about anything.

Since the parameter requirements are regular expressions, the complexity and flexibility of each requirement is entirely up to you. Suppose the homepage of your application is available in two different languages, based on the URL:

- *YAML*

```
homepage:
    pattern:   /{culture}
    defaults:  { _controller: AcmeDemoBundle:Main:homepage, culture: en }
    requirements:
        culture:  en|fr
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

        <route id="homepage" pattern="/{culture}">
            <default key="_controller">AcmeDemoBundle:Main:homepage</default>
            <default key="culture">en</default>
            <requirement key="culture">en|fr</requirement>
        </route>
    </routes>
```

- *PHP*

```
    use Symfony\Component\Routing\RouteCollection;
    use Symfony\Component\Routing\Route;

    $collection = new RouteCollection();
    $collection->add('homepage', new Route('/{culture}', array(
        '_controller' => 'AcmeDemoBundle:Main:homepage',
        'culture' => 'en',
    ), array(
        'culture' => 'en|fr',
    )));

    return $collection;
```

For incoming requests, the `{culture}` portion of the URL is matched against the regular expression `(en|fr)`.

/	{culture} = en
/en	{culture} = en
/fr	{culture} = fr
/es	*won't match this route*

Adding HTTP Method Requirements

In addition to the URL, you can also match on the *method* of the incoming request (i.e. GET, HEAD, POST, PUT, DELETE). Suppose you have a contact form with two controllers - one for displaying the form (on a GET request) and one for processing the form when it's submitted (on a POST request). This can be accomplished with the following route configuration:

- *YAML*

```
    contact:
        pattern:  /contact
        defaults: { _controller: AcmeDemoBundle:Main:contact }
        requirements:
            _method:  GET

    contact_process:
        pattern:  /contact
        defaults: { _controller: AcmeDemoBundle:Main:contactProcess }
        requirements:
            _method:  POST
```

- *XML*

```
    <?xml version="1.0" encoding="UTF-8" ?>

    <routes xmlns="http://symfony.com/schema/routing"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

        <route id="contact" pattern="/contact">
            <default key="_controller">AcmeDemoBundle:Main:contact</default>
            <requirement key="_method">GET</requirement>
        </route>

        <route id="contact_process" pattern="/contact">
            <default key="_controller">AcmeDemoBundle:Main:contactProcess</default>
            <requirement key="_method">POST</requirement>
        </route>
    </routes>
```

- *PHP*

```
    use Symfony\Component\Routing\RouteCollection;
    use Symfony\Component\Routing\Route;

    $collection = new RouteCollection();
    $collection->add('contact', new Route('/contact', array(
        '_controller' => 'AcmeDemoBundle:Main:contact',
    ), array(
        '_method' => 'GET',
    )));

    $collection->add('contact_process', new Route('/contact', array(
        '_controller' => 'AcmeDemoBundle:Main:contactProcess',
    ), array(
        '_method' => 'POST',
    )));

    return $collection;
```

Despite the fact that these two routes have identical patterns (`/contact`), the first route will match only GET requests and the second route will match only POST requests. This means that you can display the form and submit the form via the same URL, while using distinct controllers for the two actions.

Note: If no `_method` requirement is specified, the route will match on *all* methods.

Like the other requirements, the `_method` requirement is parsed as a regular expression. To match `GET` *or* `POST` requests, you can use `GET|POST`.

Advanced Routing Example

At this point, you have everything you need to create a powerful routing structure in Symfony. The following is an example of just how flexible the routing system can be:

- *YAML*

```
    article_show:
      pattern:  /articles/{culture}/{year}/{title}.{_format}
      defaults: { _controller: AcmeDemoBundle:Article:show, _format: html }
      requirements:
          culture:  en|fr
          _format:  html|rss
          year:     \d+
```

- *XML*

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="article_show" pattern="/articles/{culture}/{year}/{title}.{_format}">
        <default key="_controller">AcmeDemoBundle:Article:show</default>
        <default key="_format">html</default>
        <requirement key="culture">en|fr</requirement>
        <requirement key="_format">html|rss</requirement>
        <requirement key="year">\d+</requirement>
    </route>
</routes>
```

- *PHP*

```php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('homepage', new Route('/articles/{culture}/{year}/{title}.{_format}', array(
    '_controller' => 'AcmeDemoBundle:Article:show',
    '_format' => 'html',
), array(
    'culture' => 'en|fr',
    '_format' => 'html|rss',
    'year' => '\d+',
)));

return $collection;
```

As you've seen, this route will only match if the `{culture}` portion of the URL is either `en` or `fr` and if the `{year}` is a number. This route also shows how you can use a period between placeholders instead of a slash. URLs matching this route might look like:

- `/articles/en/2010/my-post`
- `/articles/fr/2010/my-post.rss`

The Special `_format` Routing Parameter

This example also highlights the special `_format` routing parameter. When using this parameter, the matched value becomes the "request format" of the `Request` object. Ultimately, the request format is used for such things such as setting the `Content-Type` of the response (e.g. a `json` request format translates into a `Content-Type` of `application/json`). It can also be used in the controller to render a different template for each value of `_format`. The `_format` parameter is a very powerful way to render the same content in different formats.

Special Routing Parameters

As you've seen, each routing parameter or default value is eventually available as an argument in the controller method. Additionally, there are three parameters that are special: each adds a unique piece of functionality inside your application:

- `_controller`: As you've seen, this parameter is used to determine which controller is executed when the route is matched;
- `_format`: Used to set the request format (*read more*);
- `_locale`: Used to set the locale on the request (*read more*);

Tip: If you use the `_locale` parameter in a route, that value will also be stored on the session so that subsequent requests keep this same locale.

Controller Naming Pattern

Every route must have a `_controller` parameter, which dictates which controller should be executed when that route is matched. This parameter uses a simple string pattern called the *logical controller name*, which Symfony maps to a specific PHP method and class. The pattern has three parts, each separated by a colon:

bundle:**controller**:**action**

For example, a `_controller` value of `AcmeBlogBundle:Blog:show` means:

Bundle	Controller Class	Method Name
AcmeBlogBundle	BlogController	showAction

The controller might look like this:

```
// src/Acme/BlogBundle/Controller/BlogController.php

namespace Acme\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($slug)
    {
        // ...
    }
}
```

Notice that Symfony adds the string `Controller` to the class name (`Blog` => `BlogController`) and `Action` to the method name (`show` => `showAction`).

You could also refer to this controller using its fully-qualified class name and method: `Acme\BlogBundle\Controller\BlogController::showAction`. But if you follow some simple conventions, the logical name is more concise and allows more flexibility.

Note: In addition to using the logical name or the fully-qualified class name, Symfony supports a third way of referring to a controller. This method uses just one colon separator (e.g. `service_name:indexAction`) and refers to the controller as a service (see How to define Controllers as Services).

Route Parameters and Controller Arguments

The route parameters (e.g. `{slug}`) are especially important because each is made available as an argument to the controller method:

```
public function showAction($slug)
{
```

```
  // ...
}
```

In reality, the entire `defaults` collection is merged with the parameter values to form a single array. Each key of that array is available as an argument on the controller.

In other words, for each argument of your controller method, Symfony looks for a route parameter of that name and assigns its value to that argument. In the advanced example above, any combination (in any order) of the following variables could be used as arguments to the `showAction()` method:

- `$culture`
- `$year`
- `$title`
- `$_format`
- `$_controller`

Since the placeholders and `defaults` collection are merged together, even the `$_controller` variable is available. For a more detailed discussion, see *Route Parameters as Controller Arguments*.

Tip: You can also use a special `$_route` variable, which is set to the name of the route that was matched.

Including External Routing Resources

All routes are loaded via a single configuration file - usually `app/config/routing.yml` (see *Creating Routes* above). Commonly, however, you'll want to load routes from other places, like a routing file that lives inside a bundle. This can be done by "importing" that file:

- *YAML*

```
# app/config/routing.yml
acme_hello:
    resource: "@AcmeHelloBundle/Resources/config/routing.yml"
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@AcmeHelloBundle/Resources/config/routing.xml" />
</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;

$collection = new RouteCollection();
$collection->addCollection($loader->import("@AcmeHelloBundle/Resources/config/routing.php"));

return $collection;
```

Note: When importing resources from YAML, the key (e.g. `acme_hello`) is meaningless. Just be sure that it's unique so no other lines override it.

The `resource` key loads the given routing resource. In this example the resource is the full path to a file, where the `@AcmeHelloBundle` shortcut syntax resolves to the path of that bundle. The imported file might look like this:

- *YAML*

```
 # src/Acme/HelloBundle/Resources/config/routing.yml
acme_hello:
     pattern:  /hello/{name}
     defaults: { _controller: AcmeHelloBundle:Hello:index }
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="acme_hello" pattern="/hello/{name}">
        <default key="_controller">AcmeHelloBundle:Hello:index</default>
    </route>
</routes>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('acme_hello', new Route('/hello/{name}', array(
    '_controller' => 'AcmeHelloBundle:Hello:index',
)));

return $collection;
```

The routes from this file are parsed and loaded in the same way as the main routing file.

Prefixing Imported Routes

You can also choose to provide a "prefix" for the imported routes. For example, suppose you want the `acme_hello` route to have a final pattern of `/admin/hello/{name}` instead of simply `/hello/{name}`:

- *YAML*

```
# app/config/routing.yml
acme_hello:
    resource: "@AcmeHelloBundle/Resources/config/routing.yml"
    prefix:   /admin
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
```

```
    <routes xmlns="http://symfony.com/schema/routing"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

        <import resource="@AcmeHelloBundle/Resources/config/routing.xml" prefix="/admin" />
    </routes>
```

- *PHP*

```
    // app/config/routing.php
    use Symfony\Component\Routing\RouteCollection;

    $collection = new RouteCollection();
    $collection->addCollection($loader->import("@AcmeHelloBundle/Resources/config/routing.php"), '/a

    return $collection;
```

The string `/admin` will now be prepended to the pattern of each route loaded from the new routing resource.

Visualizing & Debugging Routes

While adding and customizing routes, it's helpful to be able to visualize and get detailed information about your routes. A great way to see every route in your application is via the `router:debug` console command. Execute the command by running the following from the root of your project.

```
php app/console router:debug
```

The command will print a helpful list of *all* the configured routes in your application:

```
homepage              ANY       /
contact               GET       /contact
contact_process       POST      /contact
article_show          ANY       /articles/{culture}/{year}/{title}.{_format}
blog                  ANY       /blog/{page}
blog_show             ANY       /blog/{slug}
```

You can also get very specific information on a single route by including the route name after the command:

```
php app/console router:debug article_show
```

Generating URLs

The routing system should also be used to generate URLs. In reality, routing is a bi-directional system: mapping the URL to a controller+parameters and a route+parameters back to a URL. The **:method:'Symfony\\Component\\Routing\\Router::match'** and **:method:'Symfony\\Component\\Routing\\Router::generate'** methods form this bi-directional system. Take the `blog_show` example route from earlier:

```
$params = $router->match('/blog/my-blog-post');
// array('slug' => 'my-blog-post', '_controller' => 'AcmeBlogBundle:Blog:show')

$uri = $router->generate('blog_show', array('slug' => 'my-blog-post'));
// /blog/my-blog-post
```

To generate a URL, you need to specify the name of the route (e.g. `blog_show`) and any wildcards (e.g. `slug = my-blog-post`) used in the pattern for that route. With this information, any URL can easily be generated:

```
class MainController extends Controller
{
    public function showAction($slug)
    {
      // ...

      $url = $this->get('router')->generate('blog_show', array('slug' => 'my-blog-post'));
    }
}
```

In an upcoming section, you'll learn how to generate URLs from inside templates.

Tip: If the frontend of your application uses AJAX requests, you might want to be able to generate URLs in JavaScript based on your routing configuration. By using the FOSJsRoutingBundle, you can do exactly that:

```
var url = Routing.generate('blog_show', { "slug": 'my-blog-post'});
```

For more information, see the documentation for that bundle.

Generating Absolute URLs

By default, the router will generate relative URLs (e.g. `/blog`). To generate an absolute URL, simply pass `true` to the third argument of the `generate()` method:

```
$router->generate('blog_show', array('slug' => 'my-blog-post'), true);
// http://www.example.com/blog/my-blog-post
```

Note: The host that's used when generating an absolute URL is the host of the current `Request` object. This is detected automatically based on server information supplied by PHP. When generating absolute URLs for scripts run from the command line, you'll need to manually set the desired host on the `Request` object:

```
$request->headers->set('HOST', 'www.example.com');
```

Generating URLs with Query Strings

The `generate` method takes an array of wildcard values to generate the URI. But if you pass extra ones, they will be added to the URI as a query string:

```
$router->generate('blog', array('page' => 2, 'category' => 'Symfony'));
// /blog/2?category=Symfony
```

Generating URLs from a template

The most common place to generate a URL is from within a template when linking between pages in your application. This is done just as before, but using a template helper function:

- *Twig*

```
<a href="{{ path('blog_show', { 'slug': 'my-blog-post' }) }}">
  Read this blog post.
</a>
```

- *PHP*

```
    <a href="<?php echo $view['router']->generate('blog_show', array('slug' => 'my-blog-post')) ?>">
        Read this blog post.
    </a>
```

Absolute URLs can also be generated.

- *Twig*

```
    <a href="{{ url('blog_show', { 'slug': 'my-blog-post' }) }}">
      Read this blog post.
    </a>
```

- *PHP*

```
    <a href="<?php echo $view['router']->generate('blog_show', array('slug' => 'my-blog-post'), true
        Read this blog post.
    </a>
```

Summary

Routing is a system for mapping the URL of incoming requests to the controller function that should be called to process the request. It both allows you to specify beautiful URLs and keeps the functionality of your application decoupled from those URLs. Routing is a two-way mechanism, meaning that it should also be used to generate URLs.

Learn more from the Cookbook

- How to force routes to always use HTTPS or HTTP

2.1.7 Creating and using Templates

As you know, the controller is responsible for handling each request that comes into a Symfony2 application. In reality, the controller delegates the most of the heavy work to other places so that code can be tested and reused. When a controller needs to generate HTML, CSS or any other content, it hands the work off to the templating engine. In this chapter, you'll learn how to write powerful templates that can be used to return content to the user, populate email bodies, and more. You'll learn shortcuts, clever ways to extend templates and how to reuse template code.

Templates

A template is simply a text file that can generate any text-based format (HTML, XML, CSV, LaTeX ...). The most familiar type of template is a *PHP* template - a text file parsed by PHP that contains a mix of text and PHP code:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1><?php echo $page_title ?></h1>

        <ul id="navigation">
            <?php foreach ($navigation as $item): ?>
                <li>
                    <a href="<?php echo $item->getHref() ?>">
                        <?php echo $item->getCaption() ?>
```

```
                    </a>
                </li>
            <?php endforeach; ?>
        </ul>
    </body>
</html>
```

But Symfony2 packages an even more powerful templating language called Twig. Twig allows you to write concise, readable templates that are more friendly to web designers and, in several ways, more powerful than PHP templates:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Twig defines two types of special syntax:

- `{{ ... }}`: "Says something": prints a variable or the result of an expression to the template;
- `{% ... %}`: "Does something": a **tag** that controls the logic of the template; it is used to execute statements such as for-loops for example.

Note: There is a third syntax used for creating comments: `{# this is a comment #}`. This syntax can be used across multiple lines like the PHP-equivalent `/* comment */` syntax.

Twig also contains **filters**, which modify content before being rendered. The following makes the `title` variable all uppercase before rendering it:

```
{{ title|upper }}
```

Twig comes with a long list of tags and filters that are available by default. You can even add your own extensions to Twig as needed.

Tip: Registering a Twig extension is as easy as creating a new service and tagging it with `twig.extension` *tag*.

As you'll see throughout the documentation, Twig also supports functions and new functions can be easily added. For example, the following uses a standard `for` tag and the `cycle` function to print ten div tags, with alternating `odd`, `even` classes:

```
{% for i in 0..10 %}
    <div class="{{ cycle(['odd', 'even'], i) }}">
      <!-- some HTML here -->
    </div>
{% endfor %}
```

Throughout this chapter, template examples will be shown in both Twig and PHP.

Why Twig?

Twig templates are meant to be simple and won't process PHP tags. This is by design: the Twig template system is meant to express presentation, not program logic. The more you use Twig, the more you'll appreciate and benefit from this distinction. And of course, you'll be loved by web designers everywhere.
Twig can also do things that PHP can't, such as true template inheritance (Twig templates compile down to PHP classes that inherit from each other), whitespace control, sandboxing, and the inclusion of custom functions and filters that only affect templates. Twig contains little features that make writing templates easier and more concise. Take the following example, which combines a loop with a logical `if` statement:

```
<ul>
    {% for user in users %}
        <li>{{ user.username }}</li>
    {% else %}
        <li>No users found</li>
    {% endfor %}
</ul>
```

Twig Template Caching

Twig is fast. Each Twig template is compiled down to a native PHP class that is rendered at runtime. The compiled classes are located in the `app/cache/{environment}/twig` directory (where `{environment}` is the environment, such as `dev` or `prod`) and in some cases can be useful while debugging. See *Environments* for more information on environments.

When `debug` mode is enabled (common in the `dev` environment), a Twig template will be automatically recompiled when changes are made to it. This means that during development you can happily make changes to a Twig template and instantly see the changes without needing to worry about clearing any cache.

When `debug` mode is disabled (common in the `prod` environment), however, you must clear the Twig cache directory so that the Twig templates will regenerate. Remember to do this when deploying your application.

Template Inheritance and Layouts

More often than not, templates in a project share common elements, like the header, footer, sidebar or more. In Symfony2, we like to think about this problem differently: a template can be decorated by another one. This works exactly the same as PHP classes: template inheritance allows you to build a base "layout" template that contains all the common elements of your site defined as **blocks** (think "PHP class with base methods"). A child template can extend the base layout and override any of its blocks (think "PHP subclass that overrides certain methods of its parent class").

First, build a base layout file:

- *Twig*

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block title %}Test Application{% endblock %}</title>
    </head>
    <body>
        <div id="sidebar">
            {% block sidebar %}
```

```
                <ul>
                    <li><a href="/">Home</a></li>
                    <li><a href="/blog">Blog</a></li>
                </ul>
                {% endblock %}
            </div>

            <div id="content">
                {% block body %}{% endblock %}
            </div>
        </body>
    </html>
```

- *PHP*

```
    <!-- app/Resources/views/base.html.php -->
    <!DOCTYPE html>
    <html>
        <head>
            <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
            <title><?php $view['slots']->output('title', 'Test Application') ?></title>
        </head>
        <body>
            <div id="sidebar">
                <?php if ($view['slots']->has('sidebar')): ?>
                    <?php $view['slots']->output('sidebar') ?>
                <?php else: ?>
                    <ul>
                        <li><a href="/">Home</a></li>
                        <li><a href="/blog">Blog</a></li>
                    </ul>
                <?php endif; ?>
            </div>

            <div id="content">
                <?php $view['slots']->output('body') ?>
            </div>
        </body>
    </html>
```

Note: Though the discussion about template inheritance will be in terms of Twig, the philosophy is the same between Twig and PHP templates.

This template defines the base HTML skeleton document of a simple two-column page. In this example, three `{% block %}` areas are defined (`title`, `sidebar` and `body`). Each block may be overridden by a child template or left with its default implementation. This template could also be rendered directly. In that case the `title`, `sidebar` and `body` blocks would simply retain the default values used in this template.

A child template might look like this:

- *Twig*

```
    {# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
    {% extends '::base.html.twig' %}

    {% block title %}My cool blog posts{% endblock %}

    {% block body %}
```

```
        {% for entry in blog_entries %}
            <h2>{{ entry.title }}</h2>
            <p>{{ entry.body }}</p>
        {% endfor %}
    {% endblock %}
```

- *PHP*

```
    <!-- src/Acme/BlogBundle/Resources/views/Blog/index.html.php -->
    <?php $view->extend('::base.html.php') ?>

    <?php $view['slots']->set('title', 'My cool blog posts') ?>

    <?php $view['slots']->start('body') ?>
        <?php foreach ($blog_entries as $entry): ?>
            <h2><?php echo $entry->getTitle() ?></h2>
            <p><?php echo $entry->getBody() ?></p>
        <?php endforeach; ?>
    <?php $view['slots']->stop() ?>
```

Note: The parent template is identified by a special string syntax (`::base.html.twig`) that indicates that the template lives in the `app/Resources/views` directory of the project. This naming convention is explained fully in *Template Naming and Locations*.

The key to template inheritance is the `{% extends %}` tag. This tells the templating engine to first evaluate the base template, which sets up the layout and defines several blocks. The child template is then rendered, at which point the `title` and `body` blocks of the parent are replaced by those from the child. Depending on the value of `blog_entries`, the output might look like this:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>My cool blog posts</title>
    </head>
    <body>
        <div id="sidebar">
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
        </div>

        <div id="content">
            <h2>My first post</h2>
            <p>The body of the first post.</p>

            <h2>Another post</h2>
            <p>The body of the second post.</p>
        </div>
    </body>
</html>
```

Notice that since the child template didn't define a `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used by default.

You can use as many levels of inheritance as you want. In the next section, a common three-level inheritance model will be explained along with how templates are organized inside a Symfony2 project.

When working with template inheritance, here are some tips to keep in mind:

- If you use `{% extends %}` in a template, it must be the first tag in that template.
- The more `{% block %}` tags you have in your base templates, the better. Remember, child templates don't have to define all parent blocks, so create as many blocks in your base templates as you want and give each a sensible default. The more blocks your base templates have, the more flexible your layout will be.
- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template. In some cases, a better solution may be to move the content to a new template and `include` it (see *Including other Templates*).
- If you need to get the content of a block from the parent template, you can use the `{{ parent() }}` function. This is useful if you want to add to the contents of a parent block instead of completely overriding it:

```
{% block sidebar %}
    <h3>Table of Contents</h3>
    ...
    {{ parent() }}
{% endblock %}
```

Template Naming and Locations

By default, templates can live in two different locations:

- `app/Resources/views/`: The applications `views` directory can contain application-wide base templates (i.e. your application's layouts) as well as templates that override bundle templates (see *Overriding Bundle Templates*);
- `path/to/bundle/Resources/views/`: Each bundle houses its templates in its `Resources/views` directory (and subdirectories). The majority of templates will live inside a bundle.

Symfony2 uses a **bundle**:**controller**:**template** string syntax for templates. This allows for several different types of templates, each which lives in a specific location:

- `AcmeBlogBundle:Blog:index.html.twig`: This syntax is used to specify a template for a specific page. The three parts of the string, each separated by a colon (`:`), mean the following:
  - `AcmeBlogBundle`: (*bundle*) the template lives inside the `AcmeBlogBundle` (e.g. `src/Acme/BlogBundle`);
  - `Blog`: (*controller*) indicates that the template lives inside the `Blog` subdirectory of `Resources/views`;
  - `index.html.twig`: (*template*) the actual name of the file is `index.html.twig`.

  Assuming that the `AcmeBlogBundle` lives at `src/Acme/BlogBundle`, the final path to the layout would be `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`.
- `AcmeBlogBundle::layout.html.twig`: This syntax refers to a base template that's specific to the `AcmeBlogBundle`. Since the middle, "controller", portion is missing (e.g. `Blog`), the template lives at `Resources/views/layout.html.twig` inside `AcmeBlogBundle`.
- `::base.html.twig`: This syntax refers to an application-wide base template or layout. Notice that the string begins with two colons (`::`), meaning that both the *bundle* and *controller* portions are missing. This means that the template is not located in any bundle, but instead in the root `app/Resources/views/` directory.

In the *Overriding Bundle Templates* section, you'll find out how each template living inside the `AcmeBlogBundle`, for example, can be overridden by placing a template of the same name in the `app/Resources/AcmeBlogBundle/views/` directory. This gives the power to override templates from any vendor bundle.

Tip: Hopefully the template naming syntax looks familiar - it's the same naming convention used to refer to *Controller Naming Pattern*.

Template Suffix

The **bundle**:**controller**:**template** format of each template specifies *where* the template file is located. Every template name also has two extensions that specify the *format* and *engine* for that template.

- **AcmeBlogBundle:Blog:index.html.twig** - HTML format, Twig engine
- **AcmeBlogBundle:Blog:index.html.php** - HTML format, PHP engine
- **AcmeBlogBundle:Blog:index.css.twig** - CSS format, Twig engine

By default, any Symfony2 template can be written in either Twig or PHP, and the last part of the extension (e.g. `.twig` or `.php`) specifies which of these two *engines* should be used. The first part of the extension, (e.g. `.html`, `.css`, etc) is the final format that the template will generate. Unlike the engine, which determines how Symfony2 parses the template, this is simply an organizational tactic used in case the same resource needs to be rendered as HTML (`index.html.twig`), XML (`index.xml.twig`), or any other format. For more information, read the *Debugging* section.

Note: The available "engines" can be configured and even new engines added. See *Templating Configuration* for more details.

Tags and Helpers

You already understand the basics of templates, how they're named and how to use template inheritance. The hardest parts are already behind you. In this section, you'll learn about a large group of tools available to help perform the most common template tasks such as including other templates, linking to pages and including images.

Symfony2 comes bundled with several specialized Twig tags and functions that ease the work of the template designer. In PHP, the templating system provides an extensible *helper* system that provides useful features in a template context.

We've already seen a few built-in Twig tags (`{% block %}` & `{% extends %}`) as well as an example of a PHP helper (`$view['slots']`). Let's learn a few more.

Including other Templates

You'll often want to include the same template or code fragment on several different pages. For example, in an application with "news articles", the template code displaying an article might be used on the article detail page, on a page displaying the most popular articles, or in a list of the latest articles.

When you need to reuse a chunk of PHP code, you typically move the code to a new PHP class or function. The same is true for templates. By moving the reused template code into its own template, it can be included from any other template. First, create the template that you'll need to reuse.

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/views/Article/articleDetails.html.twig #}
<h2>{{ article.title }}</h2>
<h3 class="byline">by {{ article.authorName }}</h3>

<p>
```

```
        {{ article.body }}
    </p>
```

- *PHP*

```
    <!-- src/Acme/ArticleBundle/Resources/views/Article/articleDetails.html.php -->
    <h2><?php echo $article->getTitle() ?></h2>
    <h3 class="byline">by <?php echo $article->getAuthorName() ?></h3>

    <p>
        <?php echo $article->getBody() ?>
    </p>
```

Including this template from any other template is simple:

- *Twig*

```
    {# src/Acme/ArticleBundle/Resources/Article/list.html.twig #}
    {% extends 'AcmeArticleBundle::layout.html.twig' %}

    {% block body %}
        <h1>Recent Articles<h1>

        {% for article in articles %}
            {% include 'AcmeArticleBundle:Article:articleDetails.html.twig' with {'article': article
        {% endfor %}
    {% endblock %}
```

- *PHP*

```
    <!-- src/Acme/ArticleBundle/Resources/Article/list.html.php -->
    <?php $view->extend('AcmeArticleBundle::layout.html.php') ?>

    <?php $view['slots']->start('body') ?>
        <h1>Recent Articles</h1>

        <?php foreach ($articles as $article): ?>
            <?php echo $view->render('AcmeArticleBundle:Article:articleDetails.html.php', array('art
        <?php endforeach; ?>
    <?php $view['slots']->stop() ?>
```

The template is included using the `{% include %}` tag. Notice that the template name follows the same typical convention. The `articleDetails.html.twig` template uses an `article` variable. This is passed in by the `list.html.twig` template using the `with` command.

Tip: The `{'article': article}` syntax is the standard Twig syntax for hash maps (i.e. an array with named keys). If we needed to pass in multiple elements, it would look like this: `{'foo': foo, 'bar': bar}`.

Embedding Controllers

In some cases, you need to do more than include a simple template. Suppose you have a sidebar in your layout that contains the three most recent articles. Retrieving the three articles may include querying the database or performing other heavy logic that can't be done from within a template.

The solution is to simply embed the result of an entire controller from your template. First, create a controller that renders a certain number of recent articles:

```
// src/Acme/ArticleBundle/Controller/ArticleController.php

class ArticleController extends Controller
{
    public function recentArticlesAction($max = 3)
    {
        // make a database call or other logic to get the "$max" most recent articles
        $articles = ...;

        return $this->render('AcmeArticleBundle:Article:recentList.html.twig', array('articles' => $a
    }
}
```

The `recentList` template is perfectly straightforward:

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{% for article in articles %}
    <a href="/article/{{ article.slug }}">
        {{ article.title }}
    </a>
{% endfor %}
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/views/Article/recentList.html.php -->
<?php foreach ($articles as $article): ?>
    <a href="/article/<?php echo $article->getSlug() ?>">
        <?php echo $article->getTitle() ?>
    </a>
<?php endforeach; ?>
```

Note: Notice that we've cheated and hardcoded the article URL in this example (e.g. `/article/*slug*`). This is a bad practice. In the next section, you'll learn how to do this correctly.

To include the controller, you'll need to refer to it using the standard string syntax for controllers (i.e. **bundle**:**controller**:**action**):

- *Twig*

```
{# app/Resources/views/base.html.twig #}
...

<div id="sidebar">
    {% render "AcmeArticleBundle:Article:recentArticles" with {'max': 3} %}
</div>
```

- *PHP*

```
<!-- app/Resources/views/base.html.php -->
...

<div id="sidebar">
    <?php echo $view['actions']->render('AcmeArticleBundle:Article:recentArticles', array('max'
</div>
```

Whenever you find that you need a variable or a piece of information that you don't have access to in a template, consider rendering a controller. Controllers are fast to execute and promote good code organization and reuse.

Linking to Pages

Creating links to other pages in your application is one of the most common jobs for a template. Instead of hardcoding URLs in templates, use the `path` Twig function (or the `router` helper in PHP) to generate URLs based on the routing configuration. Later, if you want to modify the URL of a particular page, all you'll need to do is change the routing configuration; the templates will automatically generate the new URL.

First, link to the "_welcome" page, which is accessible via the following routing configuration:

- *YAML*

```
_welcome:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Welcome:index }
```

- *XML*

```
<route id="_welcome" pattern="/">
    <default key="_controller">AcmeDemoBundle:Welcome:index</default>
</route>
```

- *PHP*

```
$collection = new RouteCollection();
$collection->add('_welcome', new Route('/', array(
    '_controller' => 'AcmeDemoBundle:Welcome:index',
)));

return $collection;
```

To link to the page, just use the `path` Twig function and refer to the route:

- *Twig*

```
<a href="{{ path('_welcome') }}">Home</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('_welcome') ?>">Home</a>
```

As expected, this will generate the URL `/`. Let's see how this works with a more complicated route:

- *YAML*

```
article_show:
    pattern:  /article/{slug}
    defaults: { _controller: AcmeArticleBundle:Article:show }
```

- *XML*

```
<route id="article_show" pattern="/article/{slug}">
    <default key="_controller">AcmeArticleBundle:Article:show</default>
</route>
```

- *PHP*

```
$collection = new RouteCollection();
$collection->add('article_show', new Route('/article/{slug}', array(
    '_controller' => 'AcmeArticleBundle:Article:show',
)));

return $collection;
```

In this case, you need to specify both the route name (`article_show`) and a value for the `{slug}` parameter. Using this route, let's revisit the `recentList` template from the previous section and link to the articles correctly:

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{% for article in articles %}
    <a href="{{ path('article_show', { 'slug': article.slug }) }}">
        {{ article.title }}
    </a>
{% endfor %}
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/views/Article/recentList.html.php -->
<?php foreach ($articles in $article): ?>
    <a href="<?php echo $view['router']->generate('article_show', array('slug' => $article->getS
        <?php echo $article->getTitle() ?>
    </a>
<?php endforeach; ?>
```

Tip: You can also generate an absolute URL by using the `url` Twig function:

```
<a href="{{ url('_welcome') }}">Home</a>
```

The same can be done in PHP templates by passing a third argument to the `generate()` method:

```
<a href="<?php echo $view['router']->generate('_welcome', array(), true) ?>">Home</a>
```

Linking to Assets

Templates also commonly refer to images, Javascript, stylesheets and other assets. Of course you could hard-code the path to these assets (e.g. `/images/logo.png`), but Symfony2 provides a more dynamic option via the `assets` Twig function:

- *Twig*

```
<img src="{{ asset('images/logo.png') }}" alt="Symfony!" />

<link href="{{ asset('css/blog.css') }}" rel="stylesheet" type="text/css" />
```

- *PHP*

```
<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" alt="Symfony!" />

<link href="<?php echo $view['assets']->getUrl('css/blog.css') ?>" rel="stylesheet" type="text/c
```

The `asset` function's main purpose is to make your application more portable. If your application lives at the root of your host (e.g. http://example.com), then the rendered paths should be `/images/logo.png`. But if your application lives in a subdirectory (e.g. http://example.com/my_app), each asset path should render with the subdirectory (e.g. `/my_app/images/logo.png`). The `asset` function takes care of this by determining how your application is being used and generating the correct paths accordingly.

Additionally, if you use the `asset` function, Symfony can automatically append a query string to your asset, in order to guarantee that updated static assets won't be cached when deployed. For example, `/images/logo.png` might look like `/images/logo.png?v2`. For more information, see the *assets_version* configuration option.

Including Stylesheets and Javascripts in Twig

No site would be complete without including Javascript files and stylesheets. In Symfony, the inclusion of these assets is handled elegantly by taking advantage of Symfony's template inheritance.

Tip: This section will teach you the philosophy behind including stylesheet and Javascript assets in Symfony. Symfony also packages another library, called Assetic, which follows this philosophy but allows you to do much more interesting things with those assets. For more information on using Assetic see How to Use Assetic for Asset Management.

Start by adding two blocks to your base template that will hold your assets: one called `stylesheets` inside the `head` tag and another called `javascripts` just above the closing `body` tag. These blocks will contain all of the stylesheets and Javascripts that you'll need throughout your site:

```
{# 'app/Resources/views/base.html.twig' #}
<html>
    <head>
        {# ... #}

        {% block stylesheets %}
            <link href="{{ asset('/css/main.css') }}" type="text/css" rel="stylesheet" />
        {% endblock %}
    </head>
    <body>
        {# ... #}

        {% block javascripts %}
            <script src="{{ asset('/js/main.js') }}" type="text/javascript"></script>
        {% endblock %}
    </body>
</html>
```

That's easy enough! But what if you need to include an extra stylesheet or Javascript from a child template? For example, suppose you have a contact page and you need to include a `contact.css` stylesheet *just* on that page. From inside that contact page's template, do the following:

```
{# src/Acme/DemoBundle/Resources/views/Contact/contact.html.twig #}
{% extends '::base.html.twig' %}

{% block stylesheets %}
    {{ parent() }}

    <link href="{{ asset('/css/contact.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}

{# ... #}
```

In the child template, you simply override the `stylesheets` block and put your new stylesheet tag inside of that block. Of course, since you want to add to the parent block's content (and not actually *replace* it), you should use the `parent()` Twig function to include everything from the `stylesheets` block of the base template.

You can also include assets located in your bundles' `Resources/public` folder. You will need to run the `php app/console assets:install target [--symlink]` command, which moves (or symlinks) files into the correct location. (target is by default "web").

```
<link href="{{ asset('bundles/acmedemo/css/contact.css') }}" type="text/css" rel="stylesheet" />
```

The end result is a page that includes both the `main.css` and `contact.css` stylesheets.

Global Template Variables

During each request, Symfony2 will set a global template variable `app` in both Twig and PHP template engines by default. The `app` variable is a `Symfony\Bundle\FrameworkBundle\Templating\GlobalVariables` instance which will give you access to some application specific variables automatically:

- `app.security` - The security context.
- `app.user` - The current user object.
- `app.request` - The request object.
- `app.session` - The session object.
- `app.environment` - The current environment (dev, prod, etc).
- `app.debug` - True if in debug mode. False otherwise.
- *Twig*

```
<p>Username: {{ app.user.username }}</p>
{% if app.debug %}
    <p>Request method: {{ app.request.method }}</p>
    <p>Application Environment: {{ app.environment }}</p>
{% endif %}
```

- *PHP*

```
<p>Username: <?php echo $app->getUser()->getUsername() ?></p>
<?php if ($app->getDebug()): ?>
    <p>Request method: <?php echo $app->getRequest()->getMethod() ?></p>
    <p>Application Environment: <?php echo $app->getEnvironment() ?></p>
<?php endif; ?>
```

Tip: You can add your own global template variables. See the cookbook example on Global Variables.

Configuring and using the `templating` Service

The heart of the template system in Symfony2 is the templating `Engine`. This special object is responsible for rendering templates and returning their content. When you render a template in a controller, for example, you're actually using the templating engine service. For example:

```
return $this->render('AcmeArticleBundle:Article:index.html.twig');
```

is equivalent to

```
$engine = $this->container->get('templating');
$content = $engine->render('AcmeArticleBundle:Article:index.html.twig');

return $response = new Response($content);
```

The templating engine (or "service") is preconfigured to work automatically inside Symfony2. It can, of course, be configured further in the application configuration file:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    templating: { engines: ['twig'] }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:templating>
    <framework:engine id="twig" />
</framework:templating>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'templating'      => array(
        'engines' => array('twig'),
    ),
));
```

Several configuration options are available and are covered in the Configuration Appendix.

Note: The `twig` engine is mandatory to use the webprofiler (as well as many third-party bundles).

Overriding Bundle Templates

The Symfony2 community prides itself on creating and maintaining high quality bundles (see KnpBundles.com) for a large number of different features. Once you use a third-party bundle, you'll likely need to override and customize one or more of its templates.

Suppose you've included the imaginary open-source `AcmeBlogBundle` in your project (e.g. in the `src/Acme/BlogBundle` directory). And while you're really happy with everything, you want to override the blog "list" page to customize the markup specifically for your application. By digging into the `Blog` controller of the `AcmeBlogBundle`, you find the following:

```
public function indexAction()
{
    $blogs = // some logic to retrieve the blogs

    $this->render('AcmeBlogBundle:Blog:index.html.twig', array('blogs' => $blogs));
}
```

When the `AcmeBlogBundle:Blog:index.html.twig` is rendered, Symfony2 actually looks in two different locations for the template:

1. `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig`
2. `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`

To override the bundle template, just copy the `index.html.twig` template from the bundle to `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig` (the `app/Resources/AcmeBlogBundle` directory won't exist, so you'll need to create it). You're now free to customize the template.

This logic also applies to base bundle templates. Suppose also that each template in `AcmeBlogBundle` inherits from a base template called `AcmeBlogBundle::layout.html.twig`. Just as before, Symfony2 will look in the following two places for the template:

1. `app/Resources/AcmeBlogBundle/views/layout.html.twig`
2. `src/Acme/BlogBundle/Resources/views/layout.html.twig`

Once again, to override the template, just copy it from the bundle to `app/Resources/AcmeBlogBundle/views/layout.html.twig`. You're now free to customize this copy as you see fit.

If you take a step back, you'll see that Symfony2 always starts by looking in the `app/Resources/{BUNDLE_NAME}/views/` directory for a template. If the template doesn't exist there, it continues by checking inside the `Resources/views` directory of the bundle itself. This means that all bundle templates can be overridden by placing them in the correct `app/Resources` subdirectory.

Overriding Core Templates

Since the Symfony2 framework itself is just a bundle, core templates can be overridden in the same way. For example, the core `TwigBundle` contains a number of different "exception" and "error" templates that can be overridden by copying each from the `Resources/views/Exception` directory of the `TwigBundle` to, you guessed it, the `app/Resources/TwigBundle/views/Exception` directory.

Three-level Inheritance

One common way to use inheritance is to use a three-level approach. This method works perfectly with the three different types of templates we've just covered:

- Create a `app/Resources/views/base.html.twig` file that contains the main layout for your application (like in the previous example). Internally, this template is called `::base.html.twig`;
- Create a template for each "section" of your site. For example, an `AcmeBlogBundle`, would have a template called `AcmeBlogBundle::layout.html.twig` that contains only blog section-specific elements;

```
{# src/Acme/BlogBundle/Resources/views/layout.html.twig #}
{% extends '::base.html.twig' %}

{% block body %}
    <h1>Blog Application</h1>

    {% block content %}{% endblock %}
{% endblock %}
```

- Create individual templates for each page and make each extend the appropriate section template. For example, the "index" page would be called something close to `AcmeBlogBundle:Blog:index.html.twig` and list the actual blog posts.

```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends 'AcmeBlogBundle::layout.html.twig' %}

{% block content %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

Notice that this template extends the section template -(`AcmeBlogBundle::layout.html.twig`) which in-turn extends the base application layout (`::base.html.twig`). This is the common three-level inheritance model.

When building your application, you may choose to follow this method or simply make each page template extend the base application template directly (e.g. `{% extends '::base.html.twig' %}`). The three-template model is a best-practice method used by vendor bundles so that the base template for a bundle can be easily overridden to properly extend your application's base layout.

Output Escaping

When generating HTML from a template, there is always a risk that a template variable may output unintended HTML or dangerous client-side code. The result is that dynamic content could break the HTML of the resulting page or allow a malicious user to perform a Cross Site Scripting (XSS) attack. Consider this classic example:

- *Twig*

```
Hello {{ name }}
```

- *PHP*

```
Hello <?php echo $name ?>
```

Imagine that the user enters the following code as his/her name:

```
<script>alert('hello!')</script>
```

Without any output escaping, the resulting template will cause a JavaScript alert box to pop up:

```
Hello <script>alert('hello!')</script>
```

And while this seems harmless, if a user can get this far, that same user should also be able to write JavaScript that performs malicious actions inside the secure area of an unknowing, legitimate user.

The answer to the problem is output escaping. With output escaping on, the same template will render harmlessly, and literally print the `script` tag to the screen:

```
Hello &lt;script&gt;alert(&#39;helloe&#39;)&lt;/script&gt;
```

The Twig and PHP templating systems approach the problem in different ways. If you're using Twig, output escaping is on by default and you're protected. In PHP, output escaping is not automatic, meaning you'll need to manually escape where necessary.

Output Escaping in Twig

If you're using Twig templates, then output escaping is on by default. This means that you're protected out-of-the-box from the unintentional consequences of user-submitted code. By default, the output escaping assumes that content is being escaped for HTML output.

In some cases, you'll need to disable output escaping when you're rendering a variable that is trusted and contains markup that should not be escaped. Suppose that administrative users are able to write articles that contain HTML code. By default, Twig will escape the article body. To render it normally, add the `raw` filter: `{{ article.body|raw }}`.

You can also disable output escaping inside a `{% block %}` area or for an entire template. For more information, see Output Escaping in the Twig documentation.

Output Escaping in PHP

Output escaping is not automatic when using PHP templates. This means that unless you explicitly choose to escape a variable, you're not protected. To use output escaping, use the special `escape()` view method:

```
Hello <?php echo $view->escape($name) ?>
```

By default, the `escape()` method assumes that the variable is being rendered within an HTML context (and thus the variable is escaped to be safe for HTML). The second argument lets you change the context. For example, to output something in a JavaScript string, use the `js` context:

```
var myMsg = 'Hello <?php echo $view->escape($name, 'js') ?>';
```

Debugging

New in version 2.0.9: This feature is available as of Twig `1.5.x`, which was first shipped with Symfony 2.0.9.

When using PHP, you can use `var_dump()` if you need to quickly find the value of a variable passed. This is useful, for example, inside your controller. The same can be achieved when using Twig by using the debug extension. This needs to be enabled in the config:

- *YAML*

```
# app/config/config.yml
services:
    acme_hello.twig.extension.debug:
        class:        Twig_Extension_Debug
        tags:
             - { name: 'twig.extension' }
```

- *XML*

```
<!-- app/config/config.xml -->
<services>
    <service id="acme_hello.twig.extension.debug" class="Twig_Extension_Debug">
        <tag name="twig.extension" />
    </service>
</services>
```

- *PHP*

```
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$definition = new Definition('Twig_Extension_Debug');
$definition->addTag('twig.extension');
$container->setDefinition('acme_hello.twig.extension.debug', $definition);
```

Template parameters can then be dumped using the `dump` function:

```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}

{{ dump(articles) }}

{% for article in articles %}
    <a href="/article/{{ article.slug }}">
        {{ article.title }}
    </a>
{% endfor %}
```

The variables will only be dumped if Twig's `debug` setting (in `config.yml`) is `true`. By default this means that the variables will be dumped in the `dev` environment but not the `prod` environment.

Template Formats

Templates are a generic way to render content in *any* format. And while in most cases you'll use templates to render HTML content, a template can just as easily generate JavaScript, CSS, XML or any other format you can dream of.

For example, the same "resource" is often rendered in several different formats. To render an article index page in XML, simply include the format in the template name:

- *XML template name*: `AcmeArticleBundle:Article:index.xml.twig`
- *XML template filename*: `index.xml.twig`

In reality, this is nothing more than a naming convention and the template isn't actually rendered differently based on its format.

In many cases, you may want to allow a single controller to render multiple different formats based on the "request format". For that reason, a common pattern is to do the following:

```
public function indexAction()
{
    $format = $this->getRequest()->getRequestFormat();

    return $this->render('AcmeBlogBundle:Blog:index.'.$format.'.twig');
}
```

The `getRequestFormat` on the `Request` object defaults to `html`, but can return any other format based on the format requested by the user. The request format is most often managed by the routing, where a route can be configured so that `/contact` sets the request format to `html` while `/contact.xml` sets the format to `xml`. For more information, see the *Advanced Example in the Routing chapter*.

To create links that include the format parameter, include a `_format` key in the parameter hash:

- *Twig*

```
<a href="{{ path('article_show', {'id': 123, '_format': 'pdf'}) }}">
    PDF Version
</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('article_show', array('id' => 123, '_format' => 'p
    PDF Version
</a>
```

Final Thoughts

The templating engine in Symfony is a powerful tool that can be used each time you need to generate presentational content in HTML, XML or any other format. And though templates are a common way to generate content in a controller, their use is not mandatory. The `Response` object returned by a controller can be created with our without the use of a template:

```
// creates a Response object whose content is the rendered template
$response = $this->render('AcmeArticleBundle:Article:index.html.twig');

// creates a Response object whose content is simple text
$response = new Response('response content');
```

Symfony's templating engine is very flexible and two different template renderers are available by default: the traditional *PHP* templates and the sleek and powerful *Twig* templates. Both support a template hierarchy and come packaged with a rich set of helper functions capable of performing the most common tasks.

Overall, the topic of templating should be thought of as a powerful tool that's at your disposal. In some cases, you may not need to render a template, and in Symfony2, that's absolutely fine.

Learn more from the Cookbook

- How to use PHP instead of Twig for Templates
- How to customize Error Pages

2.1.8 Databases and Doctrine ("The Model")

Let's face it, one of the most common and challenging tasks for any application involves persisting and reading information to and from a database. Fortunately, Symfony comes integrated with Doctrine, a library whose sole goal is to give you powerful tools to make this easy. In this chapter, you'll learn the basic philosophy behind Doctrine and see how easy working with a database can be.

Note: Doctrine is totally decoupled from Symfony and using it is optional. This chapter is all about the Doctrine ORM, which aims to let you map objects to a relational database (such as *MySQL*, *PostgreSQL* or *Microsoft SQL*). If you prefer to use raw database queries, this is easy, and explained in the "How to use Doctrine's DBAL Layer" cookbook entry.

You can also persist data to MongoDB using Doctrine ODM library. For more information, read the "`/bundles/DoctrineMongoDBBundle/index`" documentation.

A Simple Example: A Product

The easiest way to understand how Doctrine works is to see it in action. In this section, you'll configure your database, create a `Product` object, persist it to the database and fetch it back out.

Code along with the example

If you want to follow along with the example in this chapter, create an `AcmeStoreBundle` via:

```
php app/console generate:bundle --namespace=Acme/StoreBundle
```

Configuring the Database

Before you really begin, you'll need to configure your database connection information. By convention, this information is usually configured in an `app/config/parameters.yml` file:

```
# app/config/parameters.yml
parameters:
    database_driver:   pdo_mysql
    database_host:     localhost
    database_name:     test_project
    database_user:     root
    database_password: password
```

Note: Defining the configuration via `parameters.yml` is just a convention. The parameters defined in that file are referenced by the main configuration file when setting up Doctrine:

```
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
```

```
        dbname:   %database_name%
        user:     %database_user%
        password: %database_password%
```

By separating the database information into a separate file, you can easily keep different versions of the file on each server. You can also easily store database configuration (or any sensitive information) outside of your project, like inside your Apache configuration, for example. For more information, see How to Set External Parameters in the Service Container.

Now that Doctrine knows about your database, you can have it create the database for you:

```
php app/console doctrine:database:create
```

Creating an Entity Class

Suppose you're building an application where products need to be displayed. Without even thinking about Doctrine or databases, you already know that you need a `Product` object to represent those products. Create this class inside the `Entity` directory of your `AcmeStoreBundle`:

```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

class Product
{
    protected $name;

    protected $price;

    protected $description;
}
```

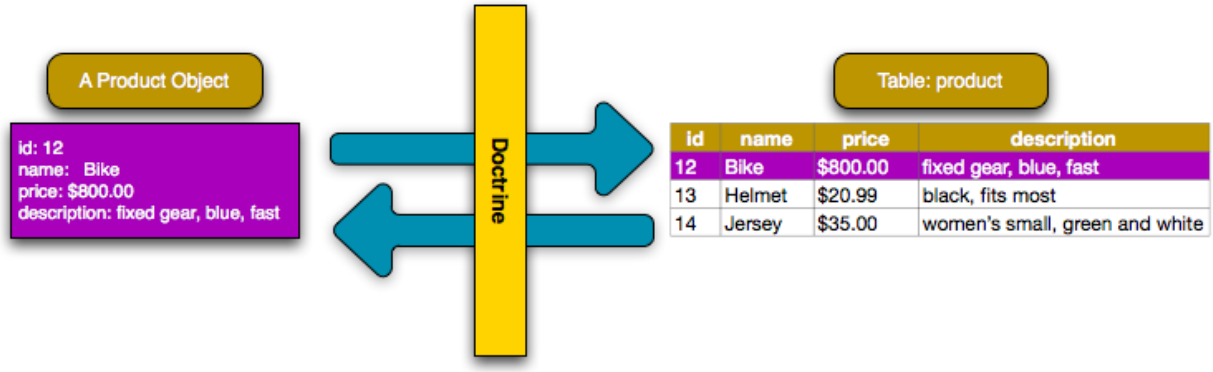
The class - often called an "entity", meaning *a basic class that holds data* - is simple and helps fulfill the business requirement of needing products in your application. This class can't be persisted to a database yet - it's just a simple PHP class.

Tip: Once you learn the concepts behind Doctrine, you can have Doctrine create this entity class for you:

```
php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Product" --fields="name:string(255
```

Add Mapping Information

Doctrine allows you to work with databases in a much more interesting way than just fetching rows of a column-based table into an array. Instead, Doctrine allows you to persist entire *objects* to the database and fetch entire objects out of the database. This works by mapping a PHP class to a database table, and the properties of that PHP class to columns on the table:



For Doctrine to be able to do this, you just have to create "metadata", or configuration that tells Doctrine exactly how the `Product` class and its properties should be *mapped* to the database. This metadata can be specified in a number of different formats including YAML, XML or directly inside the `Product` class via annotations:

Note: A bundle can accept only one metadata definition format. For example, it's not possible to mix YAML metadata definitions with annotated PHP entity class definitions.

- *Annotations*

```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $name;

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $price;

    /**
     * @ORM\Column(type="text")
     */
    protected $description;
}
```

- *YAML*

```yaml
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    table: product
    id:
        id:
            type: integer
            generator: { strategy: AUTO }
    fields:
        name:
            type: string
            length: 100
        price:
            type: decimal
            scale: 2
        description:
            type: text
```

- *XML*

```xml
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
                  http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Acme\StoreBundle\Entity\Product" table="product">
        <id name="id" type="integer" column="id">
            <generator strategy="AUTO" />
        </id>
        <field name="name" column="name" type="string" length="100" />
        <field name="price" column="price" type="decimal" scale="2" />
        <field name="description" column="description" type="text" />
    </entity>
</doctrine-mapping>
```

Tip: The table name is optional and if omitted, will be determined automatically based on the name of the entity class.

Doctrine allows you to choose from a wide variety of different field types, each with their own options. For information on the available field types, see the *Doctrine Field Types Reference* section.

See also:

You can also check out Doctrine's Basic Mapping Documentation for all details about mapping information. If you use annotations, you'll need to prepend all annotations with `ORM\` (e.g. `ORM\Column(..)`), which is not shown in Doctrine's documentation. You'll also need to include the `use Doctrine\ORM\Mapping as ORM;` statement, which *imports* the `ORM` annotations prefix.

Caution: Be careful that your class name and properties aren't mapped to a protected SQL keyword (such as `group` or `user`). For example, if your entity class name is `Group`, then, by default, your table name will be `group`, which will cause an SQL error in some engines. See Doctrine's Reserved SQL keywords documentation on how to properly escape these names.

Note: When using another library or program (ie. Doxygen) that uses annotations, you should place the

`@IgnoreAnnotation` annotation on the class to indicate which annotations Symfony should ignore.

For example, to prevent the `@fn` annotation from throwing an exception, add the following:

```
/**
 * @IgnoreAnnotation("fn")
 */
class Product
```

Generating Getters and Setters

Even though Doctrine now knows how to persist a `Product` object to the database, the class itself isn't really useful yet. Since `Product` is just a regular PHP class, you need to create getter and setter methods (e.g. `getName()`, `setName()`) in order to access its properties (since the properties are `protected`). Fortunately, Doctrine can do this for you by running:

```
php app/console doctrine:generate:entities Acme/StoreBundle/Entity/Product
```

This command makes sure that all of the getters and setters are generated for the `Product` class. This is a safe command - you can run it over and over again: it only generates getters and setters that don't exist (i.e. it doesn't replace your existing methods).

More about `doctrine:generate:entities`

With the `doctrine:generate:entities` command you can:

- generate getters and setters,
- **generate repository classes configured with the** `@ORM\Entity(repositoryClass="...")` annotation,
- generate the appropriate constructor for 1:n and n:m relations.

The `doctrine:generate:entities` command saves a backup of the original `Product.php` named `Product.php~`. In some cases, the presence of this file can cause a "Cannot redeclare class" error. It can be safely removed.

Note that you don't *need* to use this command. Doctrine doesn't rely on code generation. Like with normal PHP classes, you just need to make sure that your protected/private properties have getter and setter methods. Since this is a common thing to do when using Doctrine, this command was created.

You can also generate all known entities (i.e. any PHP class with Doctrine mapping information) of a bundle or an entire namespace:

```
php app/console doctrine:generate:entities AcmeStoreBundle
php app/console doctrine:generate:entities Acme
```

Note: Doctrine doesn't care whether your properties are `protected` or `private`, or whether or not you have a getter or setter function for a property. The getters and setters are generated here only because you'll need them to interact with your PHP object.

Creating the Database Tables/Schema

You now have a usable `Product` class with mapping information so that Doctrine knows exactly how to persist it. Of course, you don't yet have the corresponding `product` table in your database. Fortunately, Doctrine can automatically create all the database tables needed for every known entity in your application. To do this, run:

```
php app/console doctrine:schema:update --force
```

Tip: Actually, this command is incredibly powerful. It compares what your database *should* look like (based on the mapping information of your entities) with how it *actually* looks, and generates the SQL statements needed to *update* the database to where it should be. In other words, if you add a new property with mapping metadata to `Product` and run this task again, it will generate the "alter table" statement needed to add that new column to the existing `product` table.

An even better way to take advantage of this functionality is via `migrations`, which allow you to generate these SQL statements and store them in migration classes that can be run systematically on your production server in order to track and migrate your database schema safely and reliably.

Your database now has a fully-functional `product` table with columns that match the metadata you've specified.

Persisting Objects to the Database

Now that you have a mapped `Product` entity and corresponding `product` table, you're ready to persist data to the database. From inside a controller, this is pretty easy. Add the following method to the `DefaultController` of the bundle:

```
1  // src/Acme/StoreBundle/Controller/DefaultController.php
2  use Acme\StoreBundle\Entity\Product;
3  use Symfony\Component\HttpFoundation\Response;
4  // ...
5
6  public function createAction()
7  {
8      $product = new Product();
9      $product->setName('A Foo Bar');
10     $product->setPrice('19.99');
11     $product->setDescription('Lorem ipsum dolor');
12
13     $em = $this->getDoctrine()->getEntityManager();
14     $em->persist($product);
15     $em->flush();
16
17     return new Response('Created product id '.$product->getId());
18 }
```

Note: If you're following along with this example, you'll need to create a route that points to this action to see it work.

Let's walk through this example:

- **lines 8-11** In this section, you instantiate and work with the `$product` object like any other, normal PHP object;
- **line 13** This line fetches Doctrine's *entity manager* object, which is responsible for handling the process of persisting and fetching objects to and from the database;
- **line 14** The `persist()` method tells Doctrine to "manage" the `$product` object. This does not actually cause a query to be made to the database (yet).
- **line 15** When the `flush()` method is called, Doctrine looks through all of the objects that it's managing to see if they need to be persisted to the database. In this example, the `$product` object has not been persisted yet, so the entity manager executes an `INSERT` query and a row is created in the `product` table.

Note: In fact, since Doctrine is aware of all your managed entities, when you call the `flush()` method, it calculates an overall changeset and executes the most efficient query/queries possible. For example, if you persist a total of 100 `Product` objects and then subsequently call `flush()`, Doctrine will create a *single* prepared statement and re-use it for each insert. This pattern is called *Unit of Work*, and it's used because it's fast and efficient.

When creating or updating objects, the workflow is always the same. In the next section, you'll see how Doctrine is smart enough to automatically issue an `UPDATE` query if the record already exists in the database.

Tip: Doctrine provides a library that allows you to programmatically load testing data into your project (i.e. "fixture data"). For information, see `/bundles/DoctrineFixturesBundle/index`.

Fetching Objects from the Database

Fetching an object back out of the database is even easier. For example, suppose you've configured a route to display a specific `Product` based on its `id` value:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    // do something, like pass the $product object into a template
}
```

When you query for a particular type of object, you always use what's known as its "repository". You can think of a repository as a PHP class whose only job is to help you fetch entities of a certain class. You can access the repository object for an entity class via:

```
$repository = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product');
```

Note: The `AcmeStoreBundle:Product` string is a shortcut you can use anywhere in Doctrine instead of the full class name of the entity (i.e. `Acme\StoreBundle\Entity\Product`). As long as your entity lives under the `Entity` namespace of your bundle, this will work.

Once you have your repository, you have access to all sorts of helpful methods:

```
// query by the primary key (usually "id")
$product = $repository->find($id);

// dynamic method names to find based on a column value
$product = $repository->findOneById($id);
$product = $repository->findOneByName('foo');

// find *all* products
$products = $repository->findAll();

// find a group of products based on an arbitrary column value
$products = $repository->findByPrice(19.99);
```

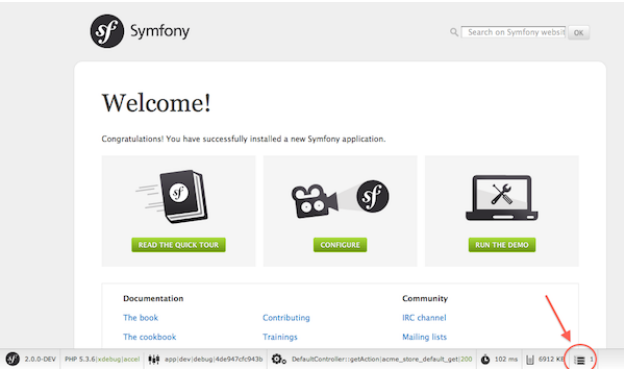
Note: Of course, you can also issue complex queries, which you'll learn more about in the *Querying for Objects* section.

You can also take advantage of the useful `findBy` and `findOneBy` methods to easily fetch objects based on multiple conditions:

```
// query for one product matching be name and price
$product = $repository->findOneBy(array('name' => 'foo', 'price' => 19.99));

// query for all products matching the name, ordered by price
$product = $repository->findBy(
    array('name' => 'foo'),
    array('price' => 'ASC')
);
```

Tip: When you render any page, you can see how many queries were made in the bottom right corner of the web debug toolbar.



If you click the icon, the profiler will open, showing you the exact queries that were made.

Updating an Object

Once you've fetched an object from Doctrine, updating it is easy. Suppose you have a route that maps a product id to an update action in a controller:

```
public function updateAction($id)
{
    $em = $this->getDoctrine()->getEntityManager();
    $product = $em->getRepository('AcmeStoreBundle:Product')->find($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    $product->setName('New product name!');
    $em->flush();

    return $this->redirect($this->generateUrl('homepage'));
}
```

Updating an object involves just three steps:

1. fetching the object from Doctrine;

2. modifying the object;
3. calling `flush()` on the entity manager

Notice that calling `$em->persist($product)` isn't necessary. Recall that this method simply tells Doctrine to manage or "watch" the `$product` object. In this case, since you fetched the `$product` object from Doctrine, it's already managed.

Deleting an Object

Deleting an object is very similar, but requires a call to the `remove()` method of the entity manager:

```
$em->remove($product);
$em->flush();
```

As you might expect, the `remove()` method notifies Doctrine that you'd like to remove the given entity from the database. The actual `DELETE` query, however, isn't actually executed until the `flush()` method is called.

Querying for Objects

You've already seen how the repository object allows you to run basic queries without any work:

```
$repository->find($id);

$repository->findOneByName('Foo');
```

Of course, Doctrine also allows you to write more complex queries using the Doctrine Query Language (DQL). DQL is similar to SQL except that you should imagine that you're querying for one or more objects of an entity class (e.g. `Product`) instead of querying for rows on a table (e.g. `product`).

When querying in Doctrine, you have two options: writing pure Doctrine queries or using Doctrine's Query Builder.

Querying for Objects with DQL

Imagine that you want to query for products, but only return products that cost more than `19.99`, ordered from cheapest to most expensive. From inside a controller, do the following:

```
$em = $this->getDoctrine()->getEntityManager();
$query = $em->createQuery(
    'SELECT p FROM AcmeStoreBundle:Product p WHERE p.price > :price ORDER BY p.price ASC'
)->setParameter('price', '19.99');

$products = $query->getResult();
```

If you're comfortable with SQL, then DQL should feel very natural. The biggest difference is that you need to think in terms of "objects" instead of rows in a database. For this reason, you select *from* `AcmeStoreBundle:Product` and then alias it as `p`.

The `getResult()` method returns an array of results. If you're querying for just one object, you can use the `getSingleResult()` method instead:

```
$product = $query->getSingleResult();
```

Caution: The `getSingleResult()` method throws a `Doctrine\ORM\NoResultException` exception if no results are returned and a `Doctrine\ORM\NonUniqueResultException` if *more* than one result is returned. If you use this method, you may need to wrap it in a try-catch block and ensure that only one result is returned (if you're querying on something that could feasibly return more than one result):

```
$query = $em->createQuery('SELECT ....')
    ->setMaxResults(1);

try {
    $product = $query->getSingleResult();
} catch (\Doctrine\Orm\NoResultException $e) {
    $product = null;
}
// ...
```

The DQL syntax is incredibly powerful, allowing you to easily join between entities (the topic of *relations* will be covered later), group, etc. For more information, see the official Doctrine Doctrine Query Language documentation.

Setting Parameters

Take note of the `setParameter()` method. When working with Doctrine, it's always a good idea to set any external values as "placeholders", which was done in the above query:

```
... WHERE p.price > :price ...
```

You can then set the value of the `price` placeholder by calling the `setParameter()` method:

```
->setParameter('price', '19.99')
```

Using parameters instead of placing values directly in the query string is done to prevent SQL injection attacks and should *always* be done. If you're using multiple parameters, you can set their values at once using the `setParameters()` method:

```
->setParameters(array(
    'price' => '19.99',
    'name'  => 'Foo',
))
```

Using Doctrine's Query Builder

Instead of writing the queries directly, you can alternatively use Doctrine's `QueryBuilder` to do the same job using a nice, object-oriented interface. If you use an IDE, you can also take advantage of auto-completion as you type the method names. From inside a controller:

```
$repository = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product');

$query = $repository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
    ->orderBy('p.price', 'ASC')
    ->getQuery();

$products = $query->getResult();
```

The `QueryBuilder` object contains every method necessary to build your query. By calling the `getQuery()`

method, the query builder returns a normal `Query` object, which is the same object you built directly in the previous section.

For more information on Doctrine's Query Builder, consult Doctrine's Query Builder documentation.

Custom Repository Classes

In the previous sections, you began constructing and using more complex queries from inside a controller. In order to isolate, test and reuse these queries, it's a good idea to create a custom repository class for your entity and add methods with your query logic there.

To do this, add the name of the repository class to your mapping definition.

- *Annotations*

```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="Acme\StoreBundle\Repository\ProductRepository")
 */
class Product
{
    //...
}
```

- *YAML*

```
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    repositoryClass: Acme\StoreBundle\Repository\ProductRepository
    # ...
```

- *XML*

```
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<!-- ... -->
<doctrine-mapping>

    <entity name="Acme\StoreBundle\Entity\Product"
            repository-class="Acme\StoreBundle\Repository\ProductRepository">
            <!-- ... -->
    </entity>
</doctrine-mapping>
```

Doctrine can generate the repository class for you by running the same command used earlier to generate the missing getter and setter methods:

```
php app/console doctrine:generate:entities Acme
```

Next, add a new method - `findAllOrderedByName()` - to the newly generated repository class. This method will query for all of the `Product` entities, ordered alphabetically.

```
// src/Acme/StoreBundle/Repository/ProductRepository.php
namespace Acme\StoreBundle\Repository;

```

```
use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery('SELECT p FROM AcmeStoreBundle:Product p ORDER BY p.name ASC')
            ->getResult();
    }
}
```

Tip: The entity manager can be accessed via `$this->getEntityManager()` from inside the repository.

You can use this new method just like the default finder methods of the repository:

```
$em = $this->getDoctrine()->getEntityManager();
$products = $em->getRepository('AcmeStoreBundle:Product')
            ->findAllOrderedByName();
```

Note: When using a custom repository class, you still have access to the default finder methods such as `find()` and `findAll()`.

Entity Relationships/Associations

Suppose that the products in your application all belong to exactly one "category". In this case, you'll need a `Category` object and a way to relate a `Product` object to a `Category` object. Start by creating the `Category` entity. Since you know that you'll eventually need to persist the class through Doctrine, you can let Doctrine create the class for you.

```
php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Category" --fields="name:string(25
```

This task generates the `Category` entity for you, with an `id` field, a `name` field and the associated getter and setter functions.

Relationship Mapping Metadata

To relate the `Category` and `Product` entities, start by creating a `products` property on the `Category` class:

- *Annotations*

```
// src/Acme/StoreBundle/Entity/Category.php
// ...
use Doctrine\Common\Collections\ArrayCollection;

class Category
{
    // ...

    /**
     * @ORM\OneToMany(targetEntity="Product", mappedBy="category")
     */
    protected $products;
```

```
        public function __construct()
        {
            $this->products = new ArrayCollection();
        }
    }
```

- *YAML*

```
    # src/Acme/StoreBundle/Resources/config/doctrine/Category.orm.yml
    Acme\StoreBundle\Entity\Category:
        type: entity
        # ...
        oneToMany:
            products:
                targetEntity: Product
                mappedBy: category
        # don't forget to init the collection in entity __construct() method
```

First, since a `Category` object will relate to many `Product` objects, a `products` array property is added to hold those `Product` objects. Again, this isn't done because Doctrine needs it, but instead because it makes sense in the application for each `Category` to hold an array of `Product` objects.

Note: The code in the `__construct()` method is important because Doctrine requires the `$products` property to be an `ArrayCollection` object. This object looks and acts almost *exactly* like an array, but has some added flexibility. If this makes you uncomfortable, don't worry. Just imagine that it's an `array` and you'll be in good shape.

Tip: The targetEntity value in the decorator used above can reference any entity with a valid namespace, not just entities defined in the same class. To relate to an entity defined in a different class or bundle, enter a full namespace as the targetEntity.

Next, since each `Product` class can relate to exactly one `Category` object, you'll want to add a `$category` property to the `Product` class:

- *Annotations*

```
    // src/Acme/StoreBundle/Entity/Product.php
    // ...

    class Product
    {
        // ...

        /**
         * @ORM\ManyToOne(targetEntity="Category", inversedBy="products")
         * @ORM\JoinColumn(name="category_id", referencedColumnName="id")
         */
        protected $category;
    }
```

- *YAML*

```
    # src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
    Acme\StoreBundle\Entity\Product:
        type: entity
        # ...
        manyToOne:
            category:
                targetEntity: Category
```

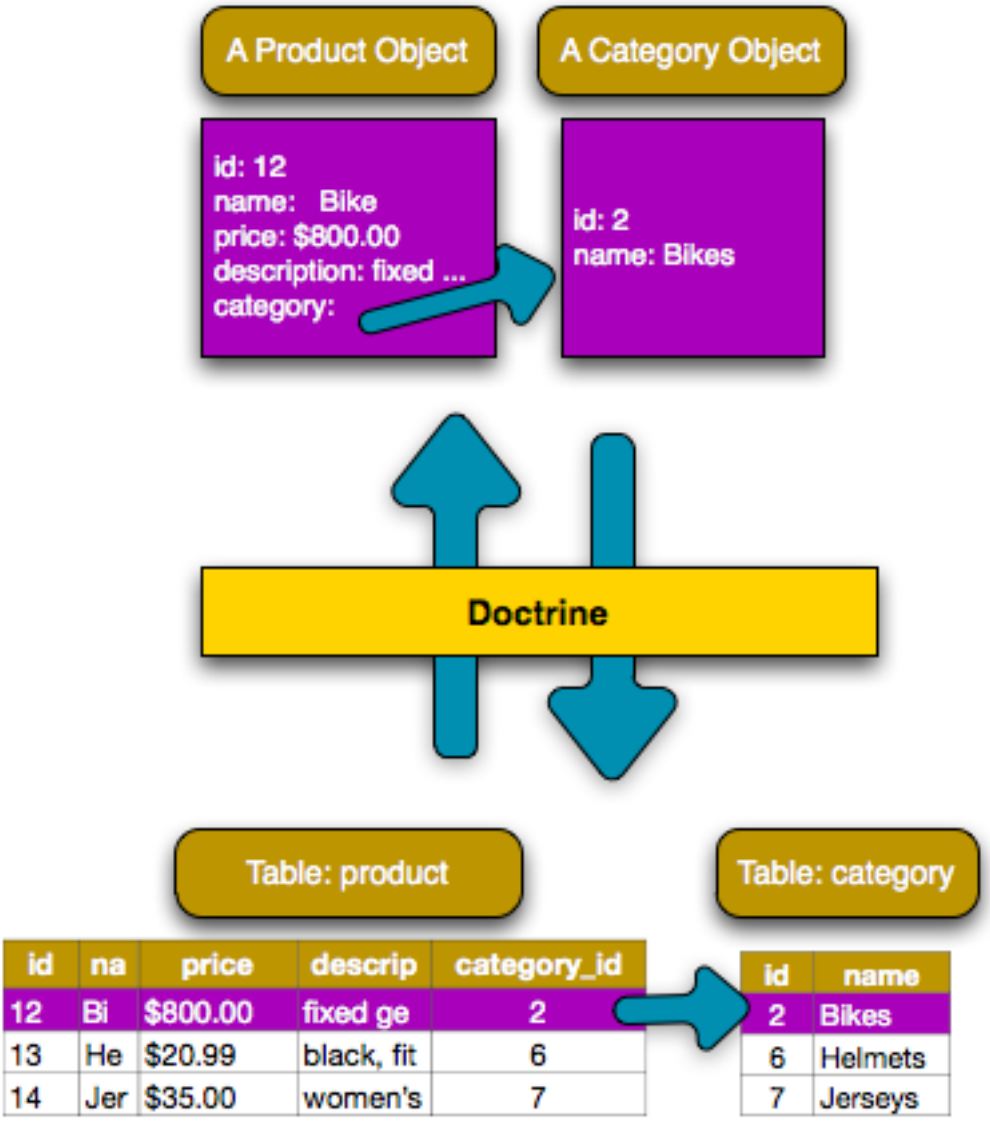
```
                inversedBy: products
                joinColumn:
                    name: category_id
                    referencedColumnName: id
```

Finally, now that you've added a new property to both the `Category` and `Product` classes, tell Doctrine to generate the missing getter and setter methods for you:

```
php app/console doctrine:generate:entities Acme
```

Ignore the Doctrine metadata for a moment. You now have two classes - `Category` and `Product` with a natural one-to-many relationship. The `Category` class holds an array of `Product` objects and the `Product` object can hold one `Category` object. In other words - you've built your classes in a way that makes sense for your needs. The fact that the data needs to be persisted to a database is always secondary.

Now, look at the metadata above the `$category` property on the `Product` class. The information here tells doctrine that the related class is `Category` and that it should store the `id` of the category record on a `category_id` field that lives on the `product` table. In other words, the related `Category` object will be stored on the `$category` property, but behind the scenes, Doctrine will persist this relationship by storing the category's id value on a `category_id` column of the `product` table.



The metadata above the `$products` property of the `Category` object is less important, and simply tells Doctrine to look at the `Product.category` property to figure out how the relationship is mapped.

Before you continue, be sure to tell Doctrine to add the new `category` table, and `product.category_id` column, and new foreign key:

```
php app/console doctrine:schema:update --force
```

Note: This task should only be really used during development. For a more robust method of systematically updating your production database, read about `Doctrine migrations`.

Saving Related Entities

Now, let's see the code in action. Imagine you're inside a controller:

```
// ...
use Acme\StoreBundle\Entity\Category;
use Acme\StoreBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

class DefaultController extends Controller
{
    public function createProductAction()
    {
        $category = new Category();
        $category->setName('Main Products');

        $product = new Product();
        $product->setName('Foo');
        $product->setPrice(19.99);
        // relate this product to the category
        $product->setCategory($category);

        $em = $this->getDoctrine()->getEntityManager();
        $em->persist($category);
        $em->persist($product);
        $em->flush();

        return new Response(
            'Created product id: '.$product->getId().' and category id: '.$category->getId()
        );
    }
}
```

Now, a single row is added to both the `category` and `product` tables. The `product.category_id` column for the new product is set to whatever the `id` is of the new category. Doctrine manages the persistence of this relationship for you.

Fetching Related Objects

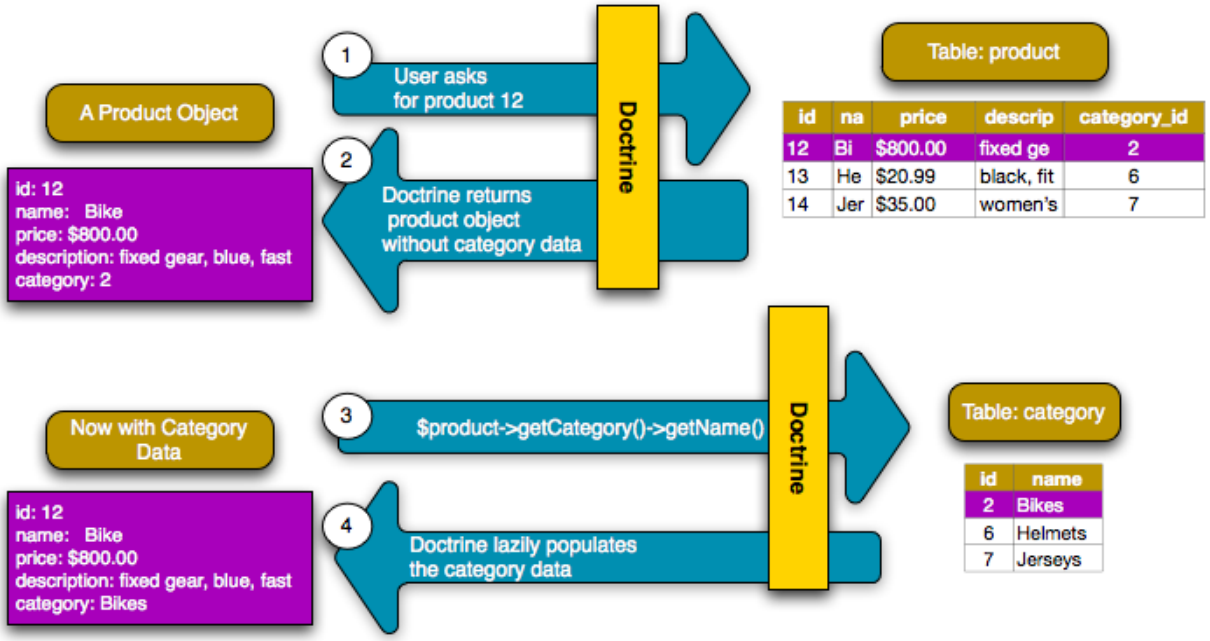
When you need to fetch associated objects, your workflow looks just like it did before. First, fetch a `$product` object and then access its related `Category`:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    $categoryName = $product->getCategory()->getName();

    // ...
}
```

In this example, you first query for a `Product` object based on the product's `id`. This issues a query for *just* the product data and hydrates the `$product` object with that data. Later, when you call `$product->getCategory()->getName()`, Doctrine silently makes a second query to find the `Category` that's related to this `Product`. It prepares the `$category` object and returns it to you.



What's important is the fact that you have easy access to the product's related category, but the category data isn't actually retrieved until you ask for the category (i.e. it's "lazily loaded").

You can also query in the other direction:

```
public function showProductAction($id)
{
    $category = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Category')
        ->find($id);

    $products = $category->getProducts();

    // ...
}
```

In this case, the same things occurs: you first query out for a single `Category` object, and then Doctrine makes a second query to retrieve the related `Product` objects, but only once/if you ask for them (i.e. when you call `->getProducts()`). The `$products` variable is an array of all `Product` objects that relate to the given `Category` object via their `category_id` value.

Relationships and Proxy Classes

This "lazy loading" is possible because, when necessary, Doctrine returns a "proxy" object in place of the true object. Look again at the above example:

```
$product = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product')
    ->find($id);

$category = $product->getCategory();

// prints "Proxies\AcmeStoreBundleEntityCategoryProxy"
echo get_class($category);
```

This proxy object extends the true `Category` object, and looks and acts exactly like it. The difference is that, by using a proxy object, Doctrine can delay querying for the real `Category` data until you actually need that data (e.g. until you call `$category->getName()`).
The proxy classes are generated by Doctrine and stored in the cache directory. And though you'll probably never even notice that your `$category` object is actually a proxy object, it's important to keep in mind.
In the next section, when you retrieve the product and category data all at once (via a *join*), Doctrine will return the *true* `Category` object, since nothing needs to be lazily loaded.

Joining to Related Records

In the above examples, two queries were made - one for the original object (e.g. a `Category`) and one for the related object(s) (e.g. the `Product` objects).

Tip: Remember that you can see all of the queries made during a request via the web debug toolbar.

Of course, if you know up front that you'll need to access both objects, you can avoid the second query by issuing a join in the original query. Add the following method to the `ProductRepository` class:

```
// src/Acme/StoreBundle/Repository/ProductRepository.php

public function findOneByIdJoinedToCategory($id)
{
    $query = $this->getEntityManager()
        ->createQuery('
            SELECT p, c FROM AcmeStoreBundle:Product p
            JOIN p.category c
            WHERE p.id = :id'
        )->setParameter('id', $id);

    try {
        return $query->getSingleResult();
    } catch (\Doctrine\ORM\NoResultException $e) {
        return null;
    }
}
```

Now, you can use this method in your controller to query for a `Product` object and its related `Category` with just one query:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
```

```
        ->getRepository('AcmeStoreBundle:Product')
        ->findOneByIdJoinedToCategory($id);

    $category = $product->getCategory();

    // ...
}
```

More Information on Associations

This section has been an introduction to one common type of entity relationship, the one-to-many relationship. For more advanced details and examples of how to use other types of relations (e.g. `one-to-one`, `many-to-many`), see Doctrine's Association Mapping Documentation.

Note: If you're using annotations, you'll need to prepend all annotations with `ORM\` (e.g. `ORM\OneToMany`), which is not reflected in Doctrine's documentation. You'll also need to include the `use Doctrine\ORM\Mapping as ORM;` statement, which *imports* the `ORM` annotations prefix.

Configuration

Doctrine is highly configurable, though you probably won't ever need to worry about most of its options. To find out more about configuring Doctrine, see the Doctrine section of the reference manual.

Lifecycle Callbacks

Sometimes, you need to perform an action right before or after an entity is inserted, updated, or deleted. These types of actions are known as "lifecycle" callbacks, as they're callback methods that you need to execute during different stages of the lifecycle of an entity (e.g. the entity is inserted, updated, deleted, etc).

If you're using annotations for your metadata, start by enabling the lifecycle callbacks. This is not necessary if you're using YAML or XML for your mapping:

```
/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Product
{
    // ...
}
```

Now, you can tell Doctrine to execute a method on any of the available lifecycle events. For example, suppose you want to set a `created` date column to the current date, only when the entity is first persisted (i.e. inserted):

- *Annotations*

```
    /**
     * @ORM\prePersist
     */
    public function setCreatedValue()
    {
        $this->created = new \DateTime();
    }
```

- *YAML*

```yaml
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    # ...
    lifecycleCallbacks:
        prePersist: [ setCreatedValue ]
```

- *XML*

```xml
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<!-- ... -->
<doctrine-mapping>

    <entity name="Acme\StoreBundle\Entity\Product">
            <!-- ... -->
            <lifecycle-callbacks>
                <lifecycle-callback type="prePersist" method="setCreatedValue" />
            </lifecycle-callbacks>
    </entity>
</doctrine-mapping>
```

Note: The above example assumes that you've created and mapped a `created` property (not shown here).

Now, right before the entity is first persisted, Doctrine will automatically call this method and the `created` field will be set to the current date.

This can be repeated for any of the other lifecycle events, which include:

- `preRemove`
- `postRemove`
- `prePersist`
- `postPersist`
- `preUpdate`
- `postUpdate`
- `postLoad`
- `loadClassMetadata`

For more information on what these lifecycle events mean and lifecycle callbacks in general, see Doctrine's Lifecycle Events documentation

Lifecycle Callbacks and Event Listeners

Notice that the `setCreatedValue()` method receives no arguments. This is always the case for lifecylce callbacks and is intentional: lifecycle callbacks should be simple methods that are concerned with internally transforming data in the entity (e.g. setting a created/updated field, generating a slug value).
If you need to do some heavier lifting - like perform logging or send an email - you should register an external class as an event listener or subscriber and give it access to whatever resources you need. For more information, see Registering Event Listeners and Subscribers.

Doctrine Extensions: Timestampable, Sluggable, etc.

Doctrine is quite flexible, and a number of third-party extensions are available that allow you to easily perform repeated and common tasks on your entities. These include thing such as *Sluggable*, *Timestampable*, *Loggable*, *Translatable*, and *Tree*.

For more information on how to find and use these extensions, see the cookbook article about using common Doctrine extensions.

Doctrine Field Types Reference

Doctrine comes with a large number of field types available. Each of these maps a PHP data type to a specific column type in whatever database you're using. The following types are supported in Doctrine:

- **Strings**
  - `string` (used for shorter strings)
  - `text` (used for larger strings)
- **Numbers**
  - `integer`
  - `smallint`
  - `bigint`
  - `decimal`
  - `float`
- **Dates and Times** (use a DateTime object for these fields in PHP)
  - `date`
  - `time`
  - `datetime`
- **Other Types**
  - `boolean`
  - `object` (serialized and stored in a `CLOB` field)
  - `array` (serialized and stored in a `CLOB` field)

For more information, see Doctrine's Mapping Types documentation.

Field Options

Each field can have a set of options applied to it. The available options include `type` (defaults to `string`), `name`, `length`, `unique` and `nullable`. Take a few examples:

- *Annotations*

```
/**
 * A string field with length 255 that cannot be null
 * (reflecting the default values for the "type", "length" and *nullable* options)
 *
 * @ORM\Column()
 */
```

```
    protected $name;

    /**
     * A string field of length 150 that persists to an "email_address" column
     * and has a unique index.
     *
     * @ORM\Column(name="email_address", unique="true", length="150")
     */
    protected $email;
```

- *YAML*

```
    fields:
        # A string field length 255 that cannot be null
        # (reflecting the default values for the "length" and *nullable* options)
        # type attribute is necessary in yaml definitions
        name:
            type: string

        # A string field of length 150 that persists to an "email_address" column
        # and has a unique index.
        email:
            type: string
            column: email_address
            length: 150
            unique: true
```

Note: There are a few more options not listed here. For more details, see Doctrine's Property Mapping documentation

Console Commands

The Doctrine2 ORM integration offers several console commands under the `doctrine` namespace. To view the command list you can run the console without any arguments:

```
php app/console
```

A list of available command will print out, many of which start with the `doctrine:` prefix. You can find out more information about any of these commands (or any Symfony command) by running the `help` command. For example, to get details about the `doctrine:database:create` task, run:

```
php app/console help doctrine:database:create
```

Some notable or interesting tasks include:

- `doctrine:ensure-production-settings` - checks to see if the current environment is configured efficiently for production. This should always be run in the `prod` environment:

```
    php app/console doctrine:ensure-production-settings --env=prod
```

- `doctrine:mapping:import` - allows Doctrine to introspect an existing database and create mapping information. For more information, see How to generate Entities from an Existing Database.
- `doctrine:mapping:info` - tells you all of the entities that Doctrine is aware of and whether or not there are any basic errors with the mapping.
- `doctrine:query:dql` and `doctrine:query:sql` - allow you to execute DQL or SQL queries directly from the command line.

Note: To be able to load data fixtures to your database, you will need to have the `DoctrineFixturesBundle` bundle installed. To learn how to do it, read the "`/bundles/DoctrineFixturesBundle/index`" entry of the documentation.

Summary

With Doctrine, you can focus on your objects and how they're useful in your application and worry about database persistence second. This is because Doctrine allows you to use any PHP object to hold your data and relies on mapping metadata information to map an object's data to a particular database table.

And even though Doctrine revolves around a simple concept, it's incredibly powerful, allowing you to create complex queries and subscribe to events that allow you to take different actions as objects go through their persistence lifecycle.

For more information about Doctrine, see the *Doctrine* section of the cookbook, which includes the following articles:

- `/bundles/DoctrineFixturesBundle/index`
- Doctrine Extensions: Timestampable: Sluggable, Translatable, etc.

2.1.9 Testing

Whenever you write a new line of code, you also potentially add new bugs. To build better and more reliable applications, you should test your code using both functional and unit tests.

The PHPUnit Testing Framework

Symfony2 integrates with an independent library - called PHPUnit - to give you a rich testing framework. This chapter won't cover PHPUnit itself, but it has its own excellent documentation.

Note: Symfony2 works with PHPUnit 3.5.11 or later, though version 3.6.4 is needed to test the Symfony core code itself.

Each test - whether it's a unit test or a functional test - is a PHP class that should live in the *Tests/* subdirectory of your bundles. If you follow this rule, then you can run all of your application's tests with the following command:

```
# specify the configuration directory on the command line
$ phpunit -c app/
```

The `-c` option tells PHPUnit to look in the `app/` directory for a configuration file. If you're curious about the PHPUnit options, check out the `app/phpunit.xml.dist` file.

Tip: Code coverage can be generated with the `--coverage-html` option.

Unit Tests

A unit test is usually a test against a specific PHP class. If you want to test the overall behavior of your application, see the section about *Functional Tests*.

Writing Symfony2 unit tests is no different than writing standard PHPUnit unit tests. Suppose, for example, that you have an *incredibly* simple class called `Calculator` in the `Utility/` directory of your bundle:

```
// src/Acme/DemoBundle/Utility/Calculator.php
namespace Acme\DemoBundle\Utility;

class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

To test this, create a `CalculatorTest` file in the `Tests/Utility` directory of your bundle:

```
// src/Acme/DemoBundle/Tests/Utility/CalculatorTest.php
namespace Acme\DemoBundle\Tests\Utility;

use Acme\DemoBundle\Utility\Calculator;

class CalculatorTest extends \PHPUnit_Framework_TestCase
{
    public function testAdd()
    {
        $calc = new Calculator();
        $result = $calc->add(30, 12);

        // assert that our calculator added the numbers correctly!
        $this->assertEquals(42, $result);
    }
}
```

Note: By convention, the `Tests/` sub-directory should replicate the directory of your bundle. So, if you're testing a class in your bundle's `Utility/` directory, put the test in the `Tests/Utility/` directory.

Just like in your real application - autoloading is automatically enabled via the `bootstrap.php.cache` file (as configured by default in the `phpunit.xml.dist` file).

Running tests for a given file or directory is also very easy:

```
# run all tests in the Utility directory
$ phpunit -c app src/Acme/DemoBundle/Tests/Utility/

# run tests for the Calculator class
$ phpunit -c app src/Acme/DemoBundle/Tests/Utility/CalculatorTest.php

# run all tests for the entire Bundle
$ phpunit -c app src/Acme/DemoBundle/
```

Functional Tests

Functional tests check the integration of the different layers of an application (from the routing to the views). They are no different from unit tests as far as PHPUnit is concerned, but they have a very specific workflow:

- Make a request;
- Test the response;
- Click on a link or submit a form;

- Test the response;
- Rinse and repeat.

Your First Functional Test

Functional tests are simple PHP files that typically live in the `Tests/Controller` directory of your bundle. If you want to test the pages handled by your `DemoController` class, start by creating a new `DemoControllerTest.php` file that extends a special `WebTestCase` class.

For example, the Symfony2 Standard Edition provides a simple functional test for its `DemoController` (DemoControllerTest) that reads as follows:

```
// src/Acme/DemoBundle/Tests/Controller/DemoControllerTest.php
namespace Acme\DemoBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DemoControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();

        $crawler = $client->request('GET', '/demo/hello/Fabien');

        $this->assertTrue($crawler->filter('html:contains("Hello Fabien")')->count() > 0);
    }
}
```

Tip: To run your functional tests, the `WebTestCase` class bootstraps the kernel of your application. In most cases, this happens automatically. However, if your kernel is in a non-standard directory, you'll need to modify your `phpunit.xml.dist` file to set the `KERNEL_DIR` environment variable to the directory of your kernel:

```
<phpunit
    <!-- ... -->
    <php>
        <server name="KERNEL_DIR" value="/path/to/your/app/" />
    </php>
    <!-- ... -->
</phpunit>
```

The `createClient()` method returns a client, which is like a browser that you'll use to crawl your site:

```
$crawler = $client->request('GET', '/demo/hello/Fabien');
```

The `request()` method (see *more about the request method*) returns a `Symfony\Component\DomCrawler\Crawler` object which can be used to select elements in the Response, click on links, and submit forms.

Tip: The Crawler only works when the response is an XML or an HTML document. To get the raw content response, call `$client->getResponse()->getContent()`.

Click on a link by first selecting it with the Crawler using either an XPath expression or a CSS selector, then use the Client to click on it. For example, the following code finds all links with the text `Greet`, then selects the second one, and ultimately clicks on it:

```
$link = $crawler->filter('a:contains("Greet")')->eq(1)->link();

$crawler = $client->click($link);
```

Submitting a form is very similar; select a form button, optionally override some form values, and submit the corresponding form:

```
$form = $crawler->selectButton('submit')->form();

// set some values
$form['name'] = 'Lucas';
$form['form_name[subject]'] = 'Hey there!';

// submit the form
$crawler = $client->submit($form);
```

Tip: The form can also handle uploads and contains methods to fill in different types of form fields (e.g. `select()` and `tick()`). For details, see the *Forms* section below.

Now that you can easily navigate through an application, use assertions to test that it actually does what you expect it to. Use the Crawler to make assertions on the DOM:

```
// Assert that the response matches a given CSS selector.
$this->assertTrue($crawler->filter('h1')->count() > 0);
```

Or, test against the Response content directly if you just want to assert that the content contains some text, or if the Response is not an XML/HTML document:

```
$this->assertRegExp('/Hello Fabien/', $client->getResponse()->getContent());
```

More about the `request()` method:

The full signature of the `request()` method is:

```
request(
    $method,
    $uri,
    array $parameters = array(),
    array $files = array(),
    array $server = array(),
    $content = null,
    $changeHistory = true
)
```

The `server` array is the raw values that you'd expect to normally find in the PHP $_SERVER superglobal. For example, to set the *Content-Type* and *Referer* HTTP headers, you'd pass the following:

```
$client->request(
    'GET',
    '/demo/hello/Fabien',
    array(),
    array(),
    array(
        'CONTENT_TYPE' => 'application/json',
        'HTTP_REFERER' => '/foo/bar',
    )
);
```

Working with the Test Client

The Test Client simulates an HTTP client like a browser and makes requests into your Symfony2 application:

```
$crawler = $client->request('GET', '/hello/Fabien');
```

The `request()` method takes the HTTP method and a URL as arguments and returns a `Crawler` instance.

Use the Crawler to find DOM elements in the Response. These elements can then be used to click on links and submit forms:

```
$link = $crawler->selectLink('Go elsewhere...')->link();
$crawler = $client->click($link);

$form = $crawler->selectButton('validate')->form();
$crawler = $client->submit($form, array('name' => 'Fabien'));
```

The `click()` and `submit()` methods both return a `Crawler` object. These methods are the best way to browse your application as it takes care of a lot of things for you, like detecting the HTTP method from a form and giving you a nice API for uploading files.

Tip: You will learn more about the `Link` and `Form` objects in the *Crawler* section below.

The `request` method can also be used to simulate form submissions directly or perform more complex requests:

```
// Directly submit a form (but using the Crawler is easier!)
$client->request('POST', '/submit', array('name' => 'Fabien'));

// Form submission with a file upload
use Symfony\Component\HttpFoundation\File\UploadedFile;

$photo = new UploadedFile(
    '/path/to/photo.jpg',
    'photo.jpg',
    'image/jpeg',
    123
);
// or
$photo = array(
    'tmp_name' => '/path/to/photo.jpg',
    'name' => 'photo.jpg',
    'type' => 'image/jpeg',
    'size' => 123,
    'error' => UPLOAD_ERR_OK
);
$client->request(
    'POST',
    '/submit',
    array('name' => 'Fabien'),
    array('photo' => $photo)
);

// Perform a DELETE requests, and pass HTTP headers
$client->request(
    'DELETE',
    '/post/12',
    array(),
    array(),
```

```
    array('PHP_AUTH_USER' => 'username', 'PHP_AUTH_PW' => 'pa$$word')
);
```

Last but not least, you can force each request to be executed in its own PHP process to avoid any side-effects when working with several clients in the same script:

```
$client->insulate();
```

Browsing

The Client supports many operations that can be done in a real browser:

```
$client->back();
$client->forward();
$client->reload();

// Clears all cookies and the history
$client->restart();
```

Accessing Internal Objects

If you use the client to test your application, you might want to access the client's internal objects:

```
$history   = $client->getHistory();
$cookieJar = $client->getCookieJar();
```

You can also get the objects related to the latest request:

```
$request  = $client->getRequest();
$response = $client->getResponse();
$crawler  = $client->getCrawler();
```

If your requests are not insulated, you can also access the `Container` and the `Kernel`:

```
$container = $client->getContainer();
$kernel    = $client->getKernel();
```

Accessing the Container

It's highly recommended that a functional test only tests the Response. But under certain very rare circumstances, you might want to access some internal objects to write assertions. In such cases, you can access the dependency injection container:

```
$container = $client->getContainer();
```

Be warned that this does not work if you insulate the client or if you use an HTTP layer. For a list of services available in your application, use the `container:debug` console task.

Tip: If the information you need to check is available from the profiler, use it instead.

Accessing the Profiler Data

On each request, the Symfony profiler collects and stores a lot of data about the internal handling of that request. For example, the profiler could be used to verify that a given page executes less than a certain number of database queries when loading.

To get the Profiler for the last request, do the following:

```
$profile = $client->getProfile();
```

For specific details on using the profiler inside a test, see the How to use the Profiler in a Functional Test cookbook entry.

Redirecting

When a request returns a redirect response, the client automatically follows it. If you want to examine the Response before redirecting, you can force the client to not follow redirects with the `followRedirects()` method:

```
$client->followRedirects(false);
```

When the client does not follow redirects, you can force the redirection with the `followRedirect()` method:

```
$crawler = $client->followRedirect();
```

The Crawler

A Crawler instance is returned each time you make a request with the Client. It allows you to traverse HTML documents, select nodes, find links and forms.

Traversing

Like jQuery, the Crawler has methods to traverse the DOM of an HTML/XML document. For example, the following finds all `input[type=submit]` elements, selects the last one on the page, and then selects its immediate parent element:

```
$newCrawler = $crawler->filter('input[type=submit]')
    ->last()
    ->parents()
    ->first()
;
```

Many other methods are also available:

Method	Description
`filter('h1.title')`	Nodes that match the CSS selector
`filterXpath('h1')`	Nodes that match the XPath expression
`eq(1)`	Node for the specified index
`first()`	First node
`last()`	Last node
`siblings()`	Siblings
`nextAll()`	All following siblings
`previousAll()`	All preceding siblings
`parents()`	Returns the parent nodes
`children()`	Returns children nodes
`reduce($lambda)`	Nodes for which the callable does not return false

Since each of these methods returns a new `Crawler` instance, you can narrow down your node selection by chaining the method calls:

```
$crawler
    ->filter('h1')
    ->reduce(function ($node, $i)
    {
        if (!$node->getAttribute('class')) {
            return false;
        }
    })
    ->first();
```

Tip: Use the `count()` function to get the number of nodes stored in a Crawler: `count($crawler)`

Extracting Information

The Crawler can extract information from the nodes:

```
// Returns the attribute value for the first node
$crawler->attr('class');

// Returns the node value for the first node
$crawler->text();

// Extracts an array of attributes for all nodes (_text returns the node value)
// returns an array for each element in crawler, each with the value and href
$info = $crawler->extract(array('_text', 'href'));

// Executes a lambda for each node and return an array of results
$data = $crawler->each(function ($node, $i)
{
    return $node->attr('href');
});
```

Links

To select links, you can use the traversing methods above or the convenient `selectLink()` shortcut:

```
$crawler->selectLink('Click here');
```

This selects all links that contain the given text, or clickable images for which the `alt` attribute contains the given text. Like the other filtering methods, this returns another `Crawler` object.

Once you've selected a link, you have access to a special `Link` object, which has helpful methods specific to links (such as `getMethod()` and `getUri()`). To click on the link, use the Client's `click()` method and pass it a `Link` object:

```
$link = $crawler->selectLink('Click here')->link();

$client->click($link);
```

Forms

Just like links, you select forms with the `selectButton()` method:

```
$buttonCrawlerNode = $crawler->selectButton('submit');
```

Note: Notice that we select form buttons and not forms as a form can have several buttons; if you use the traversing API, keep in mind that you must look for a button.

The `selectButton()` method can select `button` tags and submit `input` tags. It uses several different parts of the buttons to find them:

- The `value` attribute value;
- The `id` or `alt` attribute value for images;
- The `id` or `name` attribute value for `button` tags.

Once you have a Crawler representing a button, call the `form()` method to get a `Form` instance for the form wrapping the button node:

```
$form = $buttonCrawlerNode->form();
```

When calling the `form()` method, you can also pass an array of field values that overrides the default ones:

```
$form = $buttonCrawlerNode->form(array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

And if you want to simulate a specific HTTP method for the form, pass it as a second argument:

```
$form = $buttonCrawlerNode->form(array(), 'DELETE');
```

The Client can submit `Form` instances:

```
$client->submit($form);
```

The field values can also be passed as a second argument of the `submit()` method:

```
$client->submit($form, array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

For more complex situations, use the `Form` instance as an array to set the value of each field individually:

```
// Change the value of a field
$form['name'] = 'Fabien';
$form['my_form[subject]'] = 'Symfony rocks!';
```

There is also a nice API to manipulate the values of the fields according to their type:

```
// Select an option or a radio
$form['country']->select('France');

// Tick a checkbox
$form['like_symfony']->tick();

// Upload a file
$form['photo']->upload('/path/to/lucas.jpg');
```

Tip: You can get the values that will be submitted by calling the `getValues()` method on the `Form` object. The uploaded files are available in a separate array returned by `getFiles()`. The `getPhpValues()` and `getPhpFiles()` methods also return the submitted values, but in the PHP format (it converts the keys with square brackets notation - e.g. `my_form[subject]` - to PHP arrays).

Testing Configuration

The Client used by functional tests creates a Kernel that runs in a special `test` environment. Since Symfony loads the `app/config/config_test.yml` in the `test` environment, you can tweak any of your application's settings specifically for testing.

For example, by default, the swiftmailer is configured to *not* actually deliver emails in the `test` environment. You can see this under the `swiftmailer` configuration option:

- *YAML*

```
# app/config/config_test.yml
# ...

swiftmailer:
    disable_delivery: true
```

- *XML*

```
<!-- app/config/config_test.xml -->
<container>
    <!-- ... -->

    <swiftmailer:config disable-delivery="true" />
</container>
```

- *PHP*

```
// app/config/config_test.php
// ...

$container->loadFromExtension('swiftmailer', array(
    'disable_delivery' => true
));
```

You can also use a different environment entirely, or override the default debug mode (`true`) by passing each as options to the `createClient()` method:

```
$client = static::createClient(array(
    'environment' => 'my_test_env',
    'debug'       => false,
));
```

If your application behaves according to some HTTP headers, pass them as the second argument of `createClient()`:

```
$client = static::createClient(array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

You can also override HTTP headers on a per request basis:

```
$client->request('GET', '/', array(), array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

Tip: The test client is available as a service in the container in the `test` environment (or wherever the *framework.test* option is enabled). This means you can override the service entirely if you need to.

PHPUnit Configuration

Each application has its own PHPUnit configuration, stored in the `phpunit.xml.dist` file. You can edit this file to change the defaults or create a `phpunit.xml` file to tweak the configuration for your local machine.

Tip: Store the `phpunit.xml.dist` file in your code repository, and ignore the `phpunit.xml` file.

By default, only the tests stored in "standard" bundles are run by the `phpunit` command (standard being tests in the `src/*/Bundle/Tests` or `src/*/Bundle/*Bundle/Tests` directories) But you can easily add more directories. For instance, the following configuration adds the tests from the installed third-party bundles:

```
<!-- hello/phpunit.xml.dist -->
<testsuites>
    <testsuite name="Project Test Suite">
        <directory>../src/*/*Bundle/Tests</directory>
        <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
    </testsuite>
</testsuites>
```

To include other directories in the code coverage, also edit the `<filter>` section:

```
<filter>
    <whitelist>
        <directory>../src</directory>
        <exclude>
            <directory>../src/*/*Bundle/Resources</directory>
            <directory>../src/*/*Bundle/Tests</directory>
            <directory>../src/Acme/Bundle/*Bundle/Resources</directory>
            <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
        </exclude>
    </whitelist>
</filter>
```

Learn more from the Cookbook

- How to simulate HTTP Authentication in a Functional Test
- How to test the Interaction of several Clients
- How to use the Profiler in a Functional Test

2.1.10 Validation

Validation is a very common task in web applications. Data entered in forms needs to be validated. Data also needs to be validated before it is written into a database or passed to a web service.

Symfony2 ships with a Validator component that makes this task easy and transparent. This component is based on the JSR303 Bean Validation specification. What? A Java specification in PHP? You heard right, but it's not as bad as it sounds. Let's look at how it can be used in PHP.

The Basics of Validation

The best way to understand validation is to see it in action. To start, suppose you've created a plain-old-PHP object that you need to use somewhere in your application:

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    public $name;
}
```

So far, this is just an ordinary class that serves some purpose inside your application. The goal of validation is to tell you whether or not the data of an object is valid. For this to work, you'll configure a list of rules (called *constraints*) that the object must follow in order to be valid. These rules can be specified via a number of different formats (YAML, XML, annotations, or PHP).

For example, to guarantee that the `$name` property is not empty, add the following:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        name:
            - NotBlank: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\BlogBundle\Entity\Author">
        <property name="name">
            <constraint name="NotBlank" />
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

class Author
{
    public $name;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('name', new NotBlank());
    }
}
```

Tip: Protected and private properties can also be validated, as well as "getter" methods (see *validator-constraint-targets*).

Using the `validator` Service

Next, to actually validate an `Author` object, use the `validate` method on the `validator` service (class `Symfony\Component\Validator\Validator`). The job of the `validator` is easy: to read the constraints (i.e. rules) of a class and verify whether or not the data on the object satisfies those constraints. If validation fails, an array of errors is returned. Take this simple example from inside a controller:

```
use Symfony\Component\HttpFoundation\Response;
use Acme\BlogBundle\Entity\Author;
// ...

public function indexAction()
{
    $author = new Author();
    // ... do something to the $author object

    $validator = $this->get('validator');
    $errors = $validator->validate($author);

    if (count($errors) > 0) {
        return new Response(print_r($errors, true));
    } else {
```

```
        return new Response('The author is valid! Yes!');
    }
}
```

If the `$name` property is empty, you will see the following error message:

```
Acme\BlogBundle\Author.name:
    This value should not be blank
```

If you insert a value into the `name` property, the happy success message will appear.

Tip: Most of the time, you won't interact directly with the `validator` service or need to worry about printing out the errors. Most of the time, you'll use validation indirectly when handling submitted form data. For more information, see the *Validation and Forms*.

You could also pass the collection of errors into a template.

```
if (count($errors) > 0) {
    return $this->render('AcmeBlogBundle:Author:validate.html.twig', array(
        'errors' => $errors,
    ));
} else {
    // ...
}
```

Inside the template, you can output the list of errors exactly as needed:

- *Twig*

```
{# src/Acme/BlogBundle/Resources/views/Author/validate.html.twig #}

<h3>The author has the following errors</h3>
<ul>
{% for error in errors %}
    <li>{{ error.message }}</li>
{% endfor %}
</ul>
```

- *PHP*

```
<!-- src/Acme/BlogBundle/Resources/views/Author/validate.html.php -->

<h3>The author has the following errors</h3>
<ul>
<?php foreach ($errors as $error): ?>
    <li><?php echo $error->getMessage() ?></li>
<?php endforeach; ?>
</ul>
```

Note: Each validation error (called a "constraint violation"), is represented by a `Symfony\Component\Validator\ConstraintViolation` object.

Validation and Forms

The `validator` service can be used at any time to validate any object. In reality, however, you'll usually work with the `validator` indirectly when working with forms. Symfony's form library uses the `validator` service internally to validate the underlying object after values have been submitted and bound. The constraint violations on

the object are converted into `FieldError` objects that can easily be displayed with your form. The typical form submission workflow looks like the following from inside a controller:

```
use Acme\BlogBundle\Entity\Author;
use Acme\BlogBundle\Form\AuthorType;
use Symfony\Component\HttpFoundation\Request;
// ...

public function updateAction(Request $request)
{
    $author = new Acme\BlogBundle\Entity\Author();
    $form = $this->createForm(new AuthorType(), $author);

    if ($request->getMethod() == 'POST') {
        $form->bindRequest($request);

        if ($form->isValid()) {
            // the validation passed, do something with the $author object

            return $this->redirect($this->generateUrl('...'));
        }
    }

    return $this->render('BlogBundle:Author:form.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

Note: This example uses an `AuthorType` form class, which is not shown here.

For more information, see the Forms chapter.

Configuration

The Symfony2 validator is enabled by default, but you must explicitly enable annotations if you're using the annotation method to specify your constraints:

- *YAML*

```
# app/config/config.yml
framework:
    validation: { enable_annotations: true }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:validation enable_annotations="true" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array('validation' => array(
    'enable_annotations' => true,
)));
```

Constraints

The `validator` is designed to validate objects against *constraints* (i.e. rules). In order to validate an object, simply map one or more constraints to its class and then pass it to the `validator` service.

Behind the scenes, a constraint is simply a PHP object that makes an assertive statement. In real life, a constraint could be: "The cake must not be burned". In Symfony2, constraints are similar: they are assertions that a condition is true. Given a value, a constraint will tell you whether or not that value adheres to the rules of the constraint.

Supported Constraints

Symfony2 packages a large number of the most commonly-needed constraints:

Basic Constraints

These are the basic constraints: use them to assert very basic things about the value of properties or the return value of methods on your object.

- NotBlank
- Blank
- NotNull
- Null
- True
- False
- Type

String Constraints

- Email
- MinLength
- MaxLength
- Url
- Regex
- Ip

Number Constraints

- Max
- Min

Date Constraints

- Date
- DateTime
- Time

Collection Constraints

- Choice
- Collection
- UniqueEntity
- Language
- Locale
- Country

File Constraints

- File
- Image

Other Constraints

- Callback
- All
- UserPassword
- Valid

You can also create your own custom constraints. This topic is covered in the "How to create a Custom Validation Constraint" article of the cookbook.

Constraint Configuration

Some constraints, like NotBlank, are simple whereas others, like the Choice constraint, have several configuration options available. Suppose that the `Author` class has another property, `gender` that can be set to either "male" or "female":

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: { choices: [male, female], message: Choose a valid gender. }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice(
     *     choices = { "male", "female" },
     *     message = "Choose a valid gender."
     * )
     */
    public $gender;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\BlogBundle\Entity\Author">
        <property name="gender">
            <constraint name="Choice">
                <option name="choices">
                    <value>male</value>
                    <value>female</value>
                </option>
                <option name="message">Choose a valid gender.</option>
            </constraint>
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

class Author
{
    public $gender;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('gender', new Choice(array(
            'choices' => array('male', 'female'),
            'message' => 'Choose a valid gender.',
        )));
    }
}
```

The options of a constraint can always be passed in as an array. Some constraints, however, also allow you to pass the value of one, "*default*", option in place of the array. In the case of the `Choice` constraint, the `choices` options can be specified in this way.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: [male, female]
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice({"male", "female"})
     */
    protected $gender;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\BlogBundle\Entity\Author">
        <property name="gender">
            <constraint name="Choice">
                <value>male</value>
                <value>female</value>
            </constraint>
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Choice;

class Author
{
    protected $gender;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('gender', new Choice(array('male', 'female')));
    }
}
```

This is purely meant to make the configuration of the most common option of a constraint shorter and quicker.

If you're ever unsure of how to specify an option, either check the API documentation for the constraint or play it safe by always passing in an array of options (the first method shown above).

Constraint Targets

Constraints can be applied to a class property (e.g. `name`) or a public getter method (e.g. `getFullName`). The first is the most common and easy to use, but the second allows you to specify more complex validation rules.

Properties

Validating class properties is the most basic validation technique. Symfony2 allows you to validate private, protected or public properties. The next listing shows you how to configure the `$firstName` property of an `Author` class to have at least 3 characters.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        firstName:
            - NotBlank: ~
            - MinLength: 3
```

- *Annotations*

```
// Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     * @Assert\MinLength(3)
     */
    private $firstName;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <property name="firstName">
        <constraint name="NotBlank" />
        <constraint name="MinLength">3</constraint>
    </property>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\MinLength;

class Author
{
    private $firstName;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('firstName', new NotBlank());
```

```
            $metadata->addPropertyConstraint('firstName', new MinLength(3));
        }
    }
```

Getters

Constraints can also be applied to the return value of a method. Symfony2 allows you to add a constraint to any public method whose name starts with "get" or "is". In this guide, both of these types of methods are referred to as "getters".

The benefit of this technique is that it allows you to validate your object dynamically. For example, suppose you want to make sure that a password field doesn't match the first name of the user (for security reasons). You can do this by creating an `isPasswordLegal` method, and then asserting that this method must return `true`:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    getters:
        passwordLegal:
            - "True": { message: "The password cannot match your first name" }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\True(message = "The password cannot match your first name")
     */
    public function isPasswordLegal()
    {
        // return true or false
    }
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <getter property="passwordLegal">
        <constraint name="True">
            <option name="message">The password cannot match your first name</option>
        </constraint>
    </getter>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\True;

class Author
{
    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
```

```
            $metadata->addGetterConstraint('passwordLegal', new True(array(
                'message' => 'The password cannot match your first name',
            )));
        }
    }
```

Now, create the `isPasswordLegal()` method, and include the logic you need:

```
public function isPasswordLegal()
{
    return ($this->firstName != $this->password);
}
```

Note: The keen-eyed among you will have noticed that the prefix of the getter ("get" or "is") is omitted in the mapping. This allows you to move the constraint to a property with the same name later (or vice versa) without changing your validation logic.

Classes

Some constraints apply to the entire class being validated. For example, the Callback constraint is a generic constraint that's applied to the class itself. When that class is validated, methods specified by that constraint are simply executed so that each can provide more custom validation.

Validation Groups

So far, you've been able to add constraints to a class and ask whether or not that class passes all of the defined constraints. In some cases, however, you'll need to validate an object against only *some* of the constraints on that class. To do this, you can organize each constraint into one or more "validation groups", and then apply validation against just one group of constraints.

For example, suppose you have a `User` class, which is used both when a user registers and when a user updates his/her contact information later:

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\User:
        properties:
            email:
                - Email: { groups: [registration] }
            password:
                - NotBlank: { groups: [registration] }
                - MinLength: { limit: 7, groups: [registration] }
            city:
                - MinLength: 2
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/User.php
    namespace Acme\BlogBundle\Entity;

    use Symfony\Component\Security\Core\User\UserInterface;
    use Symfony\Component\Validator\Constraints as Assert;

    class User implements UserInterface
```

```
    {
        /**
        * @Assert\Email(groups={"registration"})
        */
        private $email;

        /**
        * @Assert\NotBlank(groups={"registration"})
        * @Assert\MinLength(limit=7, groups={"registration"})
        */
        private $password;

        /**
        * @Assert\MinLength(2)
        */
        private $city;
    }
```

- *XML*

```
    <!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
    <class name="Acme\BlogBundle\Entity\User">
        <property name="email">
            <constraint name="Email">
                <option name="groups">
                    <value>registration</value>
                </option>
            </constraint>
        </property>
        <property name="password">
            <constraint name="NotBlank">
                <option name="groups">
                    <value>registration</value>
                </option>
            </constraint>
            <constraint name="MinLength">
                <option name="limit">7</option>
                <option name="groups">
                    <value>registration</value>
                </option>
            </constraint>
        </property>
        <property name="city">
            <constraint name="MinLength">7</constraint>
        </property>
    </class>
```

- *PHP*

```
    // src/Acme/BlogBundle/Entity/User.php
    namespace Acme\BlogBundle\Entity;

    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\Email;
    use Symfony\Component\Validator\Constraints\NotBlank;
    use Symfony\Component\Validator\Constraints\MinLength;

    class User
    {
```

```
        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('email', new Email(array(
                'groups' => array('registration')
            )));

            $metadata->addPropertyConstraint('password', new NotBlank(array(
                'groups' => array('registration')
            )));
            $metadata->addPropertyConstraint('password', new MinLength(array(
                'limit'  => 7,
                'groups' => array('registration')
            )));

            $metadata->addPropertyConstraint('city', new MinLength(3));
        }
    }
```

With this configuration, there are two validation groups:

- `Default` - contains the constraints not assigned to any other group;
- `registration` - contains the constraints on the `email` and `password` fields only.

To tell the validator to use a specific group, pass one or more group names as the second argument to the `validate()` method:

```
$errors = $validator->validate($author, array('registration'));
```

Of course, you'll usually work with validation indirectly through the form library. For information on how to use validation groups inside forms, see *Validation Groups*.

Validating Values and Arrays

So far, you've seen how you can validate entire objects. But sometimes, you just want to validate a simple value - like to verify that a string is a valid email address. This is actually pretty easy to do. From inside a controller, it looks like this:

```
// add this to the top of your class
use Symfony\Component\Validator\Constraints\Email;

public function addEmailAction($email)
{
    $emailConstraint = new Email();
    // all constraint "options" can be set this way
    $emailConstraint->message = 'Invalid email address';

    // use the validator to validate the value
    $errorList = $this->get('validator')->validateValue($email, $emailConstraint);

    if (count($errorList) == 0) {
        // this IS a valid email address, do something
    } else {
        // this is *not* a valid email address
        $errorMessage = $errorList[0]->getMessage()

        // do something with the error
    }
```

```
    // ...
}
```

By calling `validateValue` on the validator, you can pass in a raw value and the constraint object that you want to validate that value against. A full list of the available constraints - as well as the full class name for each constraint - is available in the constraints reference section .

The `validateValue` method returns a `Symfony\Component\Validator\ConstraintViolationList` object, which acts just like an array of errors. Each error in the collection is a `Symfony\Component\Validator\ConstraintViolation` object, which holds the error message on its *getMessage* method.

Final Thoughts

The Symfony2 `validator` is a powerful tool that can be leveraged to guarantee that the data of any object is "valid". The power behind validation lies in "constraints", which are rules that you can apply to properties or getter methods of your object. And while you'll most commonly use the validation framework indirectly when using forms, remember that it can be used anywhere to validate any object.

Learn more from the Cookbook

- How to create a Custom Validation Constraint

2.1.11 Forms

Dealing with HTML forms is one of the most common - and challenging - tasks for a web developer. Symfony2 integrates a Form component that makes dealing with forms easy. In this chapter, you'll build a complex form from the ground-up, learning the most important features of the form library along the way.

Note: The Symfony form component is a standalone library that can be used outside of Symfony2 projects. For more information, see the Symfony2 Form Component on Github.

Creating a Simple Form

Suppose you're building a simple todo list application that will need to display "tasks". Because your users will need to edit and create tasks, you're going to need to build a form. But before you begin, first focus on the generic `Task` class that represents and stores the data for a single task:

```
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

class Task
{
    protected $task;

    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }
    public function setTask($task)
```

```
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }
    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

Note: If you're coding along with this example, create the `AcmeTaskBundle` first by running the following command (and accepting all of the default options):

```
php app/console generate:bundle --namespace=Acme/TaskBundle
```

This class is a "plain-old-PHP-object" because, so far, it has nothing to do with Symfony or any other library. It's quite simply a normal PHP object that directly solves a problem inside *your* application (i.e. the need to represent a task in your application). Of course, by the end of this chapter, you'll be able to submit data to a `Task` instance (via an HTML form), validate its data, and persist it to the database.

Building the Form

Now that you've created a `Task` class, the next step is to create and render the actual HTML form. In Symfony2, this is done by building a form object and then rendering it in a template. For now, this can all be done from inside a controller:

```
// src/Acme/TaskBundle/Controller/DefaultController.php
namespace Acme\TaskBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Acme\TaskBundle\Entity\Task;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function newAction(Request $request)
    {
        // create a task and give it some dummy data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', 'text')
            ->add('dueDate', 'date')
            ->getForm();

        return $this->render('AcmeTaskBundle:Default:new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

Tip: This example shows you how to build your form directly in the controller. Later, in the "*Creating Form Classes*" section, you'll learn how to build your form in a standalone class, which is recommended as your form becomes reusable.

Creating a form requires relatively little code because Symfony2 form objects are built with a "form builder". The form builder's purpose is to allow you to write simple form "recipes", and have it do all the heavy-lifting of actually building the form.

In this example, you've added two fields to your form - `task` and `dueDate` - corresponding to the `task` and `dueDate` properties of the `Task` class. You've also assigned each a "type" (e.g. `text`, `date`), which, among other things, determines which HTML form tag(s) is rendered for that field.

Symfony2 comes with many built-in types that will be discussed shortly (see *Built-in Field Types*).

Rendering the Form

Now that the form has been created, the next step is to render it. This is done by passing a special form "view" object to your template (notice the `$form->createView()` in the controller above) and using a set of form helper functions:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

<form action="{{ path('task_new') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Default/new.html.php -->

<form action="<?php echo $view['router']->generate('task_new') ?>" method="post" <?php echo $vie
    <?php echo $view['form']->widget($form) ?>

    <input type="submit" />
</form>
```

Note: This example assumes that you've created a route called `task_new` that points to the `AcmeTaskBundle:Default:new` controller that was created earlier.

That's it! By printing `form_widget(form)`, each field in the form is rendered, along with a label and error message (if there is one). As easy as this is, it's not very flexible (yet). Usually, you'll want to render each form

field individually so you can control how the form looks. You'll learn how to do that in the "*Rendering a Form in a Template*" section.

Before moving on, notice how the rendered `task` input field has the value of the `task` property from the `$task` object (i.e. "Write a blog post"). This is the first job of a form: to take data from an object and translate it into a format that's suitable for being rendered in an HTML form.

Tip: The form system is smart enough to access the value of the protected `task` property via the `getTask()` and `setTask()` methods on the `Task` class. Unless a property is public, it *must* have a "getter" and "setter" method so that the form component can get and put data onto the property. For a Boolean property, you can use an "isser" method (e.g. `isPublished()`) instead of a getter (e.g. `getPublished()`).

Handling Form Submissions

The second job of a form is to translate user-submitted data back to the properties of an object. To make this happen, the submitted data from the user must be bound to the form. Add the following functionality to your controller:

```
// ...

public function newAction(Request $request)
{
    // just setup a fresh $task object (remove the dummy data)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', 'text')
        ->add('dueDate', 'date')
        ->getForm();

    if ($request->getMethod() == 'POST') {
        $form->bindRequest($request);

        if ($form->isValid()) {
            // perform some action, such as saving the task to the database

            return $this->redirect($this->generateUrl('task_success'));
        }
    }

    // ...
}
```

Now, when submitting the form, the controller binds the submitted data to the form, which translates that data back to the `task` and `dueDate` properties of the `$task` object. This all happens via the `bindRequest()` method.

Note: As soon as `bindRequest()` is called, the submitted data is transferred to the underlying object immediately. This happens regardless of whether or not the underlying data is actually valid.

This controller follows a common pattern for handling forms, and has three possible paths:

1. When initially loading the page in a browser, the request method is `GET` and the form is simply created and rendered;
2. When the user submits the form (i.e. the method is `POST`) with invalid data (validation is covered in the next section), the form is bound and then rendered, this time displaying all validation errors;

3. When the user submits the form with valid data, the form is bound and you have the opportunity to perform some actions using the `$task` object (e.g. persisting it to the database) before redirecting the user to some other page (e.g. a "thank you" or "success" page).

Note: Redirecting a user after a successful form submission prevents the user from being able to hit "refresh" and re-post the data.

Form Validation

In the previous section, you learned how a form can be submitted with valid or invalid data. In Symfony2, validation is applied to the underlying object (e.g. `Task`). In other words, the question isn't whether the "form" is valid, but whether or not the `$task` object is valid after the form has applied the submitted data to it. Calling `$form->isValid()` is a shortcut that asks the `$task` object whether or not it has valid data.

Validation is done by adding a set of rules (called constraints) to a class. To see this in action, add validation constraints so that the `task` field cannot be empty and the `dueDate` field cannot be empty and must be a valid DateTime object.

- *YAML*

```
# Acme/TaskBundle/Resources/config/validation.yml
Acme\TaskBundle\Entity\Task:
    properties:
        task:
            - NotBlank: ~
        dueDate:
            - NotBlank: ~
            - Type: \DateTime
```

- *Annotations*

```
// Acme/TaskBundle/Entity/Task.php
use Symfony\Component\Validator\Constraints as Assert;

class Task
{
    /**
     * @Assert\NotBlank()
     */
    public $task;

    /**
     * @Assert\NotBlank()
     * @Assert\Type("\DateTime")
     */
    protected $dueDate;
}
```

- *XML*

```
<!-- Acme/TaskBundle/Resources/config/validation.xml -->
<class name="Acme\TaskBundle\Entity\Task">
    <property name="task">
        <constraint name="NotBlank" />
    </property>
    <property name="dueDate">
        <constraint name="NotBlank" />
        <constraint name="Type">
            <value>\DateTime</value>
```

```
            </constraint>
        </property>
    </class>
```

- *PHP*

```
    // Acme/TaskBundle/Entity/Task.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\NotBlank;
    use Symfony\Component\Validator\Constraints\Type;

    class Task
    {
        // ...

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('task', new NotBlank());

            $metadata->addPropertyConstraint('dueDate', new NotBlank());
            $metadata->addPropertyConstraint('dueDate', new Type('\DateTime'));
        }
    }
```

That's it! If you re-submit the form with invalid data, you'll see the corresponding errors printed out with the form.

HTML5 Validation

As of HTML5, many browsers can natively enforce certain validation constraints on the client side. The most common validation is activated by rendering a `required` attribute on fields that are required. For browsers that support HTML5, this will result in a native browser message being displayed if the user tries to submit the form with that field blank.
Generated forms take full advantage of this new feature by adding sensible HTML attributes that trigger the validation. The client-side validation, however, can be disabled by adding the `novalidate` attribute to the `form` tag or `formnovalidate` to the submit tag. This is especially useful when you want to test your server-side validation constraints, but are being prevented by your browser from, for example, submitting blank fields.

Validation is a very powerful feature of Symfony2 and has its own dedicated chapter.

Validation Groups

Tip: If you're not using *validation groups*, then you can skip this section.

If your object takes advantage of *validation groups*, you'll need to specify which validation group(s) your form should use:

```
$form = $this->createFormBuilder($users, array(
    'validation_groups' => array('registration'),
))->add(...)
;
```

If you're creating *form classes* (a good practice), then you'll need to add the following to the `getDefaultOptions()` method:

```
public function getDefaultOptions(array $options)
{
    return array(
        'validation_groups' => array('registration')
    );
}
```

In both of these cases, *only* the `registration` validation group will be used to validate the underlying object.

Groups based on Submitted Data

New in version 2.1: The ability to specify a callback or Closure in `validation_groups` is new to version 2.1

If you need some advanced logic to determine the validation groups (e.g. based on submitted data), you can set the `validation_groups` option to an array callback, or a `Closure`:

```
public function getDefaultOptions(array $options)
{
    return array(
        'validation_groups' => array('Acme\\AcmeBundle\\Entity\\Client', 'determineValidationGroups'
    );
}
```

This will call the static method `determineValidationGroups()` on the `Client` class after the form is bound, but before validation is executed. The Form object is passed as an argument to that method (see next example). You can also define whole logic inline by using a Closure:

```
public function getDefaultOptions(array $options)
{
    return array(
        'validation_groups' => function(FormInterface $form) {
            $data = $form->getData();
            if (Entity\Client::TYPE_PERSON == $data->getType()) {
                return array('person')
            } else {
                return array('company');
            }
        },
    );
}
```

Built-in Field Types

Symfony comes standard with a large group of field types that cover all of the common form fields and data types you'll encounter:

Text Fields

- text
- textarea
- email
- integer

- money
- number
- password
- percent
- search
- url

Choice Fields

- choice
- entity
- country
- language
- locale
- timezone

Date and Time Fields

- date
- datetime
- time
- birthday

Other Fields

- checkbox
- file
- radio

Field Groups

- collection
- repeated

Hidden Fields

- hidden
- csrf

Base Fields

- field
- form

You can also create your own custom field types. This topic is covered in the "How to Create a Custom Form Field Type" article of the cookbook.

Field Type Options

Each field type has a number of options that can be used to configure it. For example, the `dueDate` field is currently being rendered as 3 select boxes. However, the date field can be configured to be rendered as a single text box (where the user would enter the date as a string in the box):

```
->add('dueDate', 'date', array('widget' => 'single_text'))
```

Each field type has a number of different options that can be passed to it. Many of these are specific to the field type and details can be found in the documentation for each type.

The `required` option

The most common option is the `required` option, which can be applied to any field. By default, the `required` option is set to `true`, meaning that HTML5-ready browsers will apply client-side validation if the field is left blank. If you don't want this behavior, either set the `required` option on your field to `false` or *disable HTML5 validation*.
Also note that setting the `required` option to `true` will **not** result in server-side validation to be applied. In other words, if a user submits a blank value for the field (either with an old browser or web service, for example), it will be accepted as a valid value unless you use Symfony's `NotBlank` or `NotNull` validation constraint.
In other words, the `required` option is "nice", but true server-side validation should *always* be used.

The `label` option

The label for the form field can be set using the `label` option, which can be applied to any field:

```
->add('dueDate', 'date', array(
    'widget' => 'single_text',
    'label'  => 'Due Date',
))
```

The label for a field can also be set in the template rendering the form, see below.

Field Type Guessing

Now that you've added validation metadata to the `Task` class, Symfony already knows a bit about your fields. If you allow it, Symfony can "guess" the type of your field and set it up for you. In this example, Symfony can guess from the validation rules that both the `task` field is a normal `text` field and the `dueDate` field is a `date` field:

```
public function newAction()
{
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task')
        ->add('dueDate', null, array('widget' => 'single_text'))
        ->getForm();
}
```

The "guessing" is activated when you omit the second argument to the `add()` method (or if you pass `null` to it). If you pass an options array as the third argument (done for `dueDate` above), these options are applied to the guessed field.

Caution: If your form uses a specific validation group, the field type guesser will still consider *all* validation constraints when guessing your field types (including constraints that are not part of the validation group(s) being used).

Field Type Options Guessing

In addition to guessing the "type" for a field, Symfony can also try to guess the correct values of a number of field options.

Tip: When these options are set, the field will be rendered with special HTML attributes that provide for HTML5 client-side validation. However, it doesn't generate the equivalent server-side constraints (e.g. `Assert\MaxLength`). And though you'll need to manually add your server-side validation, these field type options can then be guessed from that information.

- `required`: The `required` option can be guessed based off of the validation rules (i.e. is the field `NotBlank` or `NotNull`) or the Doctrine metadata (i.e. is the field `nullable`). This is very useful, as your client-side validation will automatically match your validation rules.
- `min_length`: If the field is some sort of text field, then the `min_length` option can be guessed from the validation constrains (if `MinLength` or `Min` is used) or from the Doctrine metadata (via the field's length).
- `max_length`: Similar to `min_length`, the maximum length can also be guessed.

Note: These field options are *only* guessed if you're using Symfony to guess the field type (i.e. omit or pass `null` as the second argument to `add()`).

If you'd like to change one of the guessed values, you can override it by passing the option in the options field array:

```
->add('task', null, array('min_length' => 4))
```

Rendering a Form in a Template

So far, you've seen how an entire form can be rendered with just one line of code. Of course, you'll usually need much more flexibility when rendering:

- *Twig*

```
    {# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

    <form action="{{ path('task_new') }}" method="post" {{ form_enctype(form) }}>
```

```
        {{ form_errors(form) }}

        {{ form_row(form.task) }}
        {{ form_row(form.dueDate) }}

        {{ form_rest(form) }}

        <input type="submit" />
    </form>
```

- *PHP*

```
    <!-- // src/Acme/TaskBundle/Resources/views/Default/newAction.html.php -->

    <form action="<?php echo $view['router']->generate('task_new') ?>" method="post" <?php echo $vie
        <?php echo $view['form']->errors($form) ?>

        <?php echo $view['form']->row($form['task']) ?>
        <?php echo $view['form']->row($form['dueDate']) ?>

        <?php echo $view['form']->rest($form) ?>

        <input type="submit" />
    </form>
```

Let's take a look at each part:

- `form_enctype(form)` - If at least one field is a file upload field, this renders the obligatory `enctype="multipart/form-data"`;
- `form_errors(form)` - Renders any errors global to the whole form (field-specific errors are displayed next to each field);
- `form_row(form.dueDate)` - Renders the label, any errors, and the HTML form widget for the given field (e.g. `dueDate`) inside, by default, a `div` element;
- `form_rest(form)` - Renders any fields that have not yet been rendered. It's usually a good idea to place a call to this helper at the bottom of each form (in case you forgot to output a field or don't want to bother manually rendering hidden fields). This helper is also useful for taking advantage of the automatic *CSRF Protection*.

The majority of the work is done by the `form_row` helper, which renders the label, errors and HTML form widget of each field inside a `div` tag by default. In the *Form Theming* section, you'll learn how the `form_row` output can be customized on many different levels.

Tip: You can access the current data of your form via `form.vars.value`:

- *Twig*

```
    {{ form.vars.value.task }}
```

- *PHP*

```
    <?php echo $view['form']->get('value')->getTask() ?>
```

Rendering each Field by Hand

The `form_row` helper is great because you can very quickly render each field of your form (and the markup used for the "row" can be customized as well). But since life isn't always so simple, you can also render each field entirely by

hand. The end-product of the following is the same as when you used the `form_row` helper:

- *Twig*

```
{{ form_errors(form) }}

<div>
    {{ form_label(form.task) }}
    {{ form_errors(form.task) }}
    {{ form_widget(form.task) }}
</div>

<div>
    {{ form_label(form.dueDate) }}
    {{ form_errors(form.dueDate) }}
    {{ form_widget(form.dueDate) }}
</div>

{{ form_rest(form) }}
```

- *PHP*

```
<?php echo $view['form']->errors($form) ?>

<div>
    <?php echo $view['form']->label($form['task']) ?>
    <?php echo $view['form']->errors($form['task']) ?>
    <?php echo $view['form']->widget($form['task']) ?>
</div>

<div>
    <?php echo $view['form']->label($form['dueDate']) ?>
    <?php echo $view['form']->errors($form['dueDate']) ?>
    <?php echo $view['form']->widget($form['dueDate']) ?>
</div>

<?php echo $view['form']->rest($form) ?>
```

If the auto-generated label for a field isn't quite right, you can explicitly specify it:

- *Twig*

```
{{ form_label(form.task, 'Task Description') }}
```

- *PHP*

```
<?php echo $view['form']->label($form['task'], 'Task Description') ?>
```

Some field types have additional rendering options that can be passed to the widget. These options are documented with each type, but one common options is `attr`, which allows you to modify attributes on the form element. The following would add the `task_field` class to the rendered input text field:

- *Twig*

```
{{ form_widget(form.task, { 'attr': {'class': 'task_field'} }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['task'], array(
    'attr' => array('class' => 'task_field'),
)) ?>
```

If you need to render form fields "by hand" then you can access individual values for fields such as the `id`, `name` and `label`. For example to get the `id`:

- *Twig*

```
{{ form.task.vars.id }}
```

- *PHP*

```
<?php echo $form['task']->get('id') ?>
```

To get the value used for the form field's name attribute you need to use the `full_name` value:

- *Twig*

```
{{ form.task.vars.full_name }}
```

- *PHP*

```
<?php echo $form['task']->get('full_name') ?>
```

Twig Template Function Reference

If you're using Twig, a full reference of the form rendering functions is available in the reference manual. Read this to know everything about the helpers available and the options that can be used with each.

Creating Form Classes

As you've seen, a form can be created and used directly in a controller. However, a better practice is to build the form in a separate, standalone PHP class, which can then be reused anywhere in your application. Create a new class that will house the logic for building the task form:

```
// src/Acme/TaskBundle/Form/Type/TaskType.php

namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('task');
        $builder->add('dueDate', null, array('widget' => 'single_text'));
    }

    public function getName()
    {
        return 'task';
    }
}
```

This new class contains all the directions needed to create the task form (note that the `getName()` method should return a unique identifier for this form "type"). It can be used to quickly build a form object in the controller:

```
// src/Acme/TaskBundle/Controller/DefaultController.php

// add this new use statement at the top of the class
use Acme\TaskBundle\Form\Type\TaskType;

public function newAction()
{
    $task = // ...
    $form = $this->createForm(new TaskType(), $task);

    // ...
}
```

Placing the form logic into its own class means that the form can be easily reused elsewhere in your project. This is the best way to create forms, but the choice is ultimately up to you.

Setting the `data_class`

Every form needs to know the name of the class that holds the underlying data (e.g. `Acme\TaskBundle\Entity\Task`). Usually, this is just guessed based off of the object passed to the second argument to `createForm` (i.e. `$task`). Later, when you begin embedding forms, this will no longer be sufficient. So, while not always necessary, it's generally a good idea to explicitly specify the `data_class` option by adding the following to your form type class:

```
public function getDefaultOptions(array $options)
{
    return array(
        'data_class' => 'Acme\TaskBundle\Entity\Task',
    );
}
```

Tip: When mapping forms to objects, all fields are mapped. Any fields on the form that do not exist on the mapped object will cause an exception to be thrown.

In cases where you need extra fields in the form (for example: a "do you agree with these terms" checkbox) that will not be mapped to the underlying object, you need to set the property_path option to `false`:

```
public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('task');
    $builder->add('dueDate', null, array('property_path' => false));
}
```

Additionally, if there are any fields on the form that aren't included in the submitted data, those fields will be explicitly set to `null`.

Forms and Doctrine

The goal of a form is to translate data from an object (e.g. `Task`) to an HTML form and then translate user-submitted data back to the original object. As such, the topic of persisting the `Task` object to the database is entirely unrelated to the topic of forms. But, if you've configured the `Task` class to be persisted via Doctrine (i.e. you've added *mapping metadata* for it), then persisting it after a form submission can be done when the form is valid:

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getEntityManager();
```

```
    $em->persist($task);
    $em->flush();

    return $this->redirect($this->generateUrl('task_success'));
}
```

If, for some reason, you don't have access to your original `$task` object, you can fetch it from the form:

```
$task = $form->getData();
```

For more information, see the Doctrine ORM chapter.

The key thing to understand is that when the form is bound, the submitted data is transferred to the underlying object immediately. If you want to persist that data, you simply need to persist the object itself (which already contains the submitted data).

Embedded Forms

Often, you'll want to build a form that will include fields from many different objects. For example, a registration form may contain data belonging to a `User` object as well as many `Address` objects. Fortunately, this is easy and natural with the form component.

Embedding a Single Object

Suppose that each `Task` belongs to a simple `Category` object. Start, of course, by creating the `Category` object:

```
// src/Acme/TaskBundle/Entity/Category.php
namespace Acme\TaskBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Category
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

Next, add a new `category` property to the `Task` class:

```
// ...

class Task
{
    // ...

    /**
     * @Assert\Type(type="Acme\TaskBundle\Entity\Category")
     */
    protected $category;

    // ...

    public function getCategory()
    {
        return $this->category;
```

```
    }

    public function setCategory(Category $category = null)
    {
        $this->category = $category;
    }
}
```

Now that your application has been updated to reflect the new requirements, create a form class so that a `Category` object can be modified by the user:

```
// src/Acme/TaskBundle/Form/Type/CategoryType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class CategoryType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('name');
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class' => 'Acme\TaskBundle\Entity\Category',
        );
    }

    public function getName()
    {
        return 'category';
    }
}
```

The end goal is to allow the `Category` of a `Task` to be modified right inside the task form itself. To accomplish this, add a `category` field to the `TaskType` object whose type is an instance of the new `CategoryType` class:

```
public function buildForm(FormBuilder $builder, array $options)
{
    // ...

    $builder->add('category', new CategoryType());
}
```

The fields from `CategoryType` can now be rendered alongside those from the `TaskType` class. Render the `Category` fields in the same way as the original `Task` fields:

- *Twig*

```
{# ... #}

<h3>Category</h3>
<div class="category">
    {{ form_row(form.category.name) }}
</div>
```

```
    {{ form_rest(form) }}
    {# ... #}
```

- *PHP*

```
    <!-- ... -->

    <h3>Category</h3>
    <div class="category">
        <?php echo $view['form']->row($form['category']['name']) ?>
    </div>

    <?php echo $view['form']->rest($form) ?>
    <!-- ... -->
```

When the user submits the form, the submitted data for the `Category` fields are used to construct an instance of `Category`, which is then set on the `category` field of the `Task` instance.

The `Category` instance is accessible naturally via `$task->getCategory()` and can be persisted to the database or used however you need.

Embedding a Collection of Forms

You can also embed a collection of forms into one form (imagine a `Category` form with many `Product` sub-forms). This is done by using the `collection` field type.

For more information see the "How to Embed a Collection of Forms" cookbook entry and the collection field type reference.

Form Theming

Every part of how a form is rendered can be customized. You're free to change how each form "row" renders, change the markup used to render errors, or even customize how a `textarea` tag should be rendered. Nothing is off-limits, and different customizations can be used in different places.

Symfony uses templates to render each and every part of a form, such as `label` tags, `input` tags, error messages and everything else.

In Twig, each form "fragment" is represented by a Twig block. To customize any part of how a form renders, you just need to override the appropriate block.

In PHP, each form "fragment" is rendered via an individual template file. To customize any part of how a form renders, you just need to override the existing template by creating a new one.

To understand how this works, let's customize the `form_row` fragment and add a class attribute to the `div` element that surrounds each row. To do this, create a new template file that will store the new markup:

- *Twig*

```
    {# src/Acme/TaskBundle/Resources/views/Form/fields.html.twig #}

    {% block field_row %}
    {% spaceless %}
        <div class="form_row">
            {{ form_label(form) }}
            {{ form_errors(form) }}
            {{ form_widget(form) }}
        </div>
```

```
{% endspaceless %}
{% endblock field_row %}
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Form/field_row.html.php -->

<div class="form_row">
    <?php echo $view['form']->label($form, $label) ?>
    <?php echo $view['form']->errors($form) ?>
    <?php echo $view['form']->widget($form, $parameters) ?>
</div>
```

The `field_row` form fragment is used when rendering most fields via the `form_row` function. To tell the form component to use your new `field_row` fragment defined above, add the following to the top of the template that renders the form:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

{% form_theme form 'AcmeTaskBundle:Form:fields.html.twig' %}

<form ...>
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Default/new.html.php -->

<?php $view['form']->setTheme($form, array('AcmeTaskBundle:Form')) ?>

<form ...>
```

The `form_theme` tag (in Twig) "imports" the fragments defined in the given template and uses them when rendering the form. In other words, when the `form_row` function is called later in this template, it will use the `field_row` block from your custom theme (instead of the default `field_row` block that ships with Symfony).

To customize any portion of a form, you just need to override the appropriate fragment. Knowing exactly which block or file to override is the subject of the next section.

For a more extensive discussion, see How to customize Form Rendering.

Form Fragment Naming

In Symfony, every part of a form that is rendered - HTML form elements, errors, labels, etc - is defined in a base theme, which is a collection of blocks in Twig and a collection of template files in PHP.

In Twig, every block needed is defined in a single template file (form_div_layout.html.twig) that lives inside the Twig Bridge. Inside this file, you can see every block needed to render a form and every default field type.

In PHP, the fragments are individual template files. By default they are located in the *Resources/views/Form* directory of the framework bundle (view on GitHub).

Each fragment name follows the same basic pattern and is broken up into two pieces, separated by a single underscore character (_). A few examples are:

- `field_row` - used by `form_row` to render most fields;
- `textarea_widget` - used by `form_widget` to render a `textarea` field type;
- `field_errors` - used by `form_errors` to render errors for a field;

Each fragment follows the same basic pattern: `type_part`. The `type` portion corresponds to the field *type* being rendered (e.g. `textarea`, `checkbox`, `date`, etc) whereas the `part` portion corresponds to *what* is being rendered (e.g. `label`, `widget`, `errors`, etc). By default, there are 4 possible *parts* of a form that can be rendered:

`label`	(e.g. `field_label`)	renders the field's label
`widget`	(e.g. `field_widget`)	renders the field's HTML representation
`errors`	(e.g. `field_errors`)	renders the field's errors
`row`	(e.g. `field_row`)	renders the field's entire row (label, widget & errors)

Note: There are actually 3 other *parts* - `rows`, `rest`, and `enctype` - but you should rarely if ever need to worry about overriding them.

By knowing the field type (e.g. `textarea`) and which part you want to customize (e.g. `widget`), you can construct the fragment name that needs to be overridden (e.g. `textarea_widget`).

Template Fragment Inheritance

In some cases, the fragment you want to customize will appear to be missing. For example, there is no `textarea_errors` fragment in the default themes provided with Symfony. So how are the errors for a textarea field rendered?

The answer is: via the `field_errors` fragment. When Symfony renders the errors for a textarea type, it looks first for a `textarea_errors` fragment before falling back to the `field_errors` fragment. Each field type has a *parent* type (the parent type of `textarea` is `field`), and Symfony uses the fragment for the parent type if the base fragment doesn't exist.

So, to override the errors for *only* `textarea` fields, copy the `field_errors` fragment, rename it to `textarea_errors` and customize it. To override the default error rendering for *all* fields, copy and customize the `field_errors` fragment directly.

Tip: The "parent" type of each field type is available in the form type reference for each field type.

Global Form Theming

In the above example, you used the `form_theme` helper (in Twig) to "import" the custom form fragments into *just* that form. You can also tell Symfony to import form customizations across your entire project.

Twig To automatically include the customized blocks from the `fields.html.twig` template created earlier in *all* templates, modify your application configuration file:

- *YAML*

```
# app/config/config.yml

twig:
    form:
        resources:
            - 'AcmeTaskBundle:Form:fields.html.twig'
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->

<twig:config ...>
```

```
            <twig:form>
                <resource>AcmeTaskBundle:Form:fields.html.twig</resource>
            </twig:form>
            <!-- ... -->
    </twig:config>
```

- *PHP*

```
    // app/config/config.php

    $container->loadFromExtension('twig', array(
        'form' => array('resources' => array(
            'AcmeTaskBundle:Form:fields.html.twig',
         ))
        // ...
    ));
```

Any blocks inside the `fields.html.twig` template are now used globally to define form output.

Customizing Form Output all in a Single File with Twig

In Twig, you can also customize a form block right inside the template where that customization is needed:

```
{% extends '::base.html.twig' %}

{# import "_self" as the form theme #}
{% form_theme form _self %}

{# make the form fragment customization #}
{% block field_row %}
    {# custom field row output #}
{% endblock field_row %}

{% block content %}
    {# ... #}

    {{ form_row(form.task) }}
{% endblock %}
```

The `{% form_theme form _self %}` tag allows form blocks to be customized directly inside the template that will use those customizations. Use this method to quickly make form output customizations that will only ever be needed in a single template.

PHP To automatically include the customized templates from the `Acme/TaskBundle/Resources/views/Form` directory created earlier in *all* templates, modify your application configuration file:

- *YAML*

```
    # app/config/config.yml

    framework:
        templating:
            form:
                resources:
                    - 'AcmeTaskBundle:Form'
    # ...
```

- *XML*

```
    <!-- app/config/config.xml -->

    <framework:config ...>
        <framework:templating>
            <framework:form>
                <resource>AcmeTaskBundle:Form</resource>
            </framework:form>
        </framework:templating>
        <!-- ... -->
    </framework:config>
```

- *PHP*

```
    // app/config/config.php

    $container->loadFromExtension('framework', array(
        'templating' => array('form' =>
            array('resources' => array(
                'AcmeTaskBundle:Form',
         )))
        // ...
    ));
```

Any fragments inside the `Acme/TaskBundle/Resources/views/Form` directory are now used globally to define form output.

CSRF Protection

CSRF - or Cross-site request forgery - is a method by which a malicious user attempts to make your legitimate users unknowingly submit data that they don't intend to submit. Fortunately, CSRF attacks can be prevented by using a CSRF token inside your forms.

The good news is that, by default, Symfony embeds and validates CSRF tokens automatically for you. This means that you can take advantage of the CSRF protection without doing anything. In fact, every form in this chapter has taken advantage of the CSRF protection!

CSRF protection works by adding a hidden field to your form - called `_token` by default - that contains a value that only you and your user knows. This ensures that the user - not some other entity - is submitting the given data. Symfony automatically validates the presence and accuracy of this token.

The `_token` field is a hidden field and will be automatically rendered if you include the `form_rest()` function in your template, which ensures that all un-rendered fields are output.

The CSRF token can be customized on a form-by-form basis. For example:

```
class TaskType extends AbstractType
{
    // ...

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class'      => 'Acme\TaskBundle\Entity\Task',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // a unique key to help generate the secret token
            'intention'       => 'task_item',
        );
```

```
    }

    // ...
}
```

To disable CSRF protection, set the `csrf_protection` option to false. Customizations can also be made globally in your project. For more information, see the *form configuration reference* section.

Note: The `intention` option is optional but greatly enhances the security of the generated token by making it different for each form.

Using a Form without a Class

In most cases, a form is tied to an object, and the fields of the form get and store their data on the properties of that object. This is exactly what you've seen so far in this chapter with the *Task* class.

But sometimes, you may just want to use a form without a class, and get back an array of the submitted data. This is actually really easy:

```
// make sure you've imported the Request namespace above the class
use Symfony\Component\HttpFoundation\Request
// ...

public function contactAction(Request $request)
{
    $defaultData = array('message' => 'Type your message here');
    $form = $this->createFormBuilder($defaultData)
        ->add('name', 'text')
        ->add('email', 'email')
        ->add('message', 'textarea')
        ->getForm();

        if ($request->getMethod() == 'POST') {
            $form->bindRequest($request);

            // data is an array with "name", "email", and "message" keys
            $data = $form->getData();
        }

    // ... render the form
}
```

By default, a form actually assumes that you want to work with arrays of data, instead of an object. There are exactly two ways that you can change this behavior and tie the form to an object instead:

1. Pass an object when creating the form (as the first argument to `createFormBuilder` or the second argument to `createForm`);
2. Declare the `data_class` option on your form.

If you *don't* do either of these, then the form will return the data as an array. In this example, since `$defaultData` is not an object (and no `data_class` option is set), `$form->getData()` ultimately returns an array.

Tip: You can also access POST values (in this case "name") directly through the request object, like so:

```
$this->get('request')->request->get('name');
```

Be advised, however, that in most cases using the getData() method is a better choice, since it returns the data (usually an object) after it's been transformed by the form framework.

Adding Validation

The only missing piece is validation. Usually, when you call `$form->isValid()`, the object is validated by reading the constraints that you applied to that class. But without a class, how can you add constraints to the data of your form?

The answer is to setup the constraints yourself, and pass them into your form. The overall approach is covered a bit more in the *validation chapter*, but here's a short example:

```
// import the namespaces above your controller class
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\MinLength;
use Symfony\Component\Validator\Constraints\Collection;

$collectionConstraint = new Collection(array(
    'name' => new MinLength(5),
    'email' => new Email(array('message' => 'Invalid email address')),
));

// create a form, no default values, pass in the constraint option
$form = $this->createFormBuilder(null, array(
    'validation_constraint' => $collectionConstraint,
))->add('email', 'email')
    // ...
;
```

Now, when you call *$form->isValid()*, the constraints setup here are run against your form's data. If you're using a form class, override the `getDefaultOptions` method to specify the option:

```
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\MinLength;
use Symfony\Component\Validator\Constraints\Collection;

class ContactType extends AbstractType
{
    // ...

    public function getDefaultOptions(array $options)
    {
        $collectionConstraint = new Collection(array(
            'name' => new MinLength(5),
            'email' => new Email(array('message' => 'Invalid email address')),
        ));

        return array('validation_constraint' => $collectionConstraint);
    }
}
```

Now, you have the flexibility to create forms - with validation - that return an array of data, instead of an object. In most cases, it's better - and certainly more robust - to bind your form to an object. But for simple forms, this is a great approach.

Final Thoughts

You now know all of the building blocks necessary to build complex and functional forms for your application. When building forms, keep in mind that the first goal of a form is to translate data from an object (`Task`) to an HTML form so that the user can modify that data. The second goal of a form is to take the data submitted by the user and to re-apply it to the object.

There's still much more to learn about the powerful world of forms, such as how to handle file uploads with Doctrine or how to create a form where a dynamic number of sub-forms can be added (e.g. a todo list where you can keep adding more fields via Javascript before submitting). See the cookbook for these topics. Also, be sure to lean on the field type reference documentation, which includes examples of how to use each field type and its options.

Learn more from the Cookbook

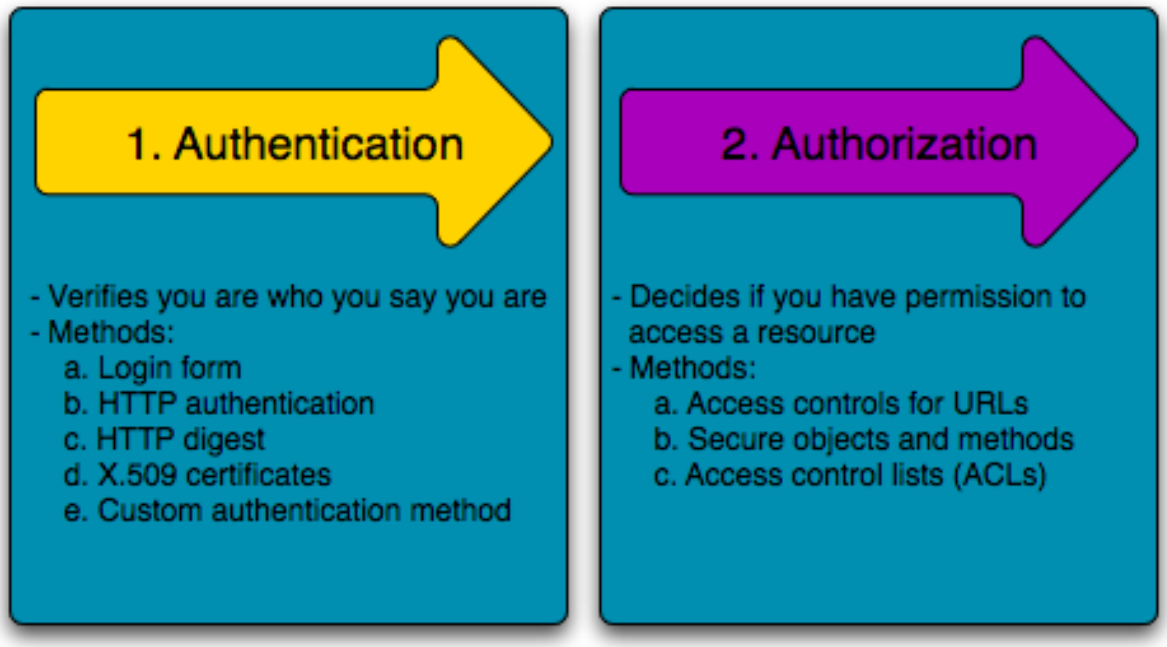
- How to handle File Uploads with Doctrine
- File Field Reference
- Creating Custom Field Types
- How to customize Form Rendering
- How to Dynamically Generate Forms Using Form Events
- Using Data Transformers

2.1.12 Security

Security is a two-step process whose goal is to prevent a user from accessing a resource that he/she should not have access to.

In the first step of the process, the security system identifies who the user is by requiring the user to submit some sort of identification. This is called **authentication**, and it means that the system is trying to find out who you are.

Once the system knows who you are, the next step is to determine if you should have access to a given resource. This part of the process is called **authorization**, and it means that the system is checking to see if you have privileges to perform a certain action.



Since the best way to learn is to see an example, let's dive right in.

Note: Symfony's security component is available as a standalone PHP library for use inside any PHP project.

Basic Example: HTTP Authentication

The security component can be configured via your application configuration. In fact, most standard security setups are just a matter of using the right configuration. The following configuration tells Symfony to secure any URL matching `/admin/*` and to ask the user for credentials using basic HTTP authentication (i.e. the old-school username/password box):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            http_basic:
                realm: "Secured Demo Area"

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }

    providers:
        in_memory:
            memory:
                users:
                    ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                    admin: { password: kitten, roles: 'ROLE_ADMIN' }
```

```
        encoders:
            Symfony\Component\Security\Core\User\User: plaintext
```

- *XML*

```
    <!-- app/config/security.xml -->
    <srv:container xmlns="http://symfony.com/schema/dic/security"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:srv="http://symfony.com/schema/dic/services"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

        <config>
            <firewall name="secured_area" pattern="^/">
                <anonymous />
                <http-basic realm="Secured Demo Area" />
            </firewall>

            <access-control>
                <rule path="^/admin" role="ROLE_ADMIN" />
            </access-control>

            <provider name="in_memory">
                <memory>
                    <user name="ryan" password="ryanpass" roles="ROLE_USER" />
                    <user name="admin" password="kitten" roles="ROLE_ADMIN" />
                </memory>
            </provider>

            <encoder class="Symfony\Component\Security\Core\User\User" algorithm="plaintext" />
        </config>
    </srv:container>
```

- *PHP*

```
    // app/config/security.php
    $container->loadFromExtension('security', array(
        'firewalls' => array(
            'secured_area' => array(
                'pattern' => '^/',
                'anonymous' => array(),
                'http_basic' => array(
                    'realm' => 'Secured Demo Area',
                ),
            ),
        ),
        'access_control' => array(
            array('path' => '^/admin', 'role' => 'ROLE_ADMIN'),
        ),
        'providers' => array(
            'in_memory' => array(
                'memory' => array(
                    'users' => array(
                        'ryan' => array('password' => 'ryanpass', 'roles' => 'ROLE_USER'),
                        'admin' => array('password' => 'kitten', 'roles' => 'ROLE_ADMIN'),
                    ),
                ),
            ),
        ),
        'encoders' => array(
```

```
            'Symfony\Component\Security\Core\User\User' => 'plaintext',
        ),
    ));
```

Tip: A standard Symfony distribution separates the security configuration into a separate file (e.g. `app/config/security.yml`). If you don't have a separate security file, you can put the configuration directly into your main config file (e.g. `app/config/config.yml`).

The end result of this configuration is a fully-functional security system that looks like the following:

- There are two users in the system (`ryan` and `admin`);
- Users authenticate themselves via the basic HTTP authentication prompt;
- Any URL matching `/admin/*` is secured, and only the `admin` user can access it;
- All URLs *not* matching `/admin/*` are accessible by all users (and the user is never prompted to login).

Let's look briefly at how security works and how each part of the configuration comes into play.

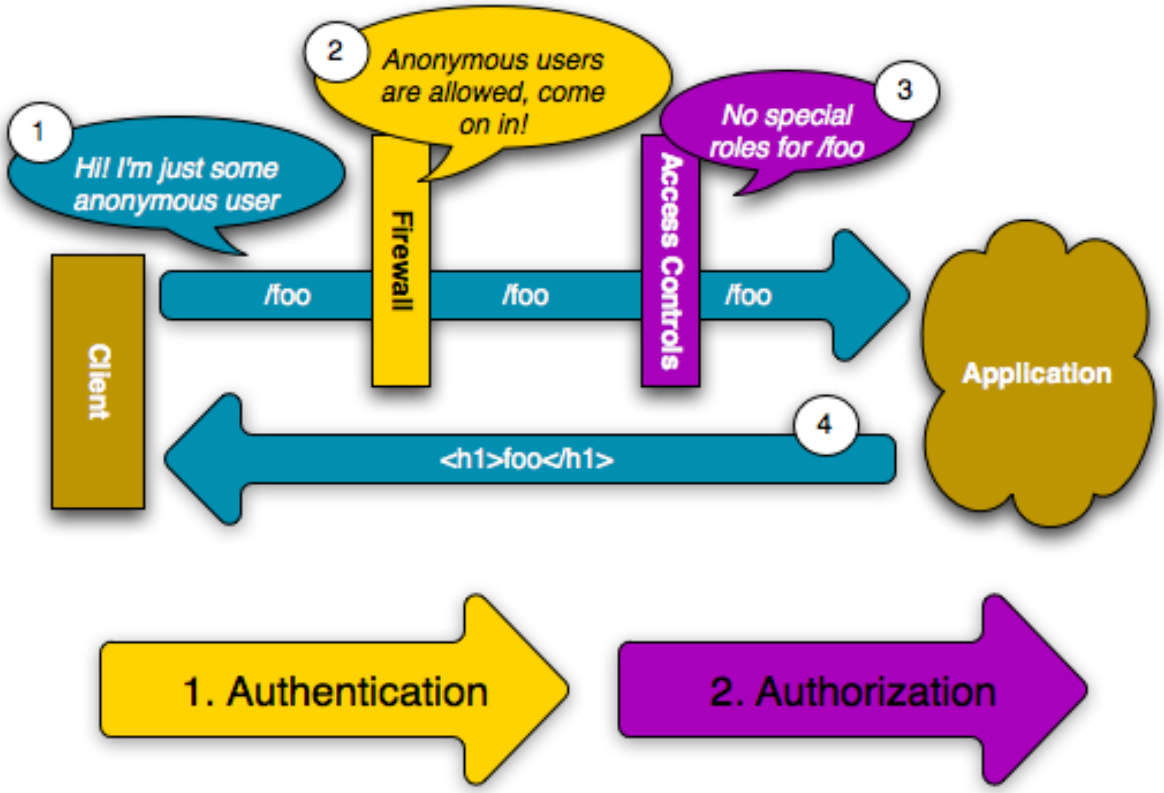
How Security Works: Authentication and Authorization

Symfony's security system works by determining who a user is (i.e. authentication) and then checking to see if that user should have access to a specific resource or URL.

Firewalls (Authentication)

When a user makes a request to a URL that's protected by a firewall, the security system is activated. The job of the firewall is to determine whether or not the user needs to be authenticated, and if he does, to send a response back to the user initiating the authentication process.

A firewall is activated when the URL of an incoming request matches the configured firewall's regular expression `pattern` config value. In this example, the `pattern` (`^/`) will match *every* incoming request. The fact that the firewall is activated does *not* mean, however, that the HTTP authentication username and password box is displayed for every URL. For example, any user can access `/foo` without being prompted to authenticate.

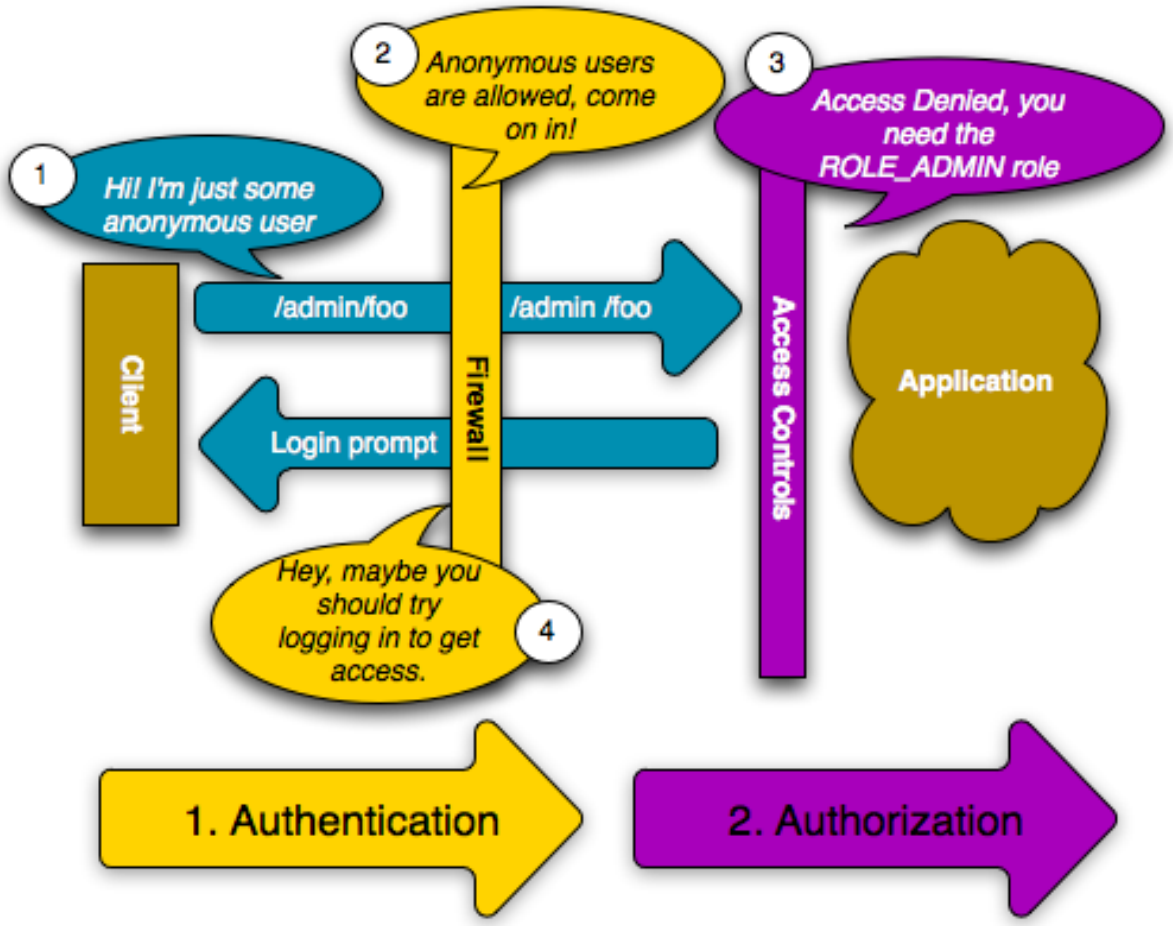


This works first because the firewall allows *anonymous users* via the `anonymous` configuration parameter. In other words, the firewall doesn't require the user to fully authenticate immediately. And because no special `role` is needed to access `/foo` (under the `access_control` section), the request can be fulfilled without ever asking the user to authenticate.

If you remove the `anonymous` key, the firewall will *always* make a user fully authenticate immediately.

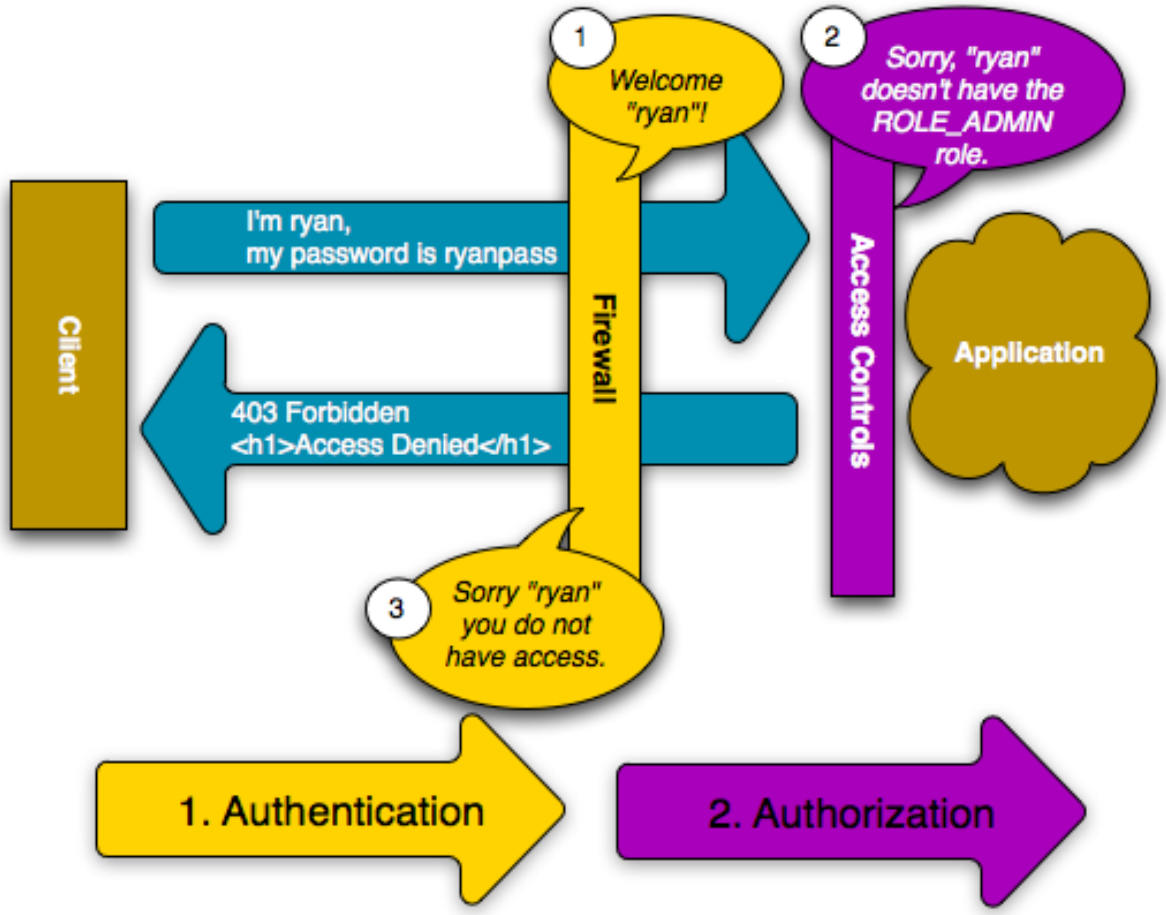
Access Controls (Authorization)

If a user requests `/admin/foo`, however, the process behaves differently. This is because of the `access_control` configuration section that says that any URL matching the regular expression pattern `^/admin` (i.e. `/admin` or anything matching `/admin/*`) requires the `ROLE_ADMIN` role. Roles are the basis for most authorization: a user can access `/admin/foo` only if it has the `ROLE_ADMIN` role.



Like before, when the user originally makes the request, the firewall doesn't ask for any identification. However, as soon as the access control layer denies the user access (because the anonymous user doesn't have the `ROLE_ADMIN` role), the firewall jumps into action and initiates the authentication process. The authentication process depends on the authentication mechanism you're using. For example, if you're using the form login authentication method, the user will be redirected to the login page. If you're using HTTP authentication, the user will be sent an HTTP 401 response so that the user sees the username and password box.

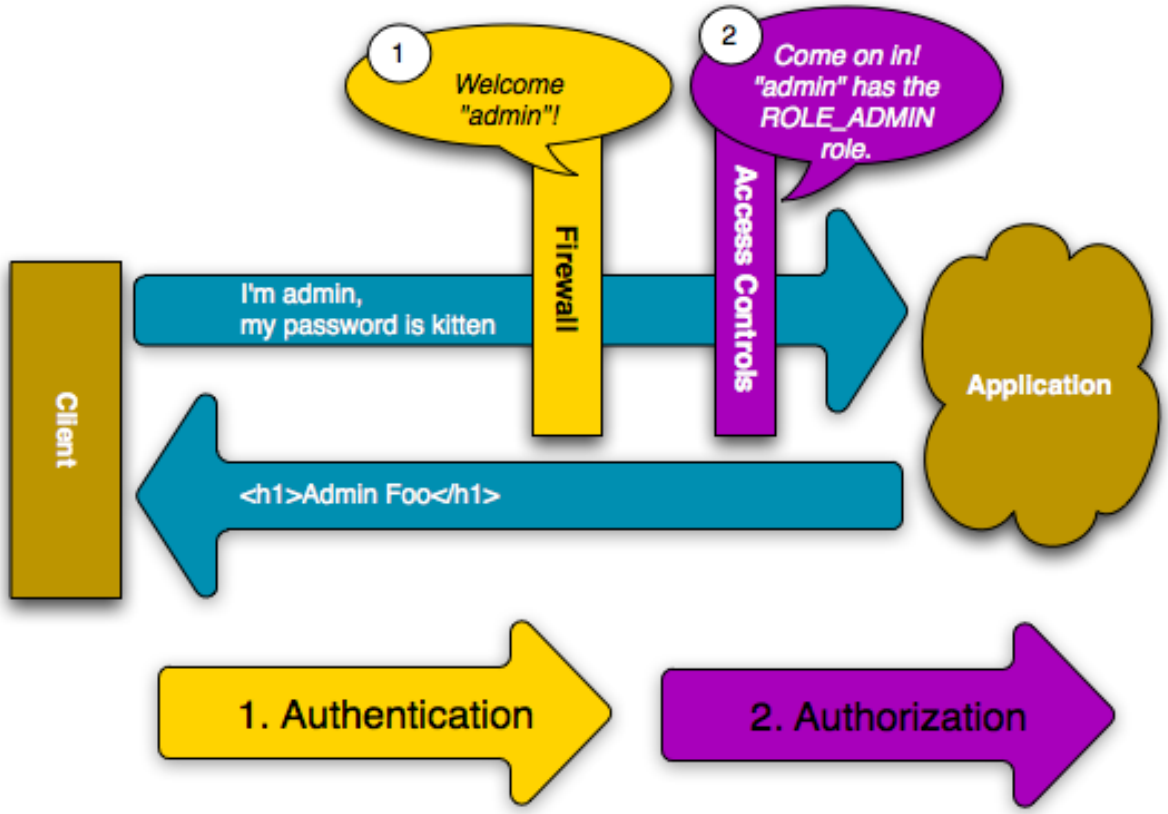
The user now has the opportunity to submit its credentials back to the application. If the credentials are valid, the original request can be re-tried.



In this example, the user `ryan` successfully authenticates with the firewall. But since `ryan` doesn't have the `ROLE_ADMIN` role, he's still denied access to `/admin/foo`. Ultimately, this means that the user will see some sort of message indicating that access has been denied.

Tip: When Symfony denies the user access, the user sees an error screen and receives a 403 HTTP status code (`Forbidden`). You can customize the access denied error screen by following the directions in the *Error Pages* cookbook entry to customize the 403 error page.

Finally, if the `admin` user requests `/admin/foo`, a similar process takes place, except now, after being authenticated, the access control layer will let the request pass through:



The request flow when a user requests a protected resource is straightforward, but incredibly flexible. As you'll see later, authentication can be handled in any number of ways, including via a form login, X.509 certificate, or by authenticating the user via Twitter. Regardless of the authentication method, the request flow is always the same:

1. A user accesses a protected resource;
2. The application redirects the user to the login form;
3. The user submits its credentials (e.g. username/password);
4. The firewall authenticates the user;
5. The authenticated user re-tries the original request.

Note: The *exact* process actually depends a little bit on which authentication mechanism you're using. For example, when using form login, the user submits its credentials to one URL that processes the form (e.g. `/login_check`) and then is redirected back to the originally requested URL (e.g. `/admin/foo`). But with HTTP authentication, the user submits its credentials directly to the original URL (e.g. `/admin/foo`) and then the page is returned to the user in that same request (i.e. no redirect).

These types of idiosyncrasies shouldn't cause you any problems, but they're good to keep in mind.

Tip: You'll also learn later how *anything* can be secured in Symfony2, including specific controllers, objects, or even PHP methods.

Using a Traditional Login Form

So far, you've seen how to blanket your application beneath a firewall and then protect access to certain areas with roles. By using HTTP Authentication, you can effortlessly tap into the native username/password box offered by all browsers. However, Symfony supports many authentication mechanisms out of the box. For details on all of them, see the Security Configuration Reference.

In this section, you'll enhance this process by allowing the user to authenticate via a traditional HTML login form.

First, enable form login under your firewall:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            form_login:
                login_path:  /login
                check_path:  /login_check
```

- *XML*

```xml
<!-- app/config/security.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

    <config>
        <firewall name="secured_area" pattern="^/">
            <anonymous />
            <form-login login_path="/login" check_path="/login_check" />
        </firewall>
    </config>
</srv:container>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            'pattern' => '^/',
            'anonymous' => array(),
            'form_login' => array(
                'login_path' => '/login',
                'check_path' => '/login_check',
            ),
        ),
    ),
));
```

Tip: If you don't need to customize your `login_path` or `check_path` values (the values used here are the default values), you can shorten your configuration:

- *YAML*

```
    form_login: ~
```

- *XML*

```
    <form-login />
```

- *PHP*

```
    'form_login' => array(),
```

Now, when the security system initiates the authentication process, it will redirect the user to the login form (`/login` by default). Implementing this login form visually is your job. First, create two routes: one that will display the login form (i.e. `/login`) and one that will handle the login form submission (i.e. `/login_check`):

- *YAML*

```
    # app/config/routing.yml
    login:
        pattern:   /login
        defaults:  { _controller: AcmeSecurityBundle:Security:login }
    login_check:
        pattern:   /login_check
```

- *XML*

```
    <!-- app/config/routing.xml -->
    <?xml version="1.0" encoding="UTF-8" ?>

    <routes xmlns="http://symfony.com/schema/routing"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

        <route id="login" pattern="/login">
            <default key="_controller">AcmeSecurityBundle:Security:login</default>
        </route>
        <route id="login_check" pattern="/login_check" />

    </routes>
```

- *PHP*

```
    // app/config/routing.php
    use Symfony\Component\Routing\RouteCollection;
    use Symfony\Component\Routing\Route;

    $collection = new RouteCollection();
    $collection->add('login', new Route('/login', array(
        '_controller' => 'AcmeDemoBundle:Security:login',
    )));
    $collection->add('login_check', new Route('/login_check', array()));

    return $collection;
```

Note: You will *not* need to implement a controller for the `/login_check` URL as the firewall will automatically catch and process any form submitted to this URL.

New in version 2.1: As of Symfony 2.1, you *must* have routes configured for your `login_path` (e.g. `/login`) and `check_path` (e.g. `/login_check`) URLs.

Notice that the name of the `login` route isn't important. What's important is that the URL of the route (`/login`) matches the `login_path` config value, as that's where the security system will redirect users that need to login.

Next, create the controller that will display the login form:

```
// src/Acme/SecurityBundle/Controller/Main;
namespace Acme\SecurityBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;

class SecurityController extends Controller
{
    public function loginAction()
    {
        $request = $this->getRequest();
        $session = $request->getSession();

        // get the login error if there is one
        if ($request->attributes->has(SecurityContext::AUTHENTICATION_ERROR)) {
            $error = $request->attributes->get(SecurityContext::AUTHENTICATION_ERROR);
        } else {
            $error = $session->get(SecurityContext::AUTHENTICATION_ERROR);
        }

        return $this->render('AcmeSecurityBundle:Security:login.html.twig', array(
            // last username entered by the user
            'last_username' => $session->get(SecurityContext::LAST_USERNAME),
            'error'         => $error,
        ));
    }
}
```

Don't let this controller confuse you. As you'll see in a moment, when the user submits the form, the security system automatically handles the form submission for you. If the user had submitted an invalid username or password, this controller reads the form submission error from the security system so that it can be displayed back to the user.

In other words, your job is to display the login form and any login errors that may have occurred, but the security system itself takes care of checking the submitted username and password and authenticating the user.

Finally, create the corresponding template:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    {#
        If you want to control the URL the user is redirected to on success (more details below)
        <input type="hidden" name="_target_path" value="/account" />
    #}
```

```
        <input type="submit" name="login" />
    </form>
```

- *PHP*

```
    <?php // src/Acme/SecurityBundle/Resources/views/Security/login.html.php ?>
    <?php if ($error): ?>
        <div><?php echo $error->getMessage() ?></div>
    <?php endif; ?>

    <form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="_username" value="<?php echo $last_username ?>" />

        <label for="password">Password:</label>
        <input type="password" id="password" name="_password" />

        <!--
            If you want to control the URL the user is redirected to on success (more details below)
            <input type="hidden" name="_target_path" value="/account" />
        -->

        <input type="submit" name="login" />
    </form>
```

Tip: The `error` variable passed into the template is an instance of `Symfony\Component\Security\Core\Exception\AuthenticationException`. It may contain more information - or even sensitive information - about the authentication failure, so use it wisely!

The form has very few requirements. First, by submitting the form to `/login_check` (via the `login_check` route), the security system will intercept the form submission and process the form for you automatically. Second, the security system expects the submitted fields to be called `_username` and `_password` (these field names can be *configured*).

And that's it! When you submit the form, the security system will automatically check the user's credentials and either authenticate the user or send the user back to the login form where the error can be displayed.

Let's review the whole process:

1. The user tries to access a resource that is protected;
2. The firewall initiates the authentication process by redirecting the user to the login form (`/login`);
3. The `/login` page renders login form via the route and controller created in this example;
4. The user submits the login form to `/login_check`;
5. The security system intercepts the request, checks the user's submitted credentials, authenticates the user if they are correct, and sends the user back to the login form if they are not.

By default, if the submitted credentials are correct, the user will be redirected to the original page that was requested (e.g. `/admin/foo`). If the user originally went straight to the login page, he'll be redirected to the homepage. This can be highly customized, allowing you to, for example, redirect the user to a specific URL.

For more details on this and how to customize the form login process in general, see How to customize your Form Login.

Avoid Common Pitfalls

When setting up your login form, watch out for a few common pitfalls.

1. Create the correct routes

First, be sure that you've defined the `/login` and `/login_check` routes correctly and that they correspond to the `login_path` and `check_path` config values. A misconfiguration here can mean that you're redirected to a 404 page instead of the login page, or that submitting the login form does nothing (you just see the login form over and over again).

2. Be sure the login page isn't secure

Also, be sure that the login page does *not* require any roles to be viewed. For example, the following configuration - which requires the `ROLE_ADMIN` role for all URLs (including the `/login` URL), will cause a redirect loop:

- *YAML*

```
access_control:
    - { path: ^/, roles: ROLE_ADMIN }
```

- *XML*

```
<access-control>
    <rule path="^/" role="ROLE_ADMIN" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/', 'role' => 'ROLE_ADMIN'),
),
```

Removing the access control on the `/login` URL fixes the problem:

- *YAML*

```
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/, roles: ROLE_ADMIN }
```

- *XML*

```
<access-control>
    <rule path="^/login" role="IS_AUTHENTICATED_ANONYMOUSLY" />
    <rule path="^/" role="ROLE_ADMIN" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/login', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY'),
    array('path' => '^/', 'role' => 'ROLE_ADMIN'),
),
```

Also, if your firewall does *not* allow for anonymous users, you'll need to create a special firewall that allows anonymous users for the login page:

- *YAML*

```
firewalls:
    login_firewall:
        pattern:    ^/login$
        anonymous:  ~
    secured_area:
        pattern:    ^/
        form_login: ~
```

- *XML*

```
<firewall name="login_firewall" pattern="^/login$">
    <anonymous />
</firewall>
<firewall name="secured_area" pattern="^/">
    <form_login />
</firewall>
```

- *PHP*

Authorization

The first step in security is always authentication: the process of verifying who the user is. With Symfony, authentication can be done in any way - via a form login, basic HTTP Authentication, or even via Facebook.

Once the user has been authenticated, authorization begins. Authorization provides a standard and powerful way to decide if a user can access any resource (a URL, a model object, a method call, ...). This works by assigning specific roles to each user, and then requiring different roles for different resources.

The process of authorization has two different sides:

1. The user has a specific set of roles;
2. A resource requires a specific role in order to be accessed.

In this section, you'll focus on how to secure different resources (e.g. URLs, method calls, etc) with different roles. Later, you'll learn more about how roles are created and assigned to users.

Securing Specific URL Patterns

The most basic way to secure part of your application is to secure an entire URL pattern. You've seen this already in the first example of this chapter, where anything matching the regular expression pattern `^/admin` requires the `ROLE_ADMIN` role.

You can define as many URL patterns as you need - each is a regular expression.

- *YAML*

```yaml
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/admin/users, roles: ROLE_SUPER_ADMIN }
        - { path: ^/admin, roles: ROLE_ADMIN }
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <!-- ... -->
    <rule path="^/admin/users" role="ROLE_SUPER_ADMIN" />
    <rule path="^/admin" role="ROLE_ADMIN" />
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    // ...
    'access_control' => array(
        array('path' => '^/admin/users', 'role' => 'ROLE_SUPER_ADMIN'),
        array('path' => '^/admin', 'role' => 'ROLE_ADMIN'),
    ),
));
```

Tip: Prepending the path with `^` ensures that only URLs *beginning* with the pattern are matched. For example, a path of simply `/admin` (without the `^`) would correctly match `/admin/foo` but would also match URLs like `/foo/admin`.

For each incoming request, Symfony2 tries to find a matching access control rule (the first one wins). If the user isn't authenticated yet, the authentication process is initiated (i.e. the user is given a chance to login). However, if the user *is* authenticated but doesn't have the required role, an `Symfony\Component\Security\Core\Exception\AccessDeniedException` exception is thrown, which you can handle and turn into a nice "access denied" error page for the user. See How to customize Error Pages for more information.

Since Symfony uses the first access control rule it matches, a URL like `/admin/users/new` will match the first rule and require only the `ROLE_SUPER_ADMIN` role. Any URL like `/admin/blog` will match the second rule and require `ROLE_ADMIN`.

Securing by IP

Certain situations may arise when you may need to restrict access to a given route based on IP. This is particularly relevant in the case of *Edge Side Includes* (ESI), for example, which utilize a route named "_internal". When ESI is used, the _internal route is required by the gateway cache to enable different caching options for subsections within a given page. This route comes with the ^/_internal prefix by default in the standard edition (assuming you've uncommented those lines from the routing file).

Here is an example of how you might secure this route from outside access:

- *YAML*

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/_internal, roles: IS_AUTHENTICATED_ANONYMOUSLY, ip: 127.0.0.1 }
```

- *XML*

```
<access-control>
    <rule path="^/_internal" role="IS_AUTHENTICATED_ANONYMOUSLY" ip="127.0.0.1" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/_internal', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY', 'ip' => '127.0.0.1'
),
```

Securing by Channel

Much like securing based on IP, requiring the use of SSL is as simple as adding a new access_control entry:

- *YAML*

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/cart/checkout, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel: https
```

- *XML*

```
<access-control>
    <rule path="^/cart/checkout" role="IS_AUTHENTICATED_ANONYMOUSLY" requires_channel="https" />
</access-control>
```

- *PHP*

```
    'access_control' => array(
        array('path' => '^/cart/checkout', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY', 'requires_chann
    ),
```

Securing a Controller

Protecting your application based on URL patterns is easy, but may not be fine-grained enough in certain cases. When necessary, you can easily force authorization from inside a controller:

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function helloAction($name)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }

    // ...
}
```

You can also choose to install and use the optional `JMSSecurityExtraBundle`, which can secure your controller using annotations:

```
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Secure(roles="ROLE_ADMIN")
 */
public function helloAction($name)
{
    // ...
}
```

For more information, see the JMSSecurityExtraBundle documentation. If you're using Symfony's Standard Distribution, this bundle is available by default. If not, you can easily download and install it.

Securing other Services

In fact, anything in Symfony can be protected using a strategy similar to the one seen in the previous section. For example, suppose you have a service (i.e. a PHP class) whose job is to send emails from one user to another. You can restrict use of this class - no matter where it's being used from - to users that have a specific role.

For more information on how you can use the security component to secure different services and methods in your application, see How to secure any Service or Method in your Application.

Access Control Lists (ACLs): Securing Individual Database Objects

Imagine you are designing a blog system where your users can comment on your posts. Now, you want a user to be able to edit his own comments, but not those of other users. Also, as the admin user, you yourself want to be able to edit *all* comments.

The security component comes with an optional access control list (ACL) system that you can use when you need to control access to individual instances of an object in your system. *Without* ACL, you can secure your system

so that only certain users can edit blog comments in general. But *with* ACL, you can restrict or allow access on a comment-by-comment basis.

For more information, see the cookbook article: Access Control Lists (ACLs).

Users

In the previous sections, you learned how you can protect different resources by requiring a set of *roles* for a resource. In this section we'll explore the other side of authorization: users.

Where do Users come from? (*User Providers*)

During authentication, the user submits a set of credentials (usually a username and password). The job of the authentication system is to match those credentials against some pool of users. So where does this list of users come from?

In Symfony2, users can come from anywhere - a configuration file, a database table, a web service, or anything else you can dream up. Anything that provides one or more users to the authentication system is known as a "user provider". Symfony2 comes standard with the two most common user providers: one that loads users from a configuration file and one that loads users from a database table.

Specifying Users in a Configuration File The easiest way to specify your users is directly in a configuration file. In fact, you've seen this already in the example in this chapter.

- *YAML*

```yaml
# app/config/security.yml
security:
    # ...
    providers:
        default_provider:
            memory:
                users:
                    ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                    admin: { password: kitten, roles: 'ROLE_ADMIN' }
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <!-- ... -->
    <provider name="default_provider">
        <memory>
            <user name="ryan" password="ryanpass" roles="ROLE_USER" />
            <user name="admin" password="kitten" roles="ROLE_ADMIN" />
        </memory>
    </provider>
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    // ...
    'providers' => array(
        'default_provider' => array(
```

```
                'memory' => array(
                    'users' => array(
                        'ryan' => array('password' => 'ryanpass', 'roles' => 'ROLE_USER'),
                        'admin' => array('password' => 'kitten', 'roles' => 'ROLE_ADMIN'),
                    ),
                ),
            ),
        ),
    ));
```

This user provider is called the "in-memory" user provider, since the users aren't stored anywhere in a database. The actual user object is provided by Symfony (`Symfony\Component\Security\Core\User\User`).

Tip: Any user provider can load users directly from configuration by specifying the `users` configuration parameter and listing the users beneath it.

Caution: If your username is completely numeric (e.g. `77`) or contains a dash (e.g. `user-name`), you should use that alternative syntax when specifying users in YAML:

```
users:
    - { name: 77, password: pass, roles: 'ROLE_USER' }
    - { name: user-name, password: pass, roles: 'ROLE_USER' }
```

For smaller sites, this method is quick and easy to setup. For more complex systems, you'll want to load your users from the database.

Loading Users from the Database If you'd like to load your users via the Doctrine ORM, you can easily do this by creating a `User` class and configuring the `entity` provider.

With this approach, you'll first create your own `User` class, which will be stored in the database.

```
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class User implements UserInterface
{
    /**
     * @ORM\Column(type="string", length="255")
     */
    protected $username;

    // ...
}
```

As far as the security system is concerned, the only requirement for your custom user class is that it implements the `Symfony\Component\Security\Core\User\UserInterface` interface. This means that your concept of a "user" can be anything, as long as it implements this interface.

New in version 2.1: In Symfony 2.1, the `equals` method was removed from `UserInterface`. If you need to override the default implementation of comparison logic, implement the new

`Symfony\Component\Security\Core\User\EquatableInterface` interface.

Note: The user object will be serialized and saved in the session during requests, therefore it is recommended that you implement the Serializable interface in your user object. This is especially important if your `User` class has a parent class with private properties.

Next, configure an `entity` user provider, and point it to your `User` class:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        main:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <provider name="main">
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'main' => array(
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'username'
        ),
    ),
));
```

With the introduction of this new provider, the authentication system will attempt to load a `User` object from the database by using the `username` field of that class.

Note: This example is just meant to show you the basic idea behind the `entity` provider. For a full working example, see How to load Security Users from the Database (the Entity Provider).

For more information on creating your own custom provider (e.g. if you needed to load users via a web service), see How to create a custom User Provider.

Encoding the User's Password

So far, for simplicity, all the examples have stored the users' passwords in plain text (whether those users are stored in a configuration file or in a database somewhere). Of course, in a real application, you'll want to encode your users' passwords for security reasons. This is easily accomplished by mapping your User class to one of several built-in "encoders". For example, to store your users in memory, but obscure their passwords via `sha1`, do the following:

- *YAML*

```
# app/config/security.yml
security:
    # ...
```

```
        providers:
            in_memory:
                memory:
                    users:
                        ryan:  { password: bb87a29949f3a1ee0559f8a57357487151281386, roles: 'ROLE_US
                        admin: { password: 74913f5cd5f61ec0bcfdb775414c2fb3d161b620, roles: 'ROLE_AD

        encoders:
            Symfony\Component\Security\Core\User\User:
                algorithm:   sha1
                iterations: 1
                encode_as_base64: false
```

- *XML*

```
    <!-- app/config/security.xml -->
    <config>
        <!-- ... -->
        <provider name="in_memory">
            <memory>
                <user name="ryan" password="bb87a29949f3a1ee0559f8a57357487151281386" roles="ROLE_US
                <user name="admin" password="74913f5cd5f61ec0bcfdb775414c2fb3d161b620" roles="ROLE_A
            </memory>
        </provider>

        <encoder class="Symfony\Component\Security\Core\User\User" algorithm="sha1" iterations="1" e
    </config>
```

- *PHP*

```
    // app/config/security.php
    $container->loadFromExtension('security', array(
        // ...
        'providers' => array(
            'in_memory' => array(
                'memory' => array(
                    'users' => array(
                        'ryan' => array('password' => 'bb87a29949f3a1ee0559f8a57357487151281386', 'r
                        'admin' => array('password' => '74913f5cd5f61ec0bcfdb775414c2fb3d161b620', '
                    ),
                ),
            ),
        ),
        'encoders' => array(
            'Symfony\Component\Security\Core\User\User' => array(
                'algorithm'         => 'sha1',
                'iterations'        => 1,
                'encode_as_base64'  => false,
            ),
        ),
    ));
```

By setting the `iterations` to `1` and the `encode_as_base64` to false, the password is simply run through the `sha1` algorithm one time and without any extra encoding. You can now calculate the hashed password either programmatically (e.g. `hash('sha1', 'ryanpass')`) or via some online tool like functions-online.com

If you're creating your users dynamically (and storing them in a database), you can use even tougher hashing algorithms and then rely on an actual password encoder object to help you encode passwords. For example, suppose your User object is `Acme\UserBundle\Entity\User` (like in the above example). First, configure the encoder for

that user:

- *YAML*

```
# app/config/security.yml
security:
    # ...

    encoders:
        Acme\UserBundle\Entity\User: sha512
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <!-- ... -->

    <encoder class="Acme\UserBundle\Entity\User" algorithm="sha512" />
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    // ...

    'encoders' => array(
        'Acme\UserBundle\Entity\User' => 'sha512',
    ),
));
```

In this case, you're using the stronger `sha512` algorithm. Also, since you've simply specified the algorithm (`sha512`) as a string, the system will default to hashing your password 5000 times in a row and then encoding it as base64. In other words, the password has been greatly obfuscated so that the hashed password can't be decoded (i.e. you can't determine the password from the hashed password).

If you have some sort of registration form for users, you'll need to be able to determine the hashed password so that you can set it on your user. No matter what algorithm you configure for your user object, the hashed password can always be determined in the following way from a controller:

```
$factory = $this->get('security.encoder_factory');
$user = new Acme\UserBundle\Entity\User();

$encoder = $factory->getEncoder($user);
$password = $encoder->encodePassword('ryanpass', $user->getSalt());
$user->setPassword($password);
```

Retrieving the User Object

After authentication, the `User` object of the current user can be accessed via the `security.context` service. From inside a controller, this will look like:

```
public function indexAction()
{
    $user = $this->get('security.context')->getToken()->getUser();
}
```

In a controller this can be shortcut to:

```
public function indexAction()
{
    $user = $this->getUser();
}
```

Note: Anonymous users are technically authenticated, meaning that the `isAuthenticated()` method of an anonymous user object will return true. To check if your user is actually authenticated, check for the `IS_AUTHENTICATED_FULLY` role.

In a Twig Template this object can be accessed via the `app.user` key, which calls the **:method:'GlobalVariables::getUser()<Symfony\\Bundle\\FrameworkBundle\\Templating\\GlobalVariables::getUser>'** method:

- *Twig*

```
<p>Username: {{ app.user.username }}</p>
```

Using Multiple User Providers

Each authentication mechanism (e.g. HTTP Authentication, form login, etc) uses exactly one user provider, and will use the first declared user provider by default. But what if you want to specify a few users via configuration and the rest of your users in the database? This is possible by creating a new provider that chains the two together:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        chain_provider:
            chain:
                providers: [in_memory, user_db]
        in_memory:
            users:
                foo: { password: test }
        user_db:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <provider name="chain_provider">
        <chain>
            <provider>in_memory</provider>
            <provider>user_db</provider>
        </chain>
    </provider>
    <provider name="in_memory">
        <user name="foo" password="test" />
    </provider>
    <provider name="user_db">
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'chain_provider' => array(
            'chain' => array(
                'providers' => array('in_memory', 'user_db'),
            ),
        ),
        'in_memory' => array(
            'users' => array(
                'foo' => array('password' => 'test'),
            ),
        ),
        'user_db' => array(
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'username'
        ),
    ),
));
```

Now, all authentication mechanisms will use the `chain_provider`, since it's the first specified. The `chain_provider` will, in turn, try to load the user from both the `in_memory` and `user_db` providers.

Tip: If you have no reasons to separate your `in_memory` users from your `user_db` users, you can accomplish this even more easily by combining the two sources into a single provider:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        main_provider:
            memory:
                users:
                    foo: { password: test }
            entity:
                class: Acme\UserBundle\Entity\User,
                property: username
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <provider name=="main_provider">
        <memory>
            <user name="foo" password="test" />
        </memory>
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'main_provider' => array(
            'memory' => array(
                'users' => array(
                    'foo' => array('password' => 'test'),
```

```
                ),
            ),
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'username'
        ),
    ),
));
```

You can also configure the firewall or individual authentication mechanisms to use a specific provider. Again, unless a provider is specified explicitly, the first provider is always used:

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        secured_area:
            # ...
            provider: user_db
            http_basic:
                realm: "Secured Demo Area"
                provider: in_memory
            form_login: ~
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall name="secured_area" pattern="^/" provider="user_db">
        <!-- ... -->
        <http-basic realm="Secured Demo Area" provider="in_memory" />
        <form-login />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            // ...
            'provider' => 'user_db',
            'http_basic' => array(
                // ...
                'provider' => 'in_memory',
            ),
            'form_login' => array(),
        ),
    ),
));
```

In this example, if a user tries to login via HTTP authentication, the authentication system will use the `in_memory` user provider. But if the user tries to login via the form login, the `user_db` provider will be used (since it's the default for the firewall as a whole).

For more information about user provider and firewall configuration, see the Security Configuration Reference.

Roles

The idea of a "role" is key to the authorization process. Each user is assigned a set of roles and then each resource requires one or more roles. If the user has the required roles, access is granted. Otherwise access is denied.

Roles are pretty simple, and are basically strings that you can invent and use as needed (though roles are objects internally). For example, if you need to start limiting access to the blog admin section of your website, you could protect that section using a `ROLE_BLOG_ADMIN` role. This role doesn't need to be defined anywhere - you can just start using it.

Note: All roles **must** begin with the `ROLE_` prefix to be managed by Symfony2. If you define your own roles with a dedicated `Role` class (more advanced), don't use the `ROLE_` prefix.

Hierarchical Roles

Instead of associating many roles to users, you can define role inheritance rules by creating a role hierarchy:

- *YAML*

```
# app/config/security.yml
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <role id="ROLE_ADMIN">ROLE_USER</role>
    <role id="ROLE_SUPER_ADMIN">ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH</role>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'role_hierarchy' => array(
        'ROLE_ADMIN'       => 'ROLE_USER',
        'ROLE_SUPER_ADMIN' => array('ROLE_ADMIN', 'ROLE_ALLOWED_TO_SWITCH'),
    ),
));
```

In the above configuration, users with `ROLE_ADMIN` role will also have the `ROLE_USER` role. The `ROLE_SUPER_ADMIN` role has `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH` and `ROLE_USER` (inherited from `ROLE_ADMIN`).

Logging Out

Usually, you'll also want your users to be able to log out. Fortunately, the firewall can handle this automatically for you when you activate the `logout` config parameter:

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
```

```
            secured_area:
                # ...
                logout:
                    path:   /logout
                    target: /
        # ...
```

- *XML*

```
    <!-- app/config/security.xml -->
    <config>
        <firewall name="secured_area" pattern="^/">
            <!-- ... -->
            <logout path="/logout" target="/" />
        </firewall>
        <!-- ... -->
    </config>
```

- *PHP*

```
    // app/config/security.php
    $container->loadFromExtension('security', array(
        'firewalls' => array(
            'secured_area' => array(
                // ...
                'logout' => array('path' => 'logout', 'target' => '/'),
            ),
        ),
        // ...
    ));
```

Once this is configured under your firewall, sending a user to `/logout` (or whatever you configure the `path` to be), will un-authenticate the current user. The user will then be sent to the homepage (the value defined by the `target` parameter). Both the `path` and `target` config parameters default to what's specified here. In other words, unless you need to customize them, you can omit them entirely and shorten your configuration:

- *YAML*

```
    logout: ~
```

- *XML*

```
    <logout />
```

- *PHP*

```
    'logout' => array(),
```

Note that you will *not* need to implement a controller for the `/logout` URL as the firewall takes care of everything. You may, however, want to create a route so that you can use it to generate the URL:

- *YAML*

```
    # app/config/routing.yml
    logout:
        pattern:   /logout
```

- *XML*

```
    <!-- app/config/routing.xml -->
    <?xml version="1.0" encoding="UTF-8" ?>
```

```
    <routes xmlns="http://symfony.com/schema/routing"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

        <route id="logout" pattern="/logout" />

    </routes>
```

- *PHP*

```
    // app/config/routing.php
    use Symfony\Component\Routing\RouteCollection;
    use Symfony\Component\Routing\Route;

    $collection = new RouteCollection();
    $collection->add('logout', new Route('/logout', array()));

    return $collection;
```

Once the user has been logged out, he will be redirected to whatever path is defined by the `target` parameter above (e.g. the `homepage`). For more information on configuring the logout, see the Security Configuration Reference.

Access Control in Templates

If you want to check if the current user has a role inside a template, use the built-in helper function:

- *Twig*

```
    {% if is_granted('ROLE_ADMIN') %}
        <a href="...">Delete</a>
    {% endif %}
```

- *PHP*

```
    <?php if ($view['security']->isGranted('ROLE_ADMIN')): ?>
        <a href="...">Delete</a>
    <?php endif; ?>
```

Note: If you use this function and are *not* at a URL where there is a firewall active, an exception will be thrown. Again, it's almost always a good idea to have a main firewall that covers all URLs (as has been shown in this chapter).

Access Control in Controllers

If you want to check if the current user has a role in your controller, use the `isGranted` method of the security context:

```
public function indexAction()
{
    // show different content to admin users
    if ($this->get('security.context')->isGranted('ROLE_ADMIN')) {
        // Load admin content here
    }
    // load other regular content here
}
```

Note: A firewall must be active or an exception will be thrown when the `isGranted` method is called. See the note above about templates for more details.

Impersonating a User

Sometimes, it's useful to be able to switch from one user to another without having to logout and login again (for instance when you are debugging or trying to understand a bug a user sees that you can't reproduce). This can be easily done by activating the `switch_user` firewall listener:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            # ...
            switch_user: true
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall>
        <!-- ... -->
        <switch-user />
    </firewall>
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main'=> array(
            // ...
            'switch_user' => true
        ),
    ),
));
```

To switch to another user, just add a query string with the `_switch_user` parameter and the username as the value to the current URL:

http://example.com/somewhere?_switch_user=thomas

To switch back to the original user, use the special `_exit` username:

http://example.com/somewhere?_switch_user=_exit

Of course, this feature needs to be made available to a small group of users. By default, access is restricted to users having the `ROLE_ALLOWED_TO_SWITCH` role. The name of this role can be modified via the `role` setting. For extra security, you can also change the query parameter name via the `parameter` setting:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
```

```
            // ...
            switch_user: { role: ROLE_ADMIN, parameter: _want_to_be_this_user }
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <!-- ... -->
        <switch-user role="ROLE_ADMIN" parameter="_want_to_be_this_user" />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main'=> array(
            // ...
            'switch_user' => array('role' => 'ROLE_ADMIN', 'parameter' => '_want_to_be_this_user
        ),
    ),
));
```

Stateless Authentication

By default, Symfony2 relies on a cookie (the Session) to persist the security context of the user. But if you use certificates or HTTP authentication for instance, persistence is not needed as credentials are available for each request. In that case, and if you don't need to store anything else between requests, you can activate the stateless authentication (which means that no cookie will be ever created by Symfony2):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            http_basic: ~
            stateless:  true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall stateless="true">
        <http-basic />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('http_basic' => array(), 'stateless' => true),
    ),
));
```

Note: If you use a form login, Symfony2 will create a cookie even if you set `stateless` to `true`.

Final Words

Security can be a deep and complex issue to solve correctly in your application. Fortunately, Symfony's security component follows a well-proven security model based around *authentication* and *authorization*. Authentication, which always happens first, is handled by a firewall whose job is to determine the identity of the user through several different methods (e.g. HTTP authentication, login form, etc). In the cookbook, you'll find examples of other methods for handling authentication, including how to implement a "remember me" cookie functionality.

Once a user is authenticated, the authorization layer can determine whether or not the user should have access to a specific resource. Most commonly, *roles* are applied to URLs, classes or methods and if the current user doesn't have that role, access is denied. The authorization layer, however, is much deeper, and follows a system of "voting" so that multiple parties can determine if the current user should have access to a given resource. Find out more about this and other topics in the cookbook.

Learn more from the Cookbook

- Forcing HTTP/HTTPS
- Blacklist users by IP address with a custom voter
- Access Control Lists (ACLs)
- How to add "Remember Me" Login Functionality

2.1.13 HTTP Cache

The nature of rich web applications means that they're dynamic. No matter how efficient your application, each request will always contain more overhead than serving a static file.

And for most Web applications, that's fine. Symfony2 is lightning fast, and unless you're doing some serious heavy-lifting, each request will come back quickly without putting too much stress on your server.

But as your site grows, that overhead can become a problem. The processing that's normally performed on every request should be done only once. This is exactly what caching aims to accomplish.

Caching on the Shoulders of Giants

The most effective way to improve performance of an application is to cache the full output of a page and then bypass the application entirely on each subsequent request. Of course, this isn't always possible for highly dynamic websites, or is it? In this chapter, we'll show you how the Symfony2 cache system works and why we think this is the best possible approach.

The Symfony2 cache system is different because it relies on the simplicity and power of the HTTP cache as defined in the HTTP specification. Instead of reinventing a caching methodology, Symfony2 embraces the standard that defines basic communication on the Web. Once you understand the fundamental HTTP validation and expiration caching models, you'll be ready to master the Symfony2 cache system.

For the purposes of learning how to cache with Symfony2, we'll cover the subject in four steps:

- **Step 1**: A *gateway cache*, or reverse proxy, is an independent layer that sits in front of your application. The reverse proxy caches responses as they're returned from your application and answers requests with cached

responses before they hit your application. Symfony2 provides its own reverse proxy, but any reverse proxy can be used.

- **Step 2**: *HTTP cache* headers are used to communicate with the gateway cache and any other caches between your application and the client. Symfony2 provides sensible defaults and a powerful interface for interacting with the cache headers.
- **Step 3**: HTTP *expiration and validation* are the two models used for determining whether cached content is *fresh* (can be reused from the cache) or *stale* (should be regenerated by the application).
- **Step 4**: *Edge Side Includes* (ESI) allow HTTP cache to be used to cache page fragments (even nested fragments) independently. With ESI, you can even cache an entire page for 60 minutes, but an embedded sidebar for only 5 minutes.

Since caching with HTTP isn't unique to Symfony, many articles already exist on the topic. If you're new to HTTP caching, we *highly* recommend Ryan Tomayko's article Things Caches Do. Another in-depth resource is Mark Nottingham's Cache Tutorial.

Caching with a Gateway Cache

When caching with HTTP, the *cache* is separated from your application entirely and sits between your application and the client making the request.

The job of the cache is to accept requests from the client and pass them back to your application. The cache will also receive responses back from your application and forward them on to the client. The cache is the "middle-man" of the request-response communication between the client and your application.

Along the way, the cache will store each response that is deemed "cacheable" (See *Introduction to HTTP Caching*). If the same resource is requested again, the cache sends the cached response to the client, ignoring your application entirely.

This type of cache is known as a HTTP gateway cache and many exist such as Varnish, Squid in reverse proxy mode, and the Symfony2 reverse proxy.

Types of Caches

But a gateway cache isn't the only type of cache. In fact, the HTTP cache headers sent by your application are consumed and interpreted by up to three different types of caches:

- *Browser caches*: Every browser comes with its own local cache that is mainly useful for when you hit "back" or for images and other assets. The browser cache is a *private* cache as cached resources aren't shared with anyone else.
- *Proxy caches*: A proxy is a *shared* cache as many people can be behind a single one. It's usually installed by large corporations and ISPs to reduce latency and network traffic.
- *Gateway caches*: Like a proxy, it's also a *shared* cache but on the server side. Installed by network administrators, it makes websites more scalable, reliable and performant.

Tip: Gateway caches are sometimes referred to as reverse proxy caches, surrogate caches, or even HTTP accelerators.

Note: The significance of *private* versus *shared* caches will become more obvious as we talk about caching responses containing content that is specific to exactly one user (e.g. account information).

Each response from your application will likely go through one or both of the first two cache types. These caches are outside of your control but follow the HTTP cache directions set in the response.

Symfony2 Reverse Proxy

Symfony2 comes with a reverse proxy (also called a gateway cache) written in PHP. Enable it and cacheable responses from your application will start to be cached right away. Installing it is just as easy. Each new Symfony2 application comes with a pre-configured caching kernel (`AppCache`) that wraps the default one (`AppKernel`). The caching Kernel *is* the reverse proxy.

To enable caching, modify the code of a front controller to use the caching kernel:

```
// web/app.php

require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';
require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
// wrap the default AppKernel with the AppCache one
$kernel = new AppCache($kernel);
$kernel->handle(Request::createFromGlobals())->send();
```

The caching kernel will immediately act as a reverse proxy - caching responses from your application and returning them to the client.

Tip: The cache kernel has a special `getLog()` method that returns a string representation of what happened in the cache layer. In the development environment, use it to debug and validate your cache strategy:

```
error_log($kernel->getLog());
```

The `AppCache` object has a sensible default configuration, but it can be finely tuned via a set of options you can set by overriding the `getOptions()` method:

```
// app/AppCache.php

use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;

class AppCache extends HttpCache
{
    protected function getOptions()
    {
        return array(
            'debug'                  => false,
            'default_ttl'            => 0,
            'private_headers'        => array('Authorization', 'Cookie'),
            'allow_reload'           => false,
            'allow_revalidate'       => false,
            'stale_while_revalidate' => 2,
            'stale_if_error'         => 60,
        );
    }
}
```

Tip: Unless overridden in `getOptions()`, the `debug` option will be set to automatically be the debug value of the wrapped `AppKernel`.

Here is a list of the main options:

- `default_ttl`: The number of seconds that a cache entry should be considered fresh when no explicit freshness information is provided in a response. Explicit `Cache-Control` or `Expires` headers override this value (default: `0`);
- `private_headers`: Set of request headers that trigger "private" `Cache-Control` behavior on responses that don't explicitly state whether the response is `public` or `private` via a `Cache-Control` directive. (default: `Authorization` and `Cookie`);
- `allow_reload`: Specifies whether the client can force a cache reload by including a `Cache-Control` "no-cache" directive in the request. Set it to `true` for compliance with RFC 2616 (default: `false`);
- `allow_revalidate`: Specifies whether the client can force a cache revalidate by including a `Cache-Control` "max-age=0" directive in the request. Set it to `true` for compliance with RFC 2616 (default: false);
- `stale_while_revalidate`: Specifies the default number of seconds (the granularity is the second as the Response TTL precision is a second) during which the cache can immediately return a stale response while it revalidates it in the background (default: `2`); this setting is overridden by the `stale-while-revalidate` HTTP `Cache-Control` extension (see RFC 5861);
- `stale_if_error`: Specifies the default number of seconds (the granularity is the second) during which the cache can serve a stale response when an error is encountered (default: `60`). This setting is overridden by the `stale-if-error` HTTP `Cache-Control` extension (see RFC 5861).

If `debug` is `true`, Symfony2 automatically adds a `X-Symfony-Cache` header to the response containing useful information about cache hits and misses.

Changing from one Reverse Proxy to Another

The Symfony2 reverse proxy is a great tool to use when developing your website or when you deploy your website to a shared host where you cannot install anything beyond PHP code. But being written in PHP, it cannot be as fast as a proxy written in C. That's why we highly recommend you to use Varnish or Squid on your production servers if possible. The good news is that the switch from one proxy server to another is easy and transparent as no code modification is needed in your application. Start easy with the Symfony2 reverse proxy and upgrade later to Varnish when your traffic increases.
For more information on using Varnish with Symfony2, see the How to use Varnish cookbook chapter.

Note: The performance of the Symfony2 reverse proxy is independent of the complexity of the application. That's because the application kernel is only booted when the request needs to be forwarded to it.

Introduction to HTTP Caching

To take advantage of the available cache layers, your application must be able to communicate which responses are cacheable and the rules that govern when/how that cache should become stale. This is done by setting HTTP cache headers on the response.

Tip: Keep in mind that "HTTP" is nothing more than the language (a simple text language) that web clients (e.g. browsers) and web servers use to communicate with each other. When we talk about HTTP caching, we're talking about the part of that language that allows clients and servers to exchange information related to caching.

HTTP specifies four response cache headers that we're concerned with:

- `Cache-Control`

- `Expires`
- `ETag`
- `Last-Modified`

The most important and versatile header is the `Cache-Control` header, which is actually a collection of various cache information.

Note: Each of the headers will be explained in full detail in the *HTTP Expiration and Validation* section.

The Cache-Control Header

The `Cache-Control` header is unique in that it contains not one, but various pieces of information about the cacheability of a response. Each piece of information is separated by a comma:

Cache-Control: private, max-age=0, must-revalidate

Cache-Control: max-age=3600, must-revalidate

Symfony provides an abstraction around the `Cache-Control` header to make its creation more manageable:

```
$response = new Response();

// mark the response as either public or private
$response->setPublic();
$response->setPrivate();

// set the private or shared max age
$response->setMaxAge(600);
$response->setSharedMaxAge(600);

// set a custom Cache-Control directive
$response->headers->addCacheControlDirective('must-revalidate', true);
```

Public vs Private Responses

Both gateway and proxy caches are considered "shared" caches as the cached content is shared by more than one user. If a user-specific response were ever mistakenly stored by a shared cache, it might be returned later to any number of different users. Imagine if your account information were cached and then returned to every subsequent user who asked for their account page!

To handle this situation, every response may be set to be public or private:

- *public*: Indicates that the response may be cached by both private and shared caches;
- *private*: Indicates that all or part of the response message is intended for a single user and must not be cached by a shared cache.

Symfony conservatively defaults each response to be private. To take advantage of shared caches (like the Symfony2 reverse proxy), the response will need to be explicitly set as public.

Safe Methods

HTTP caching only works for "safe" HTTP methods (like GET and HEAD). Being safe means that you never change the application's state on the server when serving the request (you can of course log information, cache data, etc). This has two very reasonable consequences:

- You should *never* change the state of your application when responding to a GET or HEAD request. Even if you don't use a gateway cache, the presence of proxy caches mean that any GET or HEAD request may or may not actually hit your server.
- Don't expect PUT, POST or DELETE methods to cache. These methods are meant to be used when mutating the state of your application (e.g. deleting a blog post). Caching them would prevent certain requests from hitting and mutating your application.

Caching Rules and Defaults

HTTP 1.1 allows caching anything by default unless there is an explicit `Cache-Control` header. In practice, most caches do nothing when requests have a cookie, an authorization header, use a non-safe method (i.e. PUT, POST, DELETE), or when responses have a redirect status code.

Symfony2 automatically sets a sensible and conservative `Cache-Control` header when none is set by the developer by following these rules:

- If no cache header is defined (`Cache-Control`, `Expires`, `ETag` or `Last-Modified`), `Cache-Control` is set to `no-cache`, meaning that the response will not be cached;
- If `Cache-Control` is empty (but one of the other cache headers is present), its value is set to `private, must-revalidate`;
- But if at least one `Cache-Control` directive is set, and no 'public' or `private` directives have been explicitly added, Symfony2 adds the `private` directive automatically (except when `s-maxage` is set).

HTTP Expiration and Validation

The HTTP specification defines two caching models:

- With the expiration model, you simply specify how long a response should be considered "fresh" by including a `Cache-Control` and/or an `Expires` header. Caches that understand expiration will not make the same request until the cached version reaches its expiration time and becomes "stale".
- When pages are really dynamic (i.e. their representation changes often), the validation model is often necessary. With this model, the cache stores the response, but asks the server on each request whether or not the cached response is still valid. The application uses a unique response identifier (the `Etag` header) and/or a timestamp (the `Last-Modified` header) to check if the page has changed since being cached.

The goal of both models is to never generate the same response twice by relying on a cache to store and return "fresh" responses.

Reading the HTTP Specification

The HTTP specification defines a simple but powerful language in which clients and servers can communicate. As a web developer, the request-response model of the specification dominates our work. Unfortunately, the actual specification document - RFC 2616 - can be difficult to read.
There is an on-going effort (HTTP Bis) to rewrite the RFC 2616. It does not describe a new version of HTTP, but mostly clarifies the original HTTP specification. The organization is also improved as the specification is split into seven parts; everything related to HTTP caching can be found in two dedicated parts (P4 - Conditional Requests and P6 - Caching: Browser and intermediary caches).
As a web developer, we strongly urge you to read the specification. Its clarity and power - even more than ten years after its creation - is invaluable. Don't be put-off by the appearance of the spec - its contents are much more beautiful than its cover.

Expiration

The expiration model is the more efficient and straightforward of the two caching models and should be used whenever possible. When a response is cached with an expiration, the cache will store the response and return it directly without hitting the application until it expires.

The expiration model can be accomplished using one of two, nearly identical, HTTP headers: `Expires` or `Cache-Control`.

Expiration with the `Expires` Header

According to the HTTP specification, "the `Expires` header field gives the date/time after which the response is considered stale." The `Expires` header can be set with the `setExpires() Response` method. It takes a `DateTime` instance as an argument:

```
$date = new DateTime();
$date->modify('+600 seconds');

$response->setExpires($date);
```

The resulting HTTP header will look like this:

```
Expires: Thu, 01 Mar 2011 16:00:00 GMT
```

Note: The `setExpires()` method automatically converts the date to the GMT timezone as required by the specification.

Note that in HTTP versions before 1.1 the origin server wasn't required to send the `Date` header. Consequently the cache (e.g. the browser) might need to rely onto his local clock to evaluate the `Expires` header making the lifetime calculation vulnerable to clock skew. Another limitation of the `Expires` header is that the specification states that "HTTP/1.1 servers should not send `Expires` dates more than one year in the future."

Expiration with the `Cache-Control` Header

Because of the `Expires` header limitations, most of the time, you should use the `Cache-Control` header instead. Recall that the `Cache-Control` header is used to specify many different cache directives. For expiration, there are two directives, `max-age` and `s-maxage`. The first one is used by all caches, whereas the second one is only taken into account by shared caches:

```
// Sets the number of seconds after which the response
// should no longer be considered fresh
$response->setMaxAge(600);

// Same as above but only for shared caches
$response->setSharedMaxAge(600);
```

The `Cache-Control` header would take on the following format (it may have additional directives):

```
Cache-Control: max-age=600, s-maxage=600
```

Validation

When a resource needs to be updated as soon as a change is made to the underlying data, the expiration model falls short. With the expiration model, the application won't be asked to return the updated response until the cache finally

becomes stale.

The validation model addresses this issue. Under this model, the cache continues to store responses. The difference is that, for each request, the cache asks the application whether or not the cached response is still valid. If the cache *is* still valid, your application should return a 304 status code and no content. This tells the cache that it's ok to return the cached response.

Under this model, you mainly save bandwidth as the representation is not sent twice to the same client (a 304 response is sent instead). But if you design your application carefully, you might be able to get the bare minimum data needed to send a 304 response and save CPU also (see below for an implementation example).

Tip: The 304 status code means "Not Modified". It's important because with this status code do *not* contain the actual content being requested. Instead, the response is simply a light-weight set of directions that tell cache that it should use its stored version.

Like with expiration, there are two different HTTP headers that can be used to implement the validation model: `ETag` and `Last-Modified`.

Validation with the `ETag` Header

The `ETag` header is a string header (called the "entity-tag") that uniquely identifies one representation of the target resource. It's entirely generated and set by your application so that you can tell, for example, if the `/about` resource that's stored by the cache is up-to-date with what your application would return. An `ETag` is like a fingerprint and is used to quickly compare if two different versions of a resource are equivalent. Like fingerprints, each `ETag` must be unique across all representations of the same resource.

Let's walk through a simple implementation that generates the ETag as the md5 of the content:

```
public function indexAction()
{
    $response = $this->render('MyBundle:Main:index.html.twig');
    $response->setETag(md5($response->getContent()));
    $response->isNotModified($this->getRequest());

    return $response;
}
```

The `Response::isNotModified()` method compares the `ETag` sent with the `Request` with the one set on the `Response`. If the two match, the method automatically sets the `Response` status code to 304.

This algorithm is simple enough and very generic, but you need to create the whole `Response` before being able to compute the ETag, which is sub-optimal. In other words, it saves on bandwidth, but not CPU cycles.

In the *Optimizing your Code with Validation* section, we'll show how validation can be used more intelligently to determine the validity of a cache without doing so much work.

Tip: Symfony2 also supports weak ETags by passing `true` as the second argument to the **:method:'Symfony\\Component\\HttpFoundation\\Response::setETag'** method.

Validation with the `Last-Modified` Header

The `Last-Modified` header is the second form of validation. According to the HTTP specification, "The `Last-Modified` header field indicates the date and time at which the origin server believes the representation was last modified." In other words, the application decides whether or not the cached content has been updated based on whether or not it's been updated since the response was cached.

For instance, you can use the latest update date for all the objects needed to compute the resource representation as the value for the `Last-Modified` header value:

```
public function showAction($articleSlug)
{
    // ...

    $articleDate = new \DateTime($article->getUpdatedAt());
    $authorDate = new \DateTime($author->getUpdatedAt());

    $date = $authorDate > $articleDate ? $authorDate : $articleDate;

    $response->setLastModified($date);
    $response->isNotModified($this->getRequest());

    return $response;
}
```

The `Response::isNotModified()` method compares the `If-Modified-Since` header sent by the request with the `Last-Modified` header set on the response. If they are equivalent, the `Response` will be set to a 304 status code.

Note: The `If-Modified-Since` request header equals the `Last-Modified` header of the last response sent to the client for the particular resource. This is how the client and server communicate with each other and decide whether or not the resource has been updated since it was cached.

Optimizing your Code with Validation

The main goal of any caching strategy is to lighten the load on the application. Put another way, the less you do in your application to return a 304 response, the better. The `Response::isNotModified()` method does exactly that by exposing a simple and efficient pattern:

```
public function showAction($articleSlug)
{
    // Get the minimum information to compute
    // the ETag or the Last-Modified value
    // (based on the Request, data is retrieved from
    // a database or a key-value store for instance)
    $article = // ...

    // create a Response with a ETag and/or a Last-Modified header
    $response = new Response();
    $response->setETag($article->computeETag());
    $response->setLastModified($article->getPublishedAt());

    // Check that the Response is not modified for the given Request
    if ($response->isNotModified($this->getRequest())) {
        // return the 304 Response immediately
        return $response;
    } else {
        // do more work here - like retrieving more data
        $comments = // ...

        // or render a template with the $response you've already started
        return $this->render(
            'MyBundle:MyController:article.html.twig',
```

```
            array('article' => $article, 'comments' => $comments),
            $response
        );
    }
}
```

When the `Response` is not modified, the `isNotModified()` automatically sets the response status code to `304`, removes the content, and removes some headers that must not be present for `304` responses (see **:method:'Symfony\\Component\\HttpFoundation\\Response::setNotModified'**).

Varying the Response

So far, we've assumed that each URI has exactly one representation of the target resource. By default, HTTP caching is done by using the URI of the resource as the cache key. If two people request the same URI of a cacheable resource, the second person will receive the cached version.

Sometimes this isn't enough and different versions of the same URI need to be cached based on one or more request header values. For instance, if you compress pages when the client supports it, any given URI has two representations: one when the client supports compression, and one when it does not. This determination is done by the value of the `Accept-Encoding` request header.

In this case, we need the cache to store both a compressed and uncompressed version of the response for the particular URI and return them based on the request's `Accept-Encoding` value. This is done by using the `Vary` response header, which is a comma-separated list of different headers whose values trigger a different representation of the requested resource:

```
Vary: Accept-Encoding, User-Agent
```

Tip: This particular `Vary` header would cache different versions of each resource based on the URI and the value of the `Accept-Encoding` and `User-Agent` request header.

The `Response` object offers a clean interface for managing the `Vary` header:

```
// set one vary header
$response->setVary('Accept-Encoding');

// set multiple vary headers
$response->setVary(array('Accept-Encoding', 'User-Agent'));
```

The `setVary()` method takes a header name or an array of header names for which the response varies.

Expiration and Validation

You can of course use both validation and expiration within the same `Response`. As expiration wins over validation, you can easily benefit from the best of both worlds. In other words, by using both expiration and validation, you can instruct the cache to serve the cached content, while checking back at some interval (the expiration) to verify that the content is still valid.

More Response Methods

The Response class provides many more methods related to the cache. Here are the most useful ones:

```
// Marks the Response stale
$response->expire();

// Force the response to return a proper 304 response with no content
$response->setNotModified();
```

Additionally, most cache-related HTTP headers can be set via the single `setCache()` method:

```
// Set cache settings in one call
$response->setCache(array(
    'etag'          => $etag,
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
    'public'        => true,
    // 'private'    => true,
));
```

Using Edge Side Includes

Gateway caches are a great way to make your website perform better. But they have one limitation: they can only cache whole pages. If you can't cache whole pages or if parts of a page has "more" dynamic parts, you are out of luck. Fortunately, Symfony2 provides a solution for these cases, based on a technology called ESI, or Edge Side Includes. Akamaï wrote this specification almost 10 years ago, and it allows specific parts of a page to have a different caching strategy than the main page.

The ESI specification describes tags you can embed in your pages to communicate with the gateway cache. Only one tag is implemented in Symfony2, `include`, as this is the only useful one outside of Akamaï context:

```
<html>
    <body>
        Some content

        <!-- Embed the content of another page here -->
        <esi:include src="http://..." />

        More content
    </body>
</html>
```

Note: Notice from the example that each ESI tag has a fully-qualified URL. An ESI tag represents a page fragment that can be fetched via the given URL.

When a request is handled, the gateway cache fetches the entire page from its cache or requests it from the backend application. If the response contains one or more ESI tags, these are processed in the same way. In other words, the gateway cache either retrieves the included page fragment from its cache or requests the page fragment from the backend application again. When all the ESI tags have been resolved, the gateway cache merges each into the main page and sends the final content to the client.

All of this happens transparently at the gateway cache level (i.e. outside of your application). As you'll see, if you choose to take advantage of ESI tags, Symfony2 makes the process of including them almost effortless.

Using ESI in Symfony2

First, to use ESI, be sure to enable it in your application configuration:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    esi: { enabled: true }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config ...>
    <!-- ... -->
    <framework:esi enabled="true" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'esi'    => array('enabled' => true),
));
```

Now, suppose we have a page that is relatively static, except for a news ticker at the bottom of the content. With ESI, we can cache the news ticker independent of the rest of the page.

```
public function indexAction()
{
    $response = $this->render('MyBundle:MyController:index.html.twig');
    $response->setSharedMaxAge(600);

    return $response;
}
```

In this example, we've given the full-page cache a lifetime of ten minutes. Next, let's include the news ticker in the template by embedding an action. This is done via the `render` helper (See *Embedding Controllers* for more details).

As the embedded content comes from another page (or controller for that matter), Symfony2 uses the standard `render` helper to configure ESI tags:

- *Twig*

```
{% render '...:news' with {}, {'standalone': true} %}
```

- *PHP*

```
<?php echo $view['actions']->render('...:news', array(), array('standalone' => true)) ?>
```

By setting `standalone` to `true`, you tell Symfony2 that the action should be rendered as an ESI tag. You might be wondering why you would want to use a helper instead of just writing the ESI tag yourself. That's because using a helper makes your application work even if there is no gateway cache installed. Let's see how it works.

When standalone is `false` (the default), Symfony2 merges the included page content within the main one before sending the response to the client. But when standalone is `true`, *and* if Symfony2 detects that it's talking to a gateway cache that supports ESI, it generates an ESI include tag. But if there is no gateway cache or if it does not support ESI, Symfony2 will just merge the included page content within the main one as it would have done were standalone set to `false`.

Note: Symfony2 detects if a gateway cache supports ESI via another Akamaï specification that is supported out of the box by the Symfony2 reverse proxy.

The embedded action can now specify its own caching rules, entirely independent of the master page.

```
public function newsAction()
{
  // ...

  $response->setSharedMaxAge(60);
}
```

With ESI, the full page cache will be valid for 600 seconds, but the news component cache will only last for 60 seconds.

A requirement of ESI, however, is that the embedded action be accessible via a URL so the gateway cache can fetch it independently of the rest of the page. Of course, an action can't be accessed via a URL unless it has a route that points to it. Symfony2 takes care of this via a generic route and controller. For the ESI include tag to work properly, you must define the `_internal` route:

- *YAML*

```
# app/config/routing.yml
_internal:
    resource: "@FrameworkBundle/Resources/config/routing/internal.xml"
    prefix:   /_internal
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@FrameworkBundle/Resources/config/routing/internal.xml" prefix="/_internal
</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection->addCollection($loader->import('@FrameworkBundle/Resources/config/routing/internal.x

return $collection;
```

Tip: Since this route allows all actions to be accessed via a URL, you might want to protect it by using the Symfony2 firewall feature (by allowing access to your reverse proxy's IP range). See the *Securing by IP* section of the Security Chapter for more information on how to do this.

One great advantage of this caching strategy is that you can make your application as dynamic as needed and at the same time, hit the application as little as possible.

Note: Once you start using ESI, remember to always use the `s-maxage` directive instead of `max-age`. As the browser only ever receives the aggregated resource, it is not aware of the sub-components, and so it will obey the `max-age` directive and cache the entire page. And you don't want that.

The `render` helper supports two other useful options:

- `alt`: used as the `alt` attribute on the ESI tag, which allows you to specify an alternative URL to be used if the `src` cannot be found;
- `ignore_errors`: if set to true, an `onerror` attribute will be added to the ESI with a value of `continue` indicating that, in the event of a failure, the gateway cache will simply remove the ESI tag silently.

Cache Invalidation

> "There are only two hard things in Computer Science: cache invalidation and naming things." –Phil Karlton

You should never need to invalidate cached data because invalidation is already taken into account natively in the HTTP cache models. If you use validation, you never need to invalidate anything by definition; and if you use expiration and need to invalidate a resource, it means that you set the expires date too far away in the future.

Note: Since invalidation is a topic specific to each type of reverse proxy, if you don't worry about invalidation, you can switch between reverse proxies without changing anything in your application code.

Actually, all reverse proxies provide ways to purge cached data, but you should avoid them as much as possible. The most standard way is to purge the cache for a given URL by requesting it with the special `PURGE` HTTP method.

Here is how you can configure the Symfony2 reverse proxy to support the `PURGE` HTTP method:

```
// app/AppCache.php
class AppCache extends Cache
{
    protected function invalidate(Request $request)
    {
        if ('PURGE' !== $request->getMethod()) {
            return parent::invalidate($request);
        }

        $response = new Response();
        if (!$this->getStore()->purge($request->getUri())) {
            $response->setStatusCode(404, 'Not purged');
        } else {
            $response->setStatusCode(200, 'Purged');
        }

        return $response;
    }
}
```

Caution: You must protect the `PURGE` HTTP method somehow to avoid random people purging your cached data.

Summary

Symfony2 was designed to follow the proven rules of the road: HTTP. Caching is no exception. Mastering the Symfony2 cache system means becoming familiar with the HTTP cache models and using them effectively. This means that, instead of relying only on Symfony2 documentation and code examples, you have access to a world of knowledge related to HTTP caching and gateway caches such as Varnish.

Learn more from the Cookbook

- How to use Varnish to speed up my Website

2.1.14 Translations

The term "internationalization" (often abbreviated i18n) refers to the process of abstracting strings and other locale-specific pieces out of your application and into a layer where they can be translated and converted based on the user's locale (i.e. language and country). For text, this means wrapping each with a function capable of translating the text (or "message") into the language of the user:

```
// text will *always* print out in English
echo 'Hello World';

// text can be translated into the end-user's language or default to English
echo $translator->trans('Hello World');
```

Note: The term *locale* refers roughly to the user's language and country. It can be any string that your application then uses to manage translations and other format differences (e.g. currency format). We recommended the ISO639-1 *language* code, an underscore (_), then the ISO3166 *country* code (e.g. `fr_FR` for French/France).

In this chapter, we'll learn how to prepare an application to support multiple locales and then how to create translations for multiple locales. Overall, the process has several common steps:

1. Enable and configure Symfony's `Translation` component;
2. Abstract strings (i.e. "messages") by wrapping them in calls to the `Translator`;
3. Create translation resources for each supported locale that translate each message in the application;
4. Determine, set and manage the user's locale for the request and optionally on the user's entire session.

Configuration

Translations are handled by a `Translator` service that uses the user's locale to lookup and return translated messages. Before using it, enable the `Translator` in your configuration:

- *YAML*

```
# app/config/config.yml
framework:
    translator: { fallback: en }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:translator fallback="en" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    'translator' => array('fallback' => 'en'),
));
```

The `fallback` option defines the fallback locale when a translation does not exist in the user's locale.

Tip: When a translation does not exist for a locale, the translator first tries to find the translation for the language (`fr` if the locale is `fr_FR` for instance). If this also fails, it looks for a translation using the fallback locale.

The locale used in translations is the one stored on the request. This is typically set via a `_locale` attribute on your routes (see *The Locale and the URL*).

Basic Translation

Translation of text is done through the `translator` service (`Symfony\Component\Translation\Translator`). To translate a block of text (called a *message*), use the **:method:'Symfony\\Component\\Translation\\Translator::trans'** method. Suppose, for example, that we're translating a simple message from inside a controller:

```
public function indexAction()
{
    $t = $this->get('translator')->trans('Symfony2 is great');

    return new Response($t);
}
```

When this code is executed, Symfony2 will attempt to translate the message "Symfony2 is great" based on the `locale` of the user. For this to work, we need to tell Symfony2 how to translate the message via a "translation resource", which is a collection of message translations for a given locale. This "dictionary" of translations can be created in several different formats, XLIFF being the recommended format:

- *XML*

```
<!-- messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony2 is great</source>
                <target>J'aime Symfony2</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
// messages.fr.php
return array(
    'Symfony2 is great' => 'J\'aime Symfony2',
);
```

- *YAML*

```
# messages.fr.yml
Symfony2 is great: J'aime Symfony2
```

Now, if the language of the user's locale is French (e.g. `fr_FR` or `fr_BE`), the message will be translated into `J'aime Symfony2`.

The Translation Process

To actually translate the message, Symfony2 uses a simple process:

- The `locale` of the current user, which is stored on the request (or stored as `_locale` on the session), is determined;
- A catalog of translated messages is loaded from translation resources defined for the `locale` (e.g. `fr_FR`). Messages from the fallback locale are also loaded and added to the catalog if they don't already exist. The end result is a large "dictionary" of translations. See *Message Catalogues* for more details;
- If the message is located in the catalog, the translation is returned. If not, the translator returns the original message.

When using the `trans()` method, Symfony2 looks for the exact string inside the appropriate message catalog and returns it (if it exists).

Message Placeholders

Sometimes, a message containing a variable needs to be translated:

```
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello '.$name);

    return new Response($t);
}
```

However, creating a translation for this string is impossible since the translator will try to look up the exact message, including the variable portions (e.g. "Hello Ryan" or "Hello Fabien"). Instead of writing a translation for every possible iteration of the `$name` variable, we can replace the variable with a "placeholder":

```
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello %name%', array('%name%' => $name));

    new Response($t);
}
```

Symfony2 will now look for a translation of the raw message (`Hello %name%`) and *then* replace the placeholders with their values. Creating a translation is done just as before:

- *XML*

```
<!-- messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Hello %name%</source>
                <target>Bonjour %name%</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
// messages.fr.php
return array(
    'Hello %name%' => 'Bonjour %name%',
);
```

- *YAML*

```
# messages.fr.yml
'Hello %name%': Hello %name%
```

Note: The placeholders can take on any form as the full message is reconstructed using the PHP strtr function. However, the `%var%` notation is required when translating in Twig templates, and is overall a sensible convention to follow.

As we've seen, creating a translation is a two-step process:

1. Abstract the message that needs to be translated by processing it through the `Translator`.
2. Create a translation for the message in each locale that you choose to support.

The second step is done by creating message catalogues that define the translations for any number of different locales.

Message Catalogues

When a message is translated, Symfony2 compiles a message catalogue for the user's locale and looks in it for a translation of the message. A message catalogue is like a dictionary of translations for a specific locale. For example, the catalogue for the `fr_FR` locale might contain the following translation:

Symfony2 is Great => J'aime Symfony2

It's the responsibility of the developer (or translator) of an internationalized application to create these translations. Translations are stored on the filesystem and discovered by Symfony, thanks to some conventions.

Tip: Each time you create a *new* translation resource (or install a bundle that includes a translation resource), be sure to clear your cache so that Symfony can discover the new translation resource:

```
php app/console cache:clear
```

Translation Locations and Naming Conventions

Symfony2 looks for message files (i.e. translations) in two locations:

- For messages found in a bundle, the corresponding message files should live in the `Resources/translations/` directory of the bundle;
- To override any bundle translations, place message files in the `app/Resources/translations` directory.

The filename of the translations is also important as Symfony2 uses a convention to determine details about the translations. Each message file must be named according to the following pattern: `domain.locale.loader`:

- **domain**: An optional way to organize messages into groups (e.g. `admin`, `navigation` or the default `messages`) - see *Using Message Domains*;
- **locale**: The locale that the translations are for (e.g. `en_GB`, `en`, etc);
- **loader**: How Symfony2 should load and parse the file (e.g. `xliff`, `php` or `yml`).

The loader can be the name of any registered loader. By default, Symfony provides the following loaders:

- `xliff`: XLIFF file;
- `php`: PHP file;
- `yml`: YAML file.

The choice of which loader to use is entirely up to you and is a matter of taste.

Note: You can also store translations in a database, or any other storage by providing a custom class implementing the `Symfony\Component\Translation\Loader\LoaderInterface` interface. See `Custom Translation Loaders` below to learn how to register custom loaders.

Creating Translations

The act of creating translation files is an important part of "localization" (often abbreviated L10n). Translation files consist of a series of id-translation pairs for the given domain and locale. The id is the identifier for the individual translation, and can be the message in the main locale (e.g. "Symfony is great") of your application or a unique identifier (e.g. "symfony2.great" - see the sidebar below):

- *XML*

```
<!-- src/Acme/DemoBundle/Resources/translations/messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony2 is great</source>
                <target>J'aime Symfony2</target>
            </trans-unit>
            <trans-unit id="2">
                <source>symfony2.great</source>
                <target>J'aime Symfony2</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
// src/Acme/DemoBundle/Resources/translations/messages.fr.php
return array(
    'Symfony2 is great' => 'J\'aime Symfony2',
    'symfony2.great'    => 'J\'aime Symfony2',
);
```

- *YAML*

```
# src/Acme/DemoBundle/Resources/translations/messages.fr.yml
Symfony2 is great: J'aime Symfony2
symfony2.great:    J'aime Symfony2
```

Symfony2 will discover these files and use them when translating either "Symfony2 is great" or "symfony2.great" into a French language locale (e.g. `fr_FR` or `fr_BE`).

Using Real or Keyword Messages

This example illustrates the two different philosophies when creating messages to be translated:

```
$t = $translator->trans('Symfony2 is great');

$t = $translator->trans('symfony2.great');
```

In the first method, messages are written in the language of the default locale (English in this case). That message is then used as the "id" when creating translations.
In the second method, messages are actually "keywords" that convey the idea of the message. The keyword message is then used as the "id" for any translations. In this case, translations must be made for the default locale (i.e. to translate `symfony2.great` to `Symfony2 is great`).
The second method is handy because the message key won't need to be changed in every translation file if we decide that the message should actually read "Symfony2 is really great" in the default locale.
The choice of which method to use is entirely up to you, but the "keyword" format is often recommended.
Additionally, the `php` and `yaml` file formats support nested ids to avoid repeating yourself if you use keywords instead of real text for your ids:

- *YAML*

```
symfony2:
    is:
        great: Symfony2 is great
        amazing: Symfony2 is amazing
    has:
        bundles: Symfony2 has bundles
user:
    login: Login
```

- *PHP*

```
return array(
    'symfony2' => array(
        'is' => array(
            'great' => 'Symfony2 is great',
            'amazing' => 'Symfony2 is amazing',
        ),
        'has' => array(
            'bundles' => 'Symfony2 has bundles',
        ),
    ),
    'user' => array(
        'login' => 'Login',
    ),
);
```

The multiple levels are flattened into single id/translation pairs by adding a dot (.) between every level, therefore the above examples are equivalent to the following:

- *YAML*

```
symfony2.is.great: Symfony2 is great
symfony2.is.amazing: Symfony2 is amazing
symfony2.has.bundles: Symfony2 has bundles
user.login: Login
```

- *PHP*

```
return array(
    'symfony2.is.great' => 'Symfony2 is great',
    'symfony2.is.amazing' => 'Symfony2 is amazing',
    'symfony2.has.bundles' => 'Symfony2 has bundles',
    'user.login' => 'Login',
);
```

Using Message Domains

As we've seen, message files are organized into the different locales that they translate. The message files can also be organized further into "domains". When creating message files, the domain is the first portion of the filename. The default domain is `messages`. For example, suppose that, for organization, translations were split into three different domains: `messages`, `admin` and `navigation`. The French translation would have the following message files:

- `messages.fr.xliff`
- `admin.fr.xliff`
- `navigation.fr.xliff`

When translating strings that are not in the default domain (`messages`), you must specify the domain as the third argument of `trans()`:

```
$this->get('translator')->trans('Symfony2 is great', array(), 'admin');
```

Symfony2 will now look for the message in the `admin` domain of the user's locale.

Handling the User's Locale

The locale of the current user is stored in the request and is accessible via the `request` object:

```
// access the reqest object in a standard controller
$request = $this->getRequest();

$locale = $request->getLocale();

$request->setLocale('en_US');
```

It is also possible to store the locale in the session instead of on a per request basis. If you do this, each subsequent request will have this locale.

```
$this->get('session')->set('_locale', 'en_US');
```

See the .. _book-translation-locale-url: section below about setting the locale via routing.

Fallback and Default Locale

If the locale hasn't been set explicitly in the session, the `fallback_locale` configuration parameter will be used by the `Translator`. The parameter defaults to `en` (see *Configuration*).

Alternatively, you can guarantee that a locale is set on each user's request by defining a `default_locale` for the framework:

- *YAML*

```
# app/config/config.yml
framework:
    default_locale: en
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:default-locale>en</framework:default-locale>
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    'default_locale' => 'en',
));
```

New in version 2.1: The `default_locale` parameter was defined under the session key originally, however, as of 2.1 this has been moved. This is because the locale is now set on the request instead of the session.

The Locale and the URL

Since you can store the locale of the user is in the session, it may be tempting to use the same URL to display a resource in many different languages based on the user's locale. For example, `http://www.example.com/contact` could show content in English for one user and French for another user. Unfortunately, this violates a fundamental rule of the Web: that a particular URL returns the same resource regardless of the user. To further muddy the problem, which version of the content would be indexed by search engines?

A better policy is to include the locale in the URL. This is fully-supported by the routing system using the special `_locale` parameter:

- *YAML*

```
contact:
    pattern:   /{_locale}/contact
    defaults:  { _controller: AcmeDemoBundle:Contact:index, _locale: en }
    requirements:
        _locale: en|fr|de
```

- *XML*

```
<route id="contact" pattern="/{_locale}/contact">
    <default key="_controller">AcmeDemoBundle:Contact:index</default>
    <default key="_locale">en</default>
    <requirement key="_locale">en|fr|de</requirement>
</route>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('contact', new Route('/{_locale}/contact', array(
    '_controller' => 'AcmeDemoBundle:Contact:index',
    '_locale'     => 'en',
), array(
    '_locale'     => 'en|fr|de'
)));

return $collection;
```

When using the special *_locale* parameter in a route, the matched locale will *automatically be set on the user's session*. In other words, if a user visits the URI `/fr/contact`, the locale `fr` will automatically be set as the locale for the user's session.

You can now use the user's locale to create routes to other translated pages in your application.

Pluralization

Message pluralization is a tough topic as the rules can be quite complex. For instance, here is the mathematic representation of the Russian pluralization rules:

```
(($number % 10 == 1) && ($number % 100 != 11)) ? 0 : ((($number % 10 >= 2) && ($number % 10 <= 4) &&
```

As you can see, in Russian, you can have three different plural forms, each given an index of 0, 1 or 2. For each form, the plural is different, and so the translation is also different.

When a translation has different forms due to pluralization, you can provide all the forms as a string separated by a pipe (`|`):

```
'There is one apple|There are %count% apples'
```

To translate pluralized messages, use the **:method:'Symfony\\Component\\Translation\\Translator::transChoice'** method:

```
$t = $this->get('translator')->transChoice(
    'There is one apple|There are %count% apples',
    10,
    array('%count%' => 10)
);
```

The second argument (`10` in this example), is the *number* of objects being described and is used to determine which translation to use and also to populate the `%count%` placeholder.

Based on the given number, the translator chooses the right plural form. In English, most words have a singular form when there is exactly one object and a plural form for all other numbers (0, 2, 3...). So, if `count` is `1`, the translator will use the first string (`There is one apple`) as the translation. Otherwise it will use `There are %count% apples`.

Here is the French translation:

```
'Il y a %count% pomme|Il y a %count% pommes'
```

Even if the string looks similar (it is made of two sub-strings separated by a pipe), the French rules are different: the first form (no plural) is used when `count` is `0` or `1`. So, the translator will automatically use the first string (`Il y a %count% pomme`) when `count` is `0` or `1`.

Each locale has its own set of rules, with some having as many as six different plural forms with complex rules behind which numbers map to which plural form. The rules are quite simple for English and French, but for Russian, you'd may want a hint to know which rule matches which string. To help translators, you can optionally "tag" each string:

```
'one: There is one apple|some: There are %count% apples'

'none_or_one: Il y a %count% pomme|some: Il y a %count% pommes'
```

The tags are really only hints for translators and don't affect the logic used to determine which plural form to use. The tags can be any descriptive string that ends with a colon (`:`). The tags also do not need to be the same in the original message as in the translated one.

Explicit Interval Pluralization

The easiest way to pluralize a message is to let Symfony2 use internal logic to choose which string to use based on a given number. Sometimes, you'll need more control or want a different translation for specific cases (for `0`, or when the count is negative, for example). For such cases, you can use explicit math intervals:

```
'{0} There are no apples|{1} There is one apple|]1,19] There are %count% apples|[20,Inf] There are ma
```

The intervals follow the ISO 31-11 notation. The above string specifies four different intervals: exactly `0`, exactly `1`, `2-19`, and `20` and higher.

You can also mix explicit math rules and standard rules. In this case, if the count is not matched by a specific interval, the standard rules take effect after removing the explicit rules:

```
'{0} There are no apples|[20,Inf] There are many apples|There is one apple|a_few: There are %count% a
```

For example, for `1` apple, the standard rule `There is one apple` will be used. For `2-19` apples, the second standard rule `There are %count% apples` will be selected.

An `Symfony\Component\Translation\Interval` can represent a finite set of numbers:

```
{1,2,3,4}
```

Or numbers between two other numbers:

```
[1, +Inf[
]-1,2[
```

The left delimiter can be `[` (inclusive) or `]` (exclusive). The right delimiter can be `[` (exclusive) or `]` (inclusive). Beside numbers, you can use `-Inf` and `+Inf` for the infinite.

Translations in Templates

Most of the time, translation occurs in templates. Symfony2 provides native support for both Twig and PHP templates.

Twig Templates

Symfony2 provides specialized Twig tags (`trans` and `transchoice`) to help with message translation of *static blocks of text*:

```
{% trans %}Hello %name%{% endtrans %}

{% transchoice count %}
    {0} There are no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

The `transchoice` tag automatically gets the `%count%` variable from the current context and passes it to the translator. This mechanism only works when you use a placeholder following the `%var%` pattern.

Tip: If you need to use the percent character (`%`) in a string, escape it by doubling it: `{% trans %}Percent: %percent%%%{% endtrans %}`

You can also specify the message domain and pass some additional variables:

```
{% trans with {'%name%': 'Fabien'} from "app" %}Hello %name%{% endtrans %}

{% trans with {'%name%': 'Fabien'} from "app" into "fr" %}Hello %name%{% endtrans %}

{% transchoice count with {'%name%': 'Fabien'} from "app" %}
    {0} There is no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

The `trans` and `transchoice` filters can be used to translate *variable texts* and complex expressions:

```
{{ message|trans }}

{{ message|transchoice(5) }}

{{ message|trans({'%name%': 'Fabien'}, "app") }}

{{ message|transchoice(5, {'%name%': 'Fabien'}, 'app') }}
```

Tip: Using the translation tags or filters have the same effect, but with one subtle difference: automatic output escaping is only applied to variables translated using a filter. In other words, if you need to be sure that your translated variable is *not* output escaped, you must apply the raw filter after the translation filter:

```
{# text translated between tags is never escaped #}
{% trans %}
    <h3>foo</h3>
{% endtrans %}

{% set message = '<h3>foo</h3>' %}

{# a variable translated via a filter is escaped by default #}
{{ message|trans|raw }}

{# but static strings are never escaped #}
{{ '<h3>foo</h3>'|trans }}
```

PHP Templates

The translator service is accessible in PHP templates through the `translator` helper:

```
<?php echo $view['translator']->trans('Symfony2 is great') ?>

<?php echo $view['translator']->transChoice(
    '{0} There is no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10)
) ?>
```

Forcing the Translator Locale

When translating a message, Symfony2 uses the locale from the current request or the `fallback` locale if necessary. You can also manually specify the locale to use for translation:

```
$this->get('translator')->trans(
    'Symfony2 is great',
    array(),
    'messages',
    'fr_FR',
);

$this->get('translator')->trans(
    '{0} There are no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10),
    'messages',
```

```
    'fr_FR',
);
```

Translating Database Content

The translation of database content should be handled by Doctrine through the Translatable Extension. For more information, see the documentation for that library.

Summary

With the Symfony2 Translation component, creating an internationalized application no longer needs to be a painful process and boils down to just a few basic steps:

- Abstract messages in your application by wrapping each in either the **:method:‘Symfony\\Component\\Translation\\Translator::trans‘** or **:method:‘Symfony\\Component\\Translation\\Translat** methods;
- Translate each message into multiple locales by creating translation message files. Symfony2 discovers and processes each file because its name follows a specific convention;
- Manage the user’s locale, which is stored on the request, but can also be set once the user’s session.

2.1.15 Service Container

A modern PHP application is full of objects. One object may facilitate the delivery of email messages while another may allow you to persist information into a database. In your application, you may create an object that manages your product inventory, or another object that processes data from a third-party API. The point is that a modern application does many things and is organized into many objects that handle each task.

In this chapter, we’ll talk about a special PHP object in Symfony2 that helps you instantiate, organize and retrieve the many objects of your application. This object, called a service container, will allow you to standardize and centralize the way objects are constructed in your application. The container makes your life easier, is super fast, and emphasizes an architecture that promotes reusable and decoupled code. And since all core Symfony2 classes use the container, you’ll learn how to extend, configure and use any object in Symfony2. In large part, the service container is the biggest contributor to the speed and extensibility of Symfony2.

Finally, configuring and using the service container is easy. By the end of this chapter, you’ll be comfortable creating your own objects via the container and customizing objects from any third-party bundle. You’ll begin writing code that is more reusable, testable and decoupled, simply because the service container makes writing good code so easy.

What is a Service?

Put simply, a Service is any PHP object that performs some sort of “global” task. It’s a purposefully-generic name used in computer science to describe an object that’s created for a specific purpose (e.g. delivering emails). Each service is used throughout your application whenever you need the specific functionality it provides. You don’t have to do anything special to make a service: simply write a PHP class with some code that accomplishes a specific task. Congratulations, you’ve just created a service!

Note: As a rule, a PHP object is a service if it is used globally in your application. A single `Mailer` service is used globally to send email messages whereas the many `Message` objects that it delivers are *not* services. Similarly, a `Product` object is not a service, but an object that persists `Product` objects to a database *is* a service.

So what's the big deal then? The advantage of thinking about "services" is that you begin to think about separating each piece of functionality in your application into a series of services. Since each service does just one job, you can easily access each service and use its functionality wherever you need it. Each service can also be more easily tested and configured since it's separated from the other functionality in your application. This idea is called service-oriented architecture and is not unique to Symfony2 or even PHP. Structuring your application around a set of independent service classes is a well-known and trusted object-oriented best-practice. These skills are key to being a good developer in almost any language.

What is a Service Container?

A Service Container (or *dependency injection container*) is simply a PHP object that manages the instantiation of services (i.e. objects). For example, suppose we have a simple PHP class that delivers email messages. Without a service container, we must manually create the object whenever we need it:

```
use Acme\HelloBundle\Mailer;

$mailer = new Mailer('sendmail');
$mailer->send('ryan@foobar.net', ... );
```

This is easy enough. The imaginary `Mailer` class allows us to configure the method used to deliver the email messages (e.g. `sendmail`, `smtp`, etc). But what if we wanted to use the mailer service somewhere else? We certainly don't want to repeat the mailer configuration *every* time we need to use the `Mailer` object. What if we needed to change the `transport` from `sendmail` to `smtp` everywhere in the application? We'd need to hunt down every place we create a `Mailer` service and change it.

Creating/Configuring Services in the Container

A better answer is to let the service container create the `Mailer` object for you. In order for this to work, we must *teach* the container how to create the `Mailer` service. This is done via configuration, which can be specified in YAML, XML or PHP:

- *YAML*

```
# app/config/config.yml
services:
    my_mailer:
        class:        Acme\HelloBundle\Mailer
        arguments:    [sendmail]
```

- *XML*

```
<!-- app/config/config.xml -->
<services>
    <service id="my_mailer" class="Acme\HelloBundle\Mailer">
        <argument>sendmail</argument>
    </service>
</services>
```

- *PHP*

```
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$container->setDefinition('my_mailer', new Definition(
    'Acme\HelloBundle\Mailer',
    array('sendmail')
));
```

Note: When Symfony2 initializes, it builds the service container using the application configuration (`app/config/config.yml` by default). The exact file that's loaded is dictated by the `AppKernel::registerContainerConfiguration()` method, which loads an environment-specific configuration file (e.g. `config_dev.yml` for the `dev` environment or `config_prod.yml` for `prod`).

An instance of the `Acme\HelloBundle\Mailer` object is now available via the service container. The container is available in any traditional Symfony2 controller where you can access the services of the container via the `get()` shortcut method:

```
class HelloController extends Controller
{
    // ...

    public function sendEmailAction()
    {
        // ...
        $mailer = $this->get('my_mailer');
        $mailer->send('ryan@foobar.net', ... );
    }
}
```

When we ask for the `my_mailer` service from the container, the container constructs the object and returns it. This is another major advantage of using the service container. Namely, a service is *never* constructed until it's needed. If you define a service and never use it on a request, the service is never created. This saves memory and increases the speed of your application. This also means that there's very little or no performance hit for defining lots of services. Services that are never used are never constructed.

As an added bonus, the `Mailer` service is only created once and the same instance is returned each time you ask for the service. This is almost always the behavior you'll need (it's more flexible and powerful), but we'll learn later how you can configure a service that has multiple instances.

Service Parameters

The creation of new services (i.e. objects) via the container is pretty straightforward. Parameters make defining services more organized and flexible:

- *YAML*

```
# app/config/config.yml
parameters:
    my_mailer.class:      Acme\HelloBundle\Mailer
    my_mailer.transport:  sendmail

services:
    my_mailer:
        class:        %my_mailer.class%
        arguments:    [%my_mailer.transport%]
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
    <parameter key="my_mailer.class">Acme\HelloBundle\Mailer</parameter>
    <parameter key="my_mailer.transport">sendmail</parameter>
</parameters>

<services>
```

```
        <service id="my_mailer" class="%my_mailer.class%">
            <argument>%my_mailer.transport%</argument>
        </service>
    </services>
```

- *PHP*

```
    // app/config/config.php
    use Symfony\Component\DependencyInjection\Definition;

    $container->setParameter('my_mailer.class', 'Acme\HelloBundle\Mailer');
    $container->setParameter('my_mailer.transport', 'sendmail');

    $container->setDefinition('my_mailer', new Definition(
        '%my_mailer.class%',
        array('%my_mailer.transport%')
    ));
```

The end result is exactly the same as before - the difference is only in *how* we defined the service. By surrounding the `my_mailer.class` and `my_mailer.transport` strings in percent (`%`) signs, the container knows to look for parameters with those names. When the container is built, it looks up the value of each parameter and uses it in the service definition.

The purpose of parameters is to feed information into services. Of course there was nothing wrong with defining the service without using any parameters. Parameters, however, have several advantages:

- separation and organization of all service "options" under a single `parameters` key;
- parameter values can be used in multiple service definitions;
- when creating a service in a bundle (we'll show this shortly), using parameters allows the service to be easily customized in your application.

The choice of using or not using parameters is up to you. High-quality third-party bundles will *always* use parameters as they make the service stored in the container more configurable. For the services in your application, however, you may not need the flexibility of parameters.

Array Parameters

Parameters do not need to be flat strings, they can also be arrays. For the XML format, you need to use the type="collection" attribute for all parameters that are arrays.

- *YAML*

```
    # app/config/config.yml
    parameters:
        my_mailer.gateways:
            - mail1
            - mail2
            - mail3
        my_multilang.language_fallback:
            en:
                - en
                - fr
            fr:
                - fr
                - en
```

- *XML*

```xml
<!-- app/config/config.xml -->
<parameters>
    <parameter key="my_mailer.gateways" type="collection">
        <parameter>mail1</parameter>
        <parameter>mail2</parameter>
        <parameter>mail3</parameter>
    </parameter>
    <parameter key="my_multilang.language_fallback" type="collection">
        <parameter key="en" type="collection">
            <parameter>en</parameter>
            <parameter>fr</parameter>
        </parameter>
        <parameter key="fr" type="collection">
            <parameter>fr</parameter>
            <parameter>en</parameter>
        </parameter>
    </parameter>
</parameters>
```

- *PHP*

```php
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('my_mailer.gateways', array('mail1', 'mail2', 'mail3'));
$container->setParameter('my_multilang.language_fallback',
                         array('en' => array('en', 'fr'),
                               'fr' => array('fr', 'en'),
                         ));
```

Importing other Container Configuration Resources

Tip: In this section, we'll refer to service configuration files as *resources*. This is to highlight that fact that, while most configuration resources will be files (e.g. YAML, XML, PHP), Symfony2 is so flexible that configuration could be loaded from anywhere (e.g. a database or even via an external web service).

The service container is built using a single configuration resource (`app/config/config.yml` by default). All other service configuration (including the core Symfony2 and third-party bundle configuration) must be imported from inside this file in one way or another. This gives you absolute flexibility over the services in your application.

External service configuration can be imported in two different ways. First, we'll talk about the method that you'll use most commonly in your application: the `imports` directive. In the following section, we'll introduce the second method, which is the flexible and preferred method for importing service configuration from third-party bundles.

Importing Configuration with `imports`

So far, we've placed our `my_mailer` service container definition directly in the application configuration file (e.g. `app/config/config.yml`). Of course, since the `Mailer` class itself lives inside the `AcmeHelloBundle`, it makes more sense to put the `my_mailer` container definition inside the bundle as well.

First, move the `my_mailer` container definition into a new container resource file inside `AcmeHelloBundle`. If the `Resources` or `Resources/config` directories don't exist, create them.

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    my_mailer.class:      Acme\HelloBundle\Mailer
    my_mailer.transport:  sendmail

services:
    my_mailer:
        class:        %my_mailer.class%
        arguments:    [%my_mailer.transport%]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <parameter key="my_mailer.class">Acme\HelloBundle\Mailer</parameter>
    <parameter key="my_mailer.transport">sendmail</parameter>
</parameters>

<services>
    <service id="my_mailer" class="%my_mailer.class%">
        <argument>%my_mailer.transport%</argument>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('my_mailer.class', 'Acme\HelloBundle\Mailer');
$container->setParameter('my_mailer.transport', 'sendmail');

$container->setDefinition('my_mailer', new Definition(
    '%my_mailer.class%',
    array('%my_mailer.transport%')
));
```

The definition itself hasn't changed, only its location. Of course the service container doesn't know about the new resource file. Fortunately, we can easily import the resource file using the `imports` key in the application configuration.

- *YAML*

```
# app/config/config.yml
imports:
    - { resource: @AcmeHelloBundle/Resources/config/services.yml }
```

- *XML*

```
<!-- app/config/config.xml -->
<imports>
    <import resource="@AcmeHelloBundle/Resources/config/services.xml"/>
</imports>
```

- *PHP*

```
// app/config/config.php
$this->import('@AcmeHelloBundle/Resources/config/services.php');
```

The `imports` directive allows your application to include service container configuration resources from any other

location (most commonly from bundles). The `resource` location, for files, is the absolute path to the resource file. The special `@AcmeHello` syntax resolves the directory path of the `AcmeHelloBundle` bundle. This helps you specify the path to the resource without worrying later if you move the `AcmeHelloBundle` to a different directory.

Importing Configuration via Container Extensions

When developing in Symfony2, you'll most commonly use the `imports` directive to import container configuration from the bundles you've created specifically for your application. Third-party bundle container configuration, including Symfony2 core services, are usually loaded using another method that's more flexible and easy to configure in your application.

Here's how it works. Internally, each bundle defines its services very much like we've seen so far. Namely, a bundle uses one or more configuration resource files (usually XML) to specify the parameters and services for that bundle. However, instead of importing each of these resources directly from your application configuration using the `imports` directive, you can simply invoke a *service container extension* inside the bundle that does the work for you. A service container extension is a PHP class created by the bundle author to accomplish two things:

- import all service container resources needed to configure the services for the bundle;
- provide semantic, straightforward configuration so that the bundle can be configured without interacting with the flat parameters of the bundle's service container configuration.

In other words, a service container extension configures the services for a bundle on your behalf. And as we'll see in a moment, the extension provides a sensible, high-level interface for configuring the bundle.

Take the `FrameworkBundle` - the core Symfony2 framework bundle - as an example. The presence of the following code in your application configuration invokes the service container extension inside the `FrameworkBundle`:

- *YAML*

```yaml
# app/config/config.yml
framework:
    secret:          xxxxxxxxxx
    charset:         UTF-8
    form:            true
    csrf_protection: true
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
    # ...
```

- *XML*

```xml
<!-- app/config/config.xml -->
<framework:config charset="UTF-8" secret="xxxxxxxxxx">
    <framework:form />
    <framework:csrf-protection />
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
    <!-- ... -->
</framework:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('framework', array(
    'secret'          => 'xxxxxxxxxx',
    'charset'         => 'UTF-8',
    'form'            => array(),
    'csrf-protection' => array(),
    'router'          => array('resource' => '%kernel.root_dir%/config/routing.php'),
```

```
        // ...
    ));
```

When the configuration is parsed, the container looks for an extension that can handle the `framework` configuration directive. The extension in question, which lives in the `FrameworkBundle`, is invoked and the service configuration for the `FrameworkBundle` is loaded. If you remove the `framework` key from your application configuration file entirely, the core Symfony2 services won't be loaded. The point is that you're in control: the Symfony2 framework doesn't contain any magic or perform any actions that you don't have control over.

Of course you can do much more than simply "activate" the service container extension of the `FrameworkBundle`. Each extension allows you to easily customize the bundle, without worrying about how the internal services are defined.

In this case, the extension allows you to customize the `charset`, `error_handler`, `csrf_protection`, `router` configuration and much more. Internally, the `FrameworkBundle` uses the options specified here to define and configure the services specific to it. The bundle takes care of creating all the necessary `parameters` and `services` for the service container, while still allowing much of the configuration to be easily customized. As an added bonus, most service container extensions are also smart enough to perform validation - notifying you of options that are missing or the wrong data type.

When installing or configuring a bundle, see the bundle's documentation for how the services for the bundle should be installed and configured. The options available for the core bundles can be found inside the Reference Guide.

Note: Natively, the service container only recognizes the `parameters`, `services`, and `imports` directives. Any other directives are handled by a service container extension.

Referencing (Injecting) Services

So far, our original `my_mailer` service is simple: it takes just one argument in its constructor, which is easily configurable. As you'll see, the real power of the container is realized when you need to create a service that depends on one or more other services in the container.

Let's start with an example. Suppose we have a new service, `NewsletterManager`, that helps to manage the preparation and delivery of an email message to a collection of addresses. Of course the `my_mailer` service is already really good at delivering email messages, so we'll use it inside `NewsletterManager` to handle the actual delivery of the messages. This pretend class might look something like this:

```
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Without using the service container, we can create a new `NewsletterManager` fairly easily from inside a controller:

```
public function sendNewsletterAction()
{
    $mailer = $this->get('my_mailer');
    $newsletter = new Acme\HelloBundle\Newsletter\NewsletterManager($mailer);
    // ...
}
```

This approach is fine, but what if we decide later that the `NewsletterManager` class needs a second or third constructor argument? What if we decide to refactor our code and rename the class? In both cases, you'd need to find every place where the `NewsletterManager` is instantiated and modify it. Of course, the service container gives us a much more appealing option:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

services:
    my_mailer:
        # ...
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@my_mailer]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager</par
</parameters>

<services>
    <service id="my_mailer" ... >
      <!-- ... -->
    </service>
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <argument type="service" id="my_mailer"/>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana

$container->setDefinition('my_mailer', ... );
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%',
    array(new Reference('my_mailer'))
));
```

In YAML, the special `@my_mailer` syntax tells the container to look for a service named `my_mailer` and to pass that object into the constructor of `NewsletterManager`. In this case, however, the specified service `my_mailer`

must exist. If it does not, an exception will be thrown. You can mark your dependencies as optional - this will be discussed in the next section.

Using references is a very powerful tool that allows you to create independent service classes with well-defined dependencies. In this example, the `newsletter_manager` service needs the `my_mailer` service in order to function. When you define this dependency in the service container, the container takes care of all the work of instantiating the objects.

Optional Dependencies: Setter Injection

Injecting dependencies into the constructor in this manner is an excellent way of ensuring that the dependency is available to use. If you have optional dependencies for a class, then "setter injection" may be a better option. This means injecting the dependency using a method call rather than through the constructor. The class would look like this:

```
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Injecting the dependency by the setter method just needs a change of syntax:

- *YAML*

```
    # src/Acme/HelloBundle/Resources/config/services.yml
    parameters:
        # ...
        newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

    services:
        my_mailer:
            # ...
        newsletter_manager:
            class:     %newsletter_manager.class%
            calls:
                - [ setMailer, [ @my_mailer ] ]
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/services.xml -->
    <parameters>
        <!-- ... -->
        <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager</par
    </parameters>

    <services>
        <service id="my_mailer" ... >
```

```
        <!-- ... -->
    </service>
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <call method="setMailer">
            <argument type="service" id="my_mailer" />
        </call>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana

$container->setDefinition('my_mailer', ... );
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%'
))->addMethodCall('setMailer', array(
    new Reference('my_mailer')
));
```

Note: The approaches presented in this section are called "constructor injection" and "setter injection". The Symfony2 service container also supports "property injection".

Making References Optional

Sometimes, one of your services may have an optional dependency, meaning that the dependency is not required for your service to work properly. In the example above, the `my_mailer` service *must* exist, otherwise an exception will be thrown. By modifying the `newsletter_manager` service definition, you can make this reference optional. The container will then inject it if it exists and do nothing if it doesn't:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...

services:
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@?my_mailer]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->

<services>
    <service id="my_mailer" ... >
      <!-- ... -->
    </service>
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <argument type="service" id="my_mailer" on-invalid="ignore" />
```

```
        </service>
    </services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;
    use Symfony\Component\DependencyInjection\ContainerInterface;

    // ...
    $container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana

    $container->setDefinition('my_mailer', ... );
    $container->setDefinition('newsletter_manager', new Definition(
        '%newsletter_manager.class%',
        array(new Reference('my_mailer', ContainerInterface::IGNORE_ON_INVALID_REFERENCE))
    ));
```

In YAML, the special `@?` syntax tells the service container that the dependency is optional. Of course, the `NewsletterManager` must also be written to allow for an optional dependency:

```
public function __construct(Mailer $mailer = null)
{
    // ...
}
```

Core Symfony and Third-Party Bundle Services

Since Symfony2 and all third-party bundles configure and retrieve their services via the container, you can easily access them or even use them in your own services. To keep things simple, Symfony2 by default does not require that controllers be defined as services. Furthermore Symfony2 injects the entire service container into your controller. For example, to handle the storage of information on a user's session, Symfony2 provides a `session` service, which you can access inside a standard controller as follows:

```
public function indexAction($bar)
{
    $session = $this->get('session');
    $session->set('foo', $bar);

    // ...
}
```

In Symfony2, you'll constantly use services provided by the Symfony core or other third-party bundles to perform tasks such as rendering templates (`templating`), sending emails (`mailer`), or accessing information on the request (`request`).

We can take this a step further by using these services inside services that you've created for your application. Let's modify the `NewsletterManager` to use the real Symfony2 `mailer` service (instead of the pretend `my_mailer`). Let's also pass the templating engine service to the `NewsletterManager` so that it can generate the email content via a template:

```
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\Templating\EngineInterface;

class NewsletterManager
```

```
{
    protected $mailer;

    protected $templating;

    public function __construct(\Swift_Mailer $mailer, EngineInterface $templating)
    {
        $this->mailer = $mailer;
        $this->templating = $templating;
    }

    // ...
}
```

Configuring the service container is easy:

- *YAML*

```
    services:
        newsletter_manager:
            class:     %newsletter_manager.class%
            arguments: [@mailer, @templating]
```

- *XML*

```
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <argument type="service" id="mailer"/>
        <argument type="service" id="templating"/>
    </service>
```

- *PHP*

```
    $container->setDefinition('newsletter_manager', new Definition(
        '%newsletter_manager.class%',
        array(
            new Reference('mailer'),
            new Reference('templating')
        )
    ));
```

The `newsletter_manager` service now has access to the core `mailer` and `templating` services. This is a common way to create services specific to your application that leverage the power of different services within the framework.

Tip: Be sure that `swiftmailer` entry appears in your application configuration. As we mentioned in *Importing Configuration via Container Extensions*, the `swiftmailer` key invokes the service extension from the `SwiftmailerBundle`, which registers the `mailer` service.

Advanced Container Configuration

As we've seen, defining services inside the container is easy, generally involving a `service` configuration key and a few parameters. However, the container has several other tools available that help to *tag* services for special functionality, create more complex services, and perform operations after the container is built.

Marking Services as public / private

When defining services, you'll usually want to be able to access these definitions within your application code. These services are called `public`. For example, the `doctrine` service registered with the container when using the DoctrineBundle is a public service as you can access it via:

```
$doctrine = $container->get('doctrine');
```

However, there are use-cases when you don't want a service to be public. This is common when a service is only defined because it could be used as an argument for another service.

Note: If you use a private service as an argument to more than one other service, this will result in two different instances being used as the instantiation of the private service is done inline (e.g. `new PrivateFooBar()`).

Simply said: A service will be private when you do not want to access it directly from your code.

Here is an example:

- *YAML*

```
services:
   foo:
     class: Acme\HelloBundle\Foo
     public: false
```

- *XML*

```
<service id="foo" class="Acme\HelloBundle\Foo" public="false" />
```

- *PHP*

```
$definition = new Definition('Acme\HelloBundle\Foo');
$definition->setPublic(false);
$container->setDefinition('foo', $definition);
```

Now that the service is private, you *cannot* call:

```
$container->get('foo');
```

However, if a service has been marked as private, you can still alias it (see below) to access this service (via the alias).

Note: Services are by default public.

Aliasing

When using core or third party bundles within your application, you may want to use shortcuts to access some services. You can do so by aliasing them and, furthermore, you can even alias non-public services.

- *YAML*

```
services:
   foo:
     class: Acme\HelloBundle\Foo
   bar:
     alias: foo
```

- *XML*

```
    <service id="foo" class="Acme\HelloBundle\Foo"/>

    <service id="bar" alias="foo" />
```

- *PHP*

```
    $definition = new Definition('Acme\HelloBundle\Foo');
    $container->setDefinition('foo', $definition);

    $containerBuilder->setAlias('bar', 'foo');
```

This means that when using the container directly, you can access the `foo` service by asking for the `bar` service like this:

```
$container->get('bar'); // Would return the foo service
```

Requiring files

There might be use cases when you need to include another file just before the service itself gets loaded. To do so, you can use the `file` directive.

- *YAML*

```
    services:
       foo:
         class: Acme\HelloBundle\Foo\Bar
         file: %kernel.root_dir%/src/path/to/file/foo.php
```

- *XML*

```
    <service id="foo" class="Acme\HelloBundle\Foo\Bar">
        <file>%kernel.root_dir%/src/path/to/file/foo.php</file>
    </service>
```

- *PHP*

```
    $definition = new Definition('Acme\HelloBundle\Foo\Bar');
    $definition->setFile('%kernel.root_dir%/src/path/to/file/foo.php');
    $container->setDefinition('foo', $definition);
```

Notice that symfony will internally call the PHP function require_once which means that your file will be included only once per request.

Tags (`tags`)

In the same way that a blog post on the Web might be tagged with things such as "Symfony" or "PHP", services configured in your container can also be tagged. In the service container, a tag implies that the service is meant to be used for a specific purpose. Take the following example:

- *YAML*

```
    services:
        foo.twig.extension:
            class: Acme\HelloBundle\Extension\FooExtension
            tags:
                -  { name: twig.extension }
```

- *XML*

```
<service id="foo.twig.extension" class="Acme\HelloBundle\Extension\FooExtension">
    <tag name="twig.extension" />
</service>
```

- *PHP*

```
$definition = new Definition('Acme\HelloBundle\Extension\FooExtension');
$definition->addTag('twig.extension');
$container->setDefinition('foo.twig.extension', $definition);
```

The `twig.extension` tag is a special tag that the `TwigBundle` uses during configuration. By giving the service this `twig.extension` tag, the bundle knows that the `foo.twig.extension` service should be registered as a Twig extension with Twig. In other words, Twig finds all services tagged with `twig.extension` and automatically registers them as extensions.

Tags, then, are a way to tell Symfony2 or other third-party bundles that your service should be registered or used in some special way by the bundle.

The following is a list of tags available with the core Symfony2 bundles. Each of these has a different effect on your service and many tags require additional arguments (beyond just the `name` parameter).

- assetic.filter
- assetic.templating.php
- data_collector
- form.field_factory.guesser
- kernel.cache_warmer
- kernel.event_listener
- monolog.logger
- routing.loader
- security.listener.factory
- security.voter
- templating.helper
- twig.extension
- translation.loader
- validator.constraint_validator

Learn more from the Cookbook

- How to Use a Factory to Create Services
- How to Manage Common Dependencies with Parent Services
- How to define Controllers as Services

2.1.16 Performance

Symfony2 is fast, right out of the box. Of course, if you really need speed, there are many ways that you can make Symfony even faster. In this chapter, you'll explore many of the most common and powerful ways to make your Symfony application even faster.

Use a Byte Code Cache (e.g. APC)

One the best (and easiest) things that you should do to improve your performance is to use a "byte code cache". The idea of a byte code cache is to remove the need to constantly recompile the PHP source code. There are a number of byte code caches available, some of which are open source. The most widely used byte code cache is probably APC

Using a byte code cache really has no downside, and Symfony2 has been architected to perform really well in this type of environment.

Further Optimizations

Byte code caches usually monitor the source files for changes. This ensures that if the source of a file changes, the byte code is recompiled automatically. This is really convenient, but obviously adds overhead.

For this reason, some byte code caches offer an option to disable these checks. Obviously, when disabling these checks, it will be up to the server admin to ensure that the cache is cleared whenever any source files change. Otherwise, the updates you've made won't be seen.

For example, to disable these checks in APC, simply add `apc.stat=0` to your php.ini configuration.

Use an Autoloader that caches (e.g. `ApcUniversalClassLoader`)

By default, the Symfony2 standard edition uses the `UniversalClassLoader` in the autoloader.php file. This autoloader is easy to use, as it will automatically find any new classes that you've placed in the registered directories.

Unfortunately, this comes at a cost, as the loader iterates over all configured namespaces to find a particular file, making `file_exists` calls until it finally finds the file it's looking for.

The simplest solution is to cache the location of each class after it's located the first time. Symfony comes with a class - `ApcUniversalClassLoader` - loader that extends the `UniversalClassLoader` and stores the class locations in APC.

To use this class loader, simply adapt your `autoloader.php` as follows:

```
// app/autoload.php
require __DIR__.'/../vendor/symfony/src/Symfony/Component/ClassLoader/ApcUniversalClassLoader.php';

use Symfony\Component\ClassLoader\ApcUniversalClassLoader;

$loader = new ApcUniversalClassLoader('some caching unique prefix');
// ...
```

Note: When using the APC autoloader, if you add new classes, they will be found automatically and everything will work the same as before (i.e. no reason to "clear" the cache). However, if you change the location of a particular namespace or prefix, you'll need to flush your APC cache. Otherwise, the autoloader will still be looking at the old location for all classes inside that namespace.

Use Bootstrap Files

To ensure optimal flexibility and code reuse, Symfony2 applications leverage a variety of classes and 3rd party components. But loading all of these classes from separate files on each request can result in some overhead. To reduce this overhead, the Symfony2 Standard Edition provides a script to generate a so-called bootstrap file, consisting of multiple classes definitions in a single file. By including this file (which contains a copy of many of the core classes), Symfony no longer needs to include any of the source files containing those classes. This will reduce disc IO quite a bit.

If you're using the Symfony2 Standard Edition, then you're probably already using the bootstrap file. To be sure, open your front controller (usually `app.php`) and check to make sure that the following line exists:

```
require_once __DIR__.'/../app/bootstrap.php.cache';
```

Note that there are two disadvantages when using a bootstrap file:

- the file needs to be regenerated whenever any of the original sources change (i.e. when you update the Symfony2 source or vendor libraries);
- when debugging, one will need to place break points inside the bootstrap file.

If you're using Symfony2 Standard Edition, the bootstrap file is automatically rebuilt after updating the vendor libraries via the `php bin/vendors install` command.

Bootstrap Files and Byte Code Caches

Even when using a byte code cache, performance will improve when using a bootstrap file since there will be less files to monitor for changes. Of course if this feature is disabled in the byte code cache (e.g. `apc.stat=0` in APC), there is no longer a reason to use a bootstrap file.

2.1.17 Internals

Looks like you want to understand how Symfony2 works and how to extend it. That makes me very happy! This section is an in-depth explanation of the Symfony2 internals.

Note: You need to read this section only if you want to understand how Symfony2 works behind the scene, or if you want to extend Symfony2.

Overview

The Symfony2 code is made of several independent layers. Each layer is built on top of the previous one.

Tip: Autoloading is not managed by the framework directly; it's done independently with the help of the `Symfony\Component\ClassLoader\UniversalClassLoader` class and the `src/autoload.php` file. Read the dedicated chapter for more information.

`HttpFoundation` Component

The deepest level is the **:namespace:'Symfony\\Component\\HttpFoundation'** component. HttpFoundation provides the main objects needed to deal with HTTP. It is an Object-Oriented abstraction of some native PHP functions and variables:

- The `Symfony\Component\HttpFoundation\Request` class abstracts the main PHP global variables like `$_GET`, `$_POST`, `$_COOKIE`, `$_FILES`, and `$_SERVER`;
- The `Symfony\Component\HttpFoundation\Response` class abstracts some PHP functions like `header()`, `setcookie()`, and `echo`;
- The `Symfony\Component\HttpFoundation\Session` class and `Symfony\Component\HttpFoundation\SessionStorage\SessionStorageInterface` interface abstract session management `session_*()` functions.

`HttpKernel` Component

On top of HttpFoundation is the **:namespace:'Symfony\\Component\\HttpKernel'** component. HttpKernel handles the dynamic part of HTTP; it is a thin wrapper on top of the Request and Response classes to standardize the way requests are handled. It also provides extension points and tools that makes it the ideal starting point to create a Web framework without too much overhead.

It also optionally adds configurability and extensibility, thanks to the Dependency Injection component and a powerful plugin system (bundles).

See also:

Read more about the `HttpKernel` component. Read more about Dependency Injection and Bundles.

`FrameworkBundle` Bundle

The **:namespace:'Symfony\\Bundle\\FrameworkBundle'** bundle is the bundle that ties the main components and libraries together to make a lightweight and fast MVC framework. It comes with a sensible default configuration and conventions to ease the learning curve.

Kernel

The `Symfony\Component\HttpKernel\HttpKernel` class is the central class of Symfony2 and is responsible for handling client requests. Its main goal is to "convert" a `Symfony\Component\HttpFoundation\Request` object to a `Symfony\Component\HttpFoundation\Response` object.

Every Symfony2 Kernel implements `Symfony\Component\HttpKernel\HttpKernelInterface`:

```
function handle(Request $request, $type = self::MASTER_REQUEST, $catch = true)
```

Controllers

To convert a Request to a Response, the Kernel relies on a "Controller". A Controller can be any valid PHP callable.

The Kernel delegates the selection of what Controller should be executed to an implementation of `Symfony\Component\HttpKernel\Controller\ControllerResolverInterface`:

```
public function getController(Request $request);

public function getArguments(Request $request, $controller);
```

The **:method:'Symfony\\Component\\HttpKernel\\Controller\\ControllerResolverInterface::getController'** method returns the Controller (a PHP callable) associated with the given Request. The default implementation (`Symfony\Component\HttpKernel\Controller\ControllerResolver`) looks for

a `_controller` request attribute that represents the controller name (a "class::method" string, like `Bundle\BlogBundle\PostController:indexAction`).

Tip: The default implementation uses the `Symfony\Bundle\FrameworkBundle\EventListener\RouterListener` to define the `_controller` Request attribute (see *kernel.request Event*).

The **:method:'Symfony\\Component\\HttpKernel\\Controller\\ControllerResolverInterface::getArguments'** method returns an array of arguments to pass to the Controller callable. The default implementation automatically resolves the method arguments, based on the Request attributes.

Matching Controller method arguments from Request attributes

For each method argument, Symfony2 tries to get the value of a Request attribute with the same name. If it is not defined, the argument default value is used if defined:

```
// Symfony2 will look for an 'id' attribute (mandatory)
// and an 'admin' one (optional)
public function showAction($id, $admin = true)
{
    // ...
}
```

Handling Requests

The `handle()` method takes a `Request` and *always* returns a `Response`. To convert the `Request`, `handle()` relies on the Resolver and an ordered chain of Event notifications (see the next section for more information about each Event):

1. Before doing anything else, the `kernel.request` event is notified – if one of the listeners returns a `Response`, it jumps to step 8 directly;
2. The Resolver is called to determine the Controller to execute;
3. Listeners of the `kernel.controller` event can now manipulate the Controller callable the way they want (change it, wrap it, ...);
4. The Kernel checks that the Controller is actually a valid PHP callable;
5. The Resolver is called to determine the arguments to pass to the Controller;
6. The Kernel calls the Controller;
7. If the Controller does not return a `Response`, listeners of the `kernel.view` event can convert the Controller return value to a `Response`;
8. Listeners of the `kernel.response` event can manipulate the `Response` (content and headers);
9. The Response is returned.

If an Exception is thrown during processing, the `kernel.exception` is notified and listeners are given a chance to convert the Exception to a Response. If that works, the `kernel.response` event is notified; if not, the Exception is re-thrown.

If you don't want Exceptions to be caught (for embedded requests for instance), disable the `kernel.exception` event by passing `false` as the third argument to the `handle()` method.

Internal Requests

At any time during the handling of a request (the 'master' one), a sub-request can be handled. You can pass the request type to the `handle()` method (its second argument):

- `HttpKernelInterface::MASTER_REQUEST`;
- `HttpKernelInterface::SUB_REQUEST`.

The type is passed to all events and listeners can act accordingly (some processing must only occur on the master request).

Events

Each event thrown by the Kernel is a subclass of `Symfony\Component\HttpKernel\Event\KernelEvent`. This means that each event has access to the same basic information:

- `getRequestType()` - returns the *type* of the request (`HttpKernelInterface::MASTER_REQUEST` or `HttpKernelInterface::SUB_REQUEST`);
- `getKernel()` - returns the Kernel handling the request;
- `getRequest()` - returns the current `Request` being handled.

`getRequestType()` The `getRequestType()` method allows listeners to know the type of the request. For instance, if a listener must only be active for master requests, add the following code at the beginning of your listener method:

```
use Symfony\Component\HttpKernel\HttpKernelInterface;

if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
    // return immediately
    return;
}
```

Tip: If you are not yet familiar with the Symfony2 Event Dispatcher, read the *Events* section first.

`kernel.request` Event *Event Class*: `Symfony\Component\HttpKernel\Event\GetResponseEvent`

The goal of this event is to either return a `Response` object immediately or setup variables so that a Controller can be called after the event. Any listener can return a `Response` object via the `setResponse()` method on the event. In this case, all other listeners won't be called.

This event is used by `FrameworkBundle` to populate the `_controller Request` attribute, via the `Symfony\Bundle\FrameworkBundle\EventListener\RouterListener`. RequestListener uses a `Symfony\Component\Routing\RouterInterface` object to match the `Request` and determine the Controller name (stored in the `_controller Request` attribute).

`kernel.controller` Event *Event Class*: `Symfony\Component\HttpKernel\Event\FilterControllerEvent`

This event is not used by `FrameworkBundle`, but can be an entry point used to modify the controller that should be executed:

```
use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

public function onKernelController(FilterControllerEvent $event)
{
    $controller = $event->getController();
    // ...

    // the controller can be changed to any PHP callable
    $event->setController($controller);
}
```

kernel.view Event *Event Class*: Symfony\Component\HttpKernel\Event\GetResponseForControllerResult

This event is not used by FrameworkBundle, but it can be used to implement a view sub-system. This event is called *only* if the Controller does *not* return a Response object. The purpose of the event is to allow some other return value to be converted into a Response.

The value returned by the Controller is accessible via the getControllerResult method:

```
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelView(GetResponseForControllerResultEvent $event)
{
    $val = $event->getReturnValue();
    $response = new Response();
    // some how customize the Response from the return value

    $event->setResponse($response);
}
```

kernel.response Event *Event Class*: Symfony\Component\HttpKernel\Event\FilterResponseEvent

The purpose of this event is to allow other systems to modify or replace the Response object after its creation:

```
public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    // .. modify the response object
}
```

The FrameworkBundle registers several listeners:

- Symfony\Component\HttpKernel\EventListener\ProfilerListener: collects data for the current request;
- Symfony\Bundle\WebProfilerBundle\EventListener\WebDebugToolbarListener: injects the Web Debug Toolbar;
- Symfony\Component\HttpKernel\EventListener\ResponseListener: fixes the Response Content-Type based on the request format;
- Symfony\Component\HttpKernel\EventListener\EsiListener: adds a Surrogate-Control HTTP header when the Response needs to be parsed for ESI tags.

kernel.exception Event *Event Class*: Symfony\Component\HttpKernel\Event\GetResponseForExceptionEv

`FrameworkBundle` registers an `Symfony\Component\HttpKernel\EventListener\ExceptionListener` that forwards the `Request` to a given Controller (the value of the `exception_listener.controller` parameter – must be in the `class::method` notation).

A listener on this event can create and set a `Response` object, create and set a new `Exception` object, or do nothing:

```
use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelException(GetResponseForExceptionEvent $event)
{
    $exception = $event->getException();
    $response = new Response();
    // setup the Response object based on the caught exception
    $event->setResponse($response);

    // you can alternatively set a new Exception
    // $exception = new \Exception('Some special exception');
    // $event->setException($exception);
}
```

The Event Dispatcher

Objected Oriented code has gone a long way to ensuring code extensibility. By creating classes that have well defined responsibilities, your code becomes more flexible and a developer can extend them with subclasses to modify their behaviors. But if he wants to share his changes with other developers who have also made their own subclasses, code inheritance is moot.

Consider the real-world example where you want to provide a plugin system for your project. A plugin should be able to add methods, or do something before or after a method is executed, without interfering with other plugins. This is not an easy problem to solve with single inheritance, and multiple inheritance (were it possible with PHP) has its own drawbacks.

The Symfony2 Event Dispatcher implements the Observer pattern in a simple and effective way to make all these things possible and to make your projects truly extensible.

Take a simple example from the Symfony2 HttpKernel component. Once a `Response` object has been created, it may be useful to allow other elements in the system to modify it (e.g. add some cache headers) before it's actually used. To make this possible, the Symfony2 kernel throws an event - `kernel.response`. Here's how it works:

- A *listener* (PHP object) tells a central *dispatcher* object that it wants to listen to the `kernel.response` event;
- At some point, the Symfony2 kernel tells the *dispatcher* object to dispatch the `kernel.response` event, passing with it an `Event` object that has access to the `Response` object;
- The dispatcher notifies (i.e. calls a method on) all listeners of the `kernel.response` event, allowing each of them to make modifications to the `Response` object.

Events

When an event is dispatched, it's identified by a unique name (e.g. `kernel.response`), which any number of listeners might be listening to. An `Symfony\Component\EventDispatcher\Event` instance is also created and passed to all of the listeners. As you'll see later, the `Event` object itself often contains data about the event being dispatched.

Naming Conventions The unique event name can be any string, but optionally follows a few simple naming conventions:

- use only lowercase letters, numbers, dots (`.`), and underscores (_);
- prefix names with a namespace followed by a dot (e.g. `kernel.`);
- end names with a verb that indicates what action is being taken (e.g. `request`).

Here are some examples of good event names:

- `kernel.response`
- `form.pre_set_data`

Event Names and Event Objects When the dispatcher notifies listeners, it passes an actual `Event` object to those listeners. The base `Event` class is very simple: it contains a method for stopping *event propagation*, but not much else.

Often times, data about a specific event needs to be passed along with the `Event` object so that the listeners have needed information. In the case of the `kernel.response` event, the `Event` object that's created and passed to each listener is actually of type `Symfony\Component\HttpKernel\Event\FilterResponseEvent`, a subclass of the base `Event` object. This class contains methods such as `getResponse` and `setResponse`, allowing listeners to get or even replace the `Response` object.

The moral of the story is this: when creating a listener to an event, the `Event` object that's passed to the listener may be a special subclass that has additional methods for retrieving information from and responding to the event.

The Dispatcher

The dispatcher is the central object of the event dispatcher system. In general, a single dispatcher is created, which maintains a registry of listeners. When an event is dispatched via the dispatcher, it notifies all listeners registered with that event.

```
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();
```

Connecting Listeners

To take advantage of an existing event, you need to connect a listener to the dispatcher so that it can be notified when the event is dispatched. A call to the dispatcher `addListener()` method associates any valid PHP callable to an event:

```
$listener = new AcmeListener();
$dispatcher->addListener('foo.action', array($listener, 'onFooAction'));
```

The `addListener()` method takes up to three arguments:

- The event name (string) that this listener wants to listen to;
- A PHP callable that will be notified when an event is thrown that it listens to;
- An optional priority integer (higher equals more important) that determines when a listener is triggered versus other listeners (defaults to `0`). If two listeners have the same priority, they are executed in the order that they were added to the dispatcher.

Note: A PHP callable is a PHP variable that can be used by the `call_user_func()` function and returns `true` when passed to the `is_callable()` function. It can be a `\Closure` instance, a string representing a function, or an array representing an object method or a class method.

So far, you've seen how PHP objects can be registered as listeners. You can also register PHP Closures as event listeners:

```
use Symfony\Component\EventDispatcher\Event;

$dispatcher->addListener('foo.action', function (Event $event) {
    // will be executed when the foo.action event is dispatched
});
```

Once a listener is registered with the dispatcher, it waits until the event is notified. In the above example, when the `foo.action` event is dispatched, the dispatcher calls the `AcmeListener::onFooAction` method and passes the `Event` object as the single argument:

```
use Symfony\Component\EventDispatcher\Event;

class AcmeListener
{
    // ...

    public function onFooAction(Event $event)
    {
        // do something
    }
}
```

Tip: If you use the Symfony2 MVC framework, listeners can be registered via your *configuration*. As an added bonus, the listener objects are instantiated only when needed.

In many cases, a special `Event` subclass that's specific to the given event is passed to the listener. This gives the listener access to special information about the event. Check the documentation or implementation of each event to determine the exact `Symfony\Component\EventDispatcher\Event` instance that's being passed. For example, the `kernel.event` event passes an instance of `Symfony\Component\HttpKernel\Event\FilterResponseEvent`:

```
use Symfony\Component\HttpKernel\Event\FilterResponseEvent

public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    $request = $event->getRequest();

    // ...
}
```

Creating and Dispatching an Event

In addition to registering listeners with existing events, you can create and throw your own events. This is useful when creating third-party libraries and also when you want to keep different components of your own system flexible and decoupled.

The Static `Events` Class Suppose you want to create a new Event - `store.order` - that is dispatched each time an order is created inside your application. To keep things organized, start by creating a `StoreEvents` class inside your application that serves to define and document your event:

```
namespace Acme\StoreBundle;

final class StoreEvents
{
    /**
     * The store.order event is thrown each time an order is created
     * in the system.
     *
     * The event listener receives an Acme\StoreBundle\Event\FilterOrderEvent
     * instance.
     *
     * @var string
     */
    const onStoreOrder = 'store.order';
}
```

Notice that this class doesn't actually *do* anything. The purpose of the `StoreEvents` class is just to be a location where information about common events can be centralized. Notice also that a special `FilterOrderEvent` class will be passed to each listener of this event.

Creating an Event object Later, when you dispatch this new event, you'll create an `Event` instance and pass it to the dispatcher. The dispatcher then passes this same instance to each of the listeners of the event. If you don't need to pass any information to your listeners, you can use the default `Symfony\Component\EventDispatcher\Event` class. Most of the time, however, you *will* need to pass information about the event to each listener. To accomplish this, you'll create a new class that extends `Symfony\Component\EventDispatcher\Event`.

In this example, each listener will need access to some pretend `Order` object. Create an `Event` class that makes this possible:

```
namespace Acme\StoreBundle\Event;

use Symfony\Component\EventDispatcher\Event;
use Acme\StoreBundle\Order;

class FilterOrderEvent extends Event
{
    protected $order;

    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    public function getOrder()
    {
        return $this->order;
    }
}
```

Each listener now has access to the `Order` object via the `getOrder` method.

Dispatch the Event The **:method:'Symfony\\Component\\EventDispatcher\\EventDispatcher::dispatch'** method notifies all listeners of the given event. It takes two arguments: the name of the event to dispatch and the `Event` instance to pass to each listener of that event:

```
use Acme\StoreBundle\StoreEvents;
use Acme\StoreBundle\Order;
use Acme\StoreBundle\Event\FilterOrderEvent;

// the order is somehow created or retrieved
$order = new Order();
// ...

// create the FilterOrderEvent and dispatch it
$event = new FilterOrderEvent($order);
$dispatcher->dispatch(StoreEvents::onStoreOrder, $event);
```

Notice that the special `FilterOrderEvent` object is created and passed to the `dispatch` method. Now, any listener to the `store.order` event will receive the `FilterOrderEvent` and have access to the `Order` object via the `getOrder` method:

```
// some listener class that's been registered for onStoreOrder
use Acme\StoreBundle\Event\FilterOrderEvent;

public function onStoreOrder(FilterOrderEvent $event)
{
    $order = $event->getOrder();
    // do something to or with the order
}
```

Passing along the Event Dispatcher Object

If you have a look at the `EventDispatcher` class, you will notice that the class does not act as a Singleton (there is no `getInstance()` static method). That is intentional, as you might want to have several concurrent event dispatchers in a single PHP request. But it also means that you need a way to pass the dispatcher to the objects that need to connect or notify events.

The best practice is to inject the event dispatcher object into your objects, aka dependency injection.

You can use constructor injection:

```
class Foo
{
    protected $dispatcher = null;

    public function __construct(EventDispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }
}
```

Or setter injection:

```
class Foo
{
    protected $dispatcher = null;

    public function setEventDispatcher(EventDispatcher $dispatcher)
    {
```

```
        $this->dispatcher = $dispatcher;
    }
}
```

Choosing between the two is really a matter of taste. Many tend to prefer the constructor injection as the objects are fully initialized at construction time. But when you have a long list of dependencies, using setter injection can be the way to go, especially for optional dependencies.

Tip: If you use dependency injection like we did in the two examples above, you can then use the Symfony2 Dependency Injection component to elegantly manage the injection of the `event_dispatcher` service for these objects.

```
    # src/Acme/HelloBundle/Resources/config/services.yml
    services:
        foo_service:
            class: Acme/HelloBundle/Foo/FooService
            arguments: [@event_dispatcher]
```

Using Event Subscribers

The most common way to listen to an event is to register an *event listener* with the dispatcher. This listener can listen to one or more events and is notified each time those events are dispatched.

Another way to listen to events is via an *event subscriber*. An event subscriber is a PHP class that's able to tell the dispatcher exactly which events it should subscribe to. It implements the `Symfony\Component\EventDispatcher\EventSubscriberInterface` interface, which requires a single static method called `getSubscribedEvents`. Take the following example of a subscriber that subscribes to the `kernel.response` and `store.order` events:

```
namespace Acme\StoreBundle\Event;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class StoreSubscriber implements EventSubscriberInterface
{
    static public function getSubscribedEvents()
    {
        return array(
            'kernel.response' => 'onKernelResponse',
            'store.order'     => 'onStoreOrder',
        );
    }

    public function onKernelResponse(FilterResponseEvent $event)
    {
        // ...
    }

    public function onStoreOrder(FilterOrderEvent $event)
    {
        // ...
    }
}
```

This is very similar to a listener class, except that the class itself can tell the dispatcher

which events it should listen to. To register a subscriber with the dispatcher, use the **:method:'Symfony\\Component\\EventDispatcher\\EventDispatcher::addSubscriber'** method:

```
use Acme\StoreBundle\Event\StoreSubscriber;

$subscriber = new StoreSubscriber();
$dispatcher->addSubscriber($subscriber);
```

The dispatcher will automatically register the subscriber for each event returned by the `getSubscribedEvents` method. This method returns an array indexed by event names and whose values are either the method name to call or an array composed of the method name to call and a priority.

Tip: If you use the Symfony2 MVC framework, subscribers can be registered via your *configuration*. As an added bonus, the subscriber objects are instantiated only when needed.

Stopping Event Flow/Propagation

In some cases, it may make sense for a listener to prevent any other listeners from being called. In other words, the listener needs to be able to tell the dispatcher to stop all propagation of the event to future listeners (i.e. to not notify any more listeners). This can be accomplished from inside a listener via the **:method:'Symfony\\Component\\EventDispatcher\\Event::stopPropagation'** method:

```
use Acme\StoreBundle\Event\FilterOrderEvent;

public function onStoreOrder(FilterOrderEvent $event)
{
    // ...

    $event->stopPropagation();
}
```

Now, any listeners to `store.order` that have not yet been called will *not* be called.

Profiler

When enabled, the Symfony2 profiler collects useful information about each request made to your application and store them for later analysis. Use the profiler in the development environment to help you to debug your code and enhance performance; use it in the production environment to explore problems after the fact.

You rarely have to deal with the profiler directly as Symfony2 provides visualizer tools like the Web Debug Toolbar and the Web Profiler. If you use the Symfony2 Standard Edition, the profiler, the web debug toolbar, and the web profiler are all already configured with sensible settings.

Note: The profiler collects information for all requests (simple requests, redirects, exceptions, Ajax requests, ESI requests; and for all HTTP methods and all formats). It means that for a single URL, you can have several associated profiling data (one per external request/response pair).

Visualizing Profiling Data

Using the Web Debug Toolbar In the development environment, the web debug toolbar is available at the bottom of all pages. It displays a good summary of the profiling data that gives you instant access to a lot of useful information when something does not work as expected.

If the summary provided by the Web Debug Toolbar is not enough, click on the token link (a string made of 13 random characters) to access the Web Profiler.

Note: If the token is not clickable, it means that the profiler routes are not registered (see below for configuration information).

Analyzing Profiling data with the Web Profiler The Web Profiler is a visualization tool for profiling data that you can use in development to debug your code and enhance performance; but it can also be used to explore problems that occur in production. It exposes all information collected by the profiler in a web interface.

Accessing the Profiling information You don't need to use the default visualizer to access the profiling information. But how can you retrieve profiling information for a specific request after the fact? When the profiler stores data about a Request, it also associates a token with it; this token is available in the `X-Debug-Token` HTTP header of the Response:

```
$profile = $container->get('profiler')->loadProfileFromResponse($response);

$profile = $container->get('profiler')->loadProfile($token);
```

Tip: When the profiler is enabled but not the web debug toolbar, or when you want to get the token for an Ajax request, use a tool like Firebug to get the value of the `X-Debug-Token` HTTP header.

Use the `find()` method to access tokens based on some criteria:

```
// get the latest 10 tokens
$tokens = $container->get('profiler')->find('', '', 10);

// get the latest 10 tokens for all URL containing /admin/
$tokens = $container->get('profiler')->find('', '/admin/', 10);

// get the latest 10 tokens for local requests
$tokens = $container->get('profiler')->find('127.0.0.1', '', 10);
```

If you want to manipulate profiling data on a different machine than the one where the information were generated, use the `export()` and `import()` methods:

```
// on the production machine
$profile = $container->get('profiler')->loadProfile($token);
$data = $profiler->export($profile);

// on the development machine
$profiler->import($data);
```

Configuration The default Symfony2 configuration comes with sensible settings for the profiler, the web debug toolbar, and the web profiler. Here is for instance the configuration for the development environment:

- *YAML*

```
# load the profiler
framework:
    profiler: { only_exceptions: false }

# enable the web profiler
web_profiler:
```

```
        toolbar: true
        intercept_redirects: true
        verbose: true
```

- *XML*

```
    <!-- xmlns:webprofiler="http://symfony.com/schema/dic/webprofiler" -->
    <!-- xsi:schemaLocation="http://symfony.com/schema/dic/webprofiler http://symfony.com/schema/dic

    <!-- load the profiler -->
    <framework:config>
        <framework:profiler only-exceptions="false" />
    </framework:config>

    <!-- enable the web profiler -->
    <webprofiler:config
        toolbar="true"
        intercept-redirects="true"
        verbose="true"
    />
```

- *PHP*

```
    // load the profiler
    $container->loadFromExtension('framework', array(
        'profiler' => array('only-exceptions' => false),
    ));

    // enable the web profiler
    $container->loadFromExtension('web_profiler', array(
        'toolbar' => true,
        'intercept-redirects' => true,
        'verbose' => true,
    ));
```

When `only-exceptions` is set to `true`, the profiler only collects data when an exception is thrown by the application.

When `intercept-redirects` is set to `true`, the web profiler intercepts the redirects and gives you the opportunity to look at the collected data before following the redirect.

When `verbose` is set to `true`, the Web Debug Toolbar displays a lot of information. Setting `verbose` to `false` hides some secondary information to make the toolbar shorter.

If you enable the web profiler, you also need to mount the profiler routes:

- *YAML*

```
    _profiler:
        resource: @WebProfilerBundle/Resources/config/routing/profiler.xml
        prefix:   /_profiler
```

- *XML*

```
    <import resource="@WebProfilerBundle/Resources/config/routing/profiler.xml" prefix="/_profiler"
```

- *PHP*

```
    $collection->addCollection($loader->import("@WebProfilerBundle/Resources/config/routing/profiler
```

As the profiler adds some overhead, you might want to enable it only under certain circumstances in the production environment. The `only-exceptions` settings limits profiling to 500 pages, but what if you want to get information when the client IP comes from a specific address, or for a limited portion of the website? You can use a request matcher:

- *YAML*

```yaml
# enables the profiler only for request coming for the 192.168.0.0 network
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24 }

# enables the profiler only for the /admin URLs
framework:
    profiler:
        matcher: { path: "^/admin/" }

# combine rules
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24, path: "^/admin/" }

# use a custom matcher instance defined in the "custom_matcher" service
framework:
    profiler:
        matcher: { service: custom_matcher }
```

- *XML*

```xml
<!-- enables the profiler only for request coming for the 192.168.0.0 network -->
<framework:config>
    <framework:profiler>
        <framework:matcher ip="192.168.0.0/24" />
    </framework:profiler>
</framework:config>

<!-- enables the profiler only for the /admin URLs -->
<framework:config>
    <framework:profiler>
        <framework:matcher path="^/admin/" />
    </framework:profiler>
</framework:config>

<!-- combine rules -->
<framework:config>
    <framework:profiler>
        <framework:matcher ip="192.168.0.0/24" path="^/admin/" />
    </framework:profiler>
</framework:config>

<!-- use a custom matcher instance defined in the "custom_matcher" service -->
<framework:config>
    <framework:profiler>
        <framework:matcher service="custom_matcher" />
    </framework:profiler>
</framework:config>
```

- *PHP*

```
    // enables the profiler only for request coming for the 192.168.0.0 network
    $container->loadFromExtension('framework', array(
        'profiler' => array(
            'matcher' => array('ip' => '192.168.0.0/24'),
        ),
    ));

    // enables the profiler only for the /admin URLs
    $container->loadFromExtension('framework', array(
        'profiler' => array(
            'matcher' => array('path' => '^/admin/'),
        ),
    ));

    // combine rules
    $container->loadFromExtension('framework', array(
        'profiler' => array(
            'matcher' => array('ip' => '192.168.0.0/24', 'path' => '^/admin/'),
        ),
    ));

    # use a custom matcher instance defined in the "custom_matcher" service
    $container->loadFromExtension('framework', array(
        'profiler' => array(
            'matcher' => array('service' => 'custom_matcher'),
        ),
    ));
```

Learn more from the Cookbook

- How to use the Profiler in a Functional Test
- How to create a custom Data Collector
- How to extend a Class without using Inheritance
- How to customize a Method Behavior without using Inheritance

2.1.18 The Symfony2 Stable API

The Symfony2 stable API is a subset of all Symfony2 published public methods (components and core bundles) that share the following properties:

- The namespace and class name won't change;
- The method name won't change;
- The method signature (arguments and return value type) won't change;
- The semantic of what the method does won't change.

The implementation itself can change though. The only valid case for a change in the stable API is in order to fix a security issue.

The stable API is based on a whitelist, tagged with *@api*. Therefore, everything not tagged explicitly is not part of the stable API.

Tip: Any third party bundle should also publish its own stable API.

As of Symfony 2.0, the following components have a public tagged API:

- BrowserKit
- ClassLoader
- Console
- CssSelector
- DependencyInjection
- DomCrawler
- EventDispatcher
- Finder
- HttpFoundation
- HttpKernel
- Locale
- Process
- Routing
- Templating
- Translation
- Validator
- Yaml
- Symfony2 and HTTP Fundamentals
- Symfony2 versus Flat PHP
- Installing and Configuring Symfony
- Creating Pages in Symfony2
- Controller
- Routing
- Creating and using Templates
- Databases and Doctrine ("The Model")
- Testing
- Validation
- Forms
- Security
- HTTP Cache
- Translations
- Service Container
- Performance
- Internals
- The Symfony2 Stable API

CHAPTER 3

Cookbook

3.1 Cookbook

3.1.1 How to Create and store a Symfony2 Project in git

Tip: Though this entry is specifically about git, the same generic principles will apply if you're storing your project in Subversion.

Once you've read through Creating Pages in Symfony2 and become familiar with using Symfony, you'll no-doubt be ready to start your own project. In this cookbook article, you'll learn the best way to start a new Symfony2 project that's stored using the git source control management system.

Initial Project Setup

To get started, you'll need to download Symfony and initialize your local git repository:

1. Download the Symfony2 Standard Edition without vendors.
2. Unzip/untar the distribution. It will create a folder called Symfony with your new project structure, config files, etc. Rename it to whatever you like.
3. Create a new file called `.gitignore` at the root of your new project (e.g. next to the `deps` file) and paste the following into it. Files matching these patterns will be ignored by git:

```
/web/bundles/
/app/bootstrap*
/app/cache/*
/app/logs/*
/vendor/
/app/config/parameters.yml
```

4. Copy `app/config/parameters.yml` to `app/config/parameters.yml.dist`. The `parameters.yml` file is ignored by git (see above) so that machine-specific settings like database passwords aren't committed. By creating the `parameters.yml.dist` file, new developers can quickly clone the project, copy this file to `parameters.yml`, customize it, and start developing.
5. Initialize your git repository:

```
$ git init
```

6. Add all of the initial files to git:

```
$ git add .
```

7. Create an initial commit with your started project:

```
$ git commit -m "Initial commit"
```

8. Finally, download all of the third-party vendor libraries:

```
$ php bin/vendors install
```

At this point, you have a fully-functional Symfony2 project that's correctly committed to git. You can immediately begin development, committing the new changes to your git repository.

Tip: After execution of the command:

```
$ php bin/vendors install
```

your project will contain complete the git history of all the bundles and libraries defined in the `deps` file. It can be as much as 100 MB! You can remove the git history directories with the following command:

```
$ find vendor -name .git -type d | xargs rm -rf
```

The command removes all `.git` directories contained inside the `vendor` directory.

If you want to update bundles defined in `deps` file after this, you will have to reinstall them:

```
$ php bin/vendors install --reinstall
```

You can continue to follow along with the Creating Pages in Symfony2 chapter to learn more about how to configure and develop inside your application.

Tip: The Symfony2 Standard Edition comes with some example functionality. To remove the sample code, follow the instructions on the Standard Edition Readme.

Vendors and Submodules

Instead of using the `deps`, `bin/vendors` system for managing your vendor libraries, you may instead choose to use native git submodules. There is nothing wrong with this approach, though the `deps` system is the official way to solve this problem and git submodules can be difficult to work with at times.

Storing your Project on a Remote Server

You now have a fully-functional Symfony2 project stored in git. However, in most cases, you'll also want to store your project on a remote server both for backup purposes, and so that other developers can collaborate on the project.

The easiest way to store your project on a remote server is via GitHub. Public repositories are free, however you will need to pay a monthly fee to host private repositories.

Alternatively, you can store your git repository on any server by creating a barebones repository and then pushing to it. One library that helps manage this is Gitolite.

3.1.2 How to Create and store a Symfony2 Project in Subversion

---

Tip: This entry is specifically about Subversion, and based on principles found in How to Create and store a Symfony2

Project in git.

Once you've read through Creating Pages in Symfony2 and become familiar with using Symfony, you'll no-doubt be ready to start your own project. The preferred method to manage Symfony2 projects is using git but some prefer to use Subversion which is totally fine!. In this cookbook article, you'll learn how to manage your project using svn in a similar manner you would do with git.

Tip: This is **a** method to tracking your Symfony2 project in a Subversion repository. There are several ways to do and this one is simply one that works.

The Subversion Repository

For this article we will suppose that your repository layout follows the widespread standard structure:

```
myproject/
    branches/
    tags/
    trunk/
```

Tip: Most subversion hosting should follow this standard practice. This is the recommended layout in Version Control with Subversion and the layout used by most free hosting (see *Subversion hosting solutions*).

Initial Project Setup

To get started, you'll need to download Symfony2 and get the basic Subversion setup:

1. Download the Symfony2 Standard Edition without or without vendors.
2. Unzip/untar the distribution. It will create a folder called Symfony with your new project structure, config files, etc. Rename it to whatever you like.
3. Checkout the Subversion repository that will host this project. Let's say it is hosted on Google code and called `myproject`:

```
$ svn checkout http://myproject.googlecode.com/svn/trunk myproject
```

4. Copy the Symfony2 project files in the subversion folder:

```
$ mv Symfony/* myproject/
```

5. Let's now set the ignore rules. Not everything *should* be stored in your subversion repository. Some files (like the cache) are generated and others (like the database configuration) are meant to be customized on each machine. This makes use of the `svn:ignore` property, so that we can ignore specific files.

```
$ cd myproject/
$ svn add --depth=empty app app/cache app/logs app/config web

$ svn propset svn:ignore "vendor" .
$ svn propset svn:ignore "bootstrap*" app/
$ svn propset svn:ignore "parameters.ini" app/config/
$ svn propset svn:ignore "*" app/cache/
$ svn propset svn:ignore "*" app/logs/

$ svn propset svn:ignore "bundles" web
```

```
$ svn ci -m "commit basic symfony ignore list (vendor, app/bootstrap*, app/config/parameters
```

6. The rest of the files can now be added and committed to the project:

```
$ svn add --force .
$ svn ci -m "add basic Symfony Standard 2.X.Y"
```

7. Copy `app/config/parameters.ini` to `app/config/parameters.ini.dist`. The `parameters.ini` file is ignored by svn (see above) so that machine-specific settings like database passwords aren't committed. By creating the `parameters.ini.dist` file, new developers can quickly clone the project, copy this file to `parameters.ini`, customize it, and start developing.

8. Finally, download all of the third-party vendor libraries:

```
$ php bin/vendors install
```

Tip: git has to be installed to run `bin/vendors`, this is the protocol used to fetch vendor libraries. This only means that `git` is used as a tool to basically help download the libraries in the `vendor/` directory.

At this point, you have a fully-functional Symfony2 project stored in your Subversion repository. The development can start with commits in the Subversion repository.

You can continue to follow along with the Creating Pages in Symfony2 chapter to learn more about how to configure and develop inside your application.

Tip: The Symfony2 Standard Edition comes with some example functionality. To remove the sample code, follow the instructions on the Standard Edition Readme.

Managing Vendor Libraries with bin/vendors and deps

Every Symfony project uses a group of third-party "vendor" libraries. One way or another the goal is to download these files into your `vendor/` directory and, ideally, to give you some sane way to manage the exact version you need for each.

By default, these libraries are downloaded by running a `php bin/vendors install` "downloader" script. This script reads from the `deps` file at the root of your project. This is an ini-formatted script, which holds a list of each of the external libraries you need, the directory each should be downloaded to, and (optionally) the version to be downloaded. The `bin/vendors` script uses `git` to downloaded these, solely because these external libraries themselves tend to be stored via git. The `bin/vendors` script also reads the `deps.lock` file, which allows you to pin each library to an exact git commit hash.

It's important to realize that these vendor libraries are *not* actually part of *your* repository. Instead, they're simply un-tracked files that are downloaded into the `vendor/` directory by the `bin/vendors` script. But since all the information needed to download these files is saved in `deps` and `deps.lock` (which *are* stored) in our repository), any other developer can use our project, run `php bin/vendors install`, and download the exact same set of vendor libraries. This means that you're controlling exactly what each vendor library looks like, without needing to actually commit them to *your* repository.

So, whenever a developer uses your project, he/she should run the `php bin/vendors install` script to ensure that all of the needed vendor libraries are downloaded.

Upgrading Symfony

Since Symfony is just a group of third-party libraries and third-party libraries are entirely controlled through `deps` and `deps.lock`, upgrading Symfony means simply upgrading each of these files to match their state in the latest Symfony Standard Edition.
Of course, if you've added new entries to `deps` or `deps.lock`, be sure to replace only the original parts (i.e. be sure not to also delete any of your custom entries).

Caution: There is also a `php bin/vendors update` command, but this has nothing to do with upgrading your project and you will normally not need to use it. This command is used to freeze the versions of all of your vendor libraries by updating them to the version specified in `deps` and recording it into the `deps.lock` file.

Subversion hosting solutions

The biggest difference between git and svn is that Subversion *needs* a central repository to work. You then have several solutions:

- Self hosting: create your own repository and access it either through the filesystem or the network. To help in this task you can read Version Control with Subversion.
- Third party hosting: there are a lot of serious free hosting solutions available like GitHub, Google code, SourceForge or Gna. Some of them offer git hosting as well.

3.1.3 How to customize Error Pages

When any exception is thrown in Symfony2, the exception is caught inside the `Kernel` class and eventually forwarded to a special controller, `TwigBundle:Exception:show` for handling. This controller, which lives inside the core `TwigBundle`, determines which error template to display and the status code that should be set for the given exception.

Error pages can be customized in two different ways, depending on how much control you need:

1. Customize the error templates of the different error pages (explained below);
2. Replace the default exception controller `TwigBundle::Exception:show` with your own controller and handle it however you want (see *exception_controller in the Twig reference*);

Tip: The customization of exception handling is actually much more powerful than what's written here. An internal event, `kernel.exception`, is thrown which allows complete control over exception handling. For more information, see *kernel.exception Event*.

All of the error templates live inside `TwigBundle`. To override the templates, we simply rely on the standard method for overriding templates that live inside a bundle. For more information, see *Overriding Bundle Templates*.

For example, to override the default error template that's shown to the end-user, create a new template located at `app/Resources/TwigBundle/views/Exception/error.html.twig`:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>An Error Occurred: {{ status_text }}</title>
</head>
<body>
    <h1>Oops! An Error Occurred</h1>
```

```
    <h2>The server returned a "{{ status_code }} {{ status_text }}".</h2>
</body>
</html>
```

Tip: If you're not familiar with Twig, don't worry. Twig is a simple, powerful and optional templating engine that integrates with `Symfony2`. For more information about Twig see Creating and using Templates.

In addition to the standard HTML error page, Symfony provides a default error page for many of the most common response formats, including JSON (`error.json.twig`), XML, (`error.xml.twig`), and even Javascript (`error.js.twig`), to name a few. To override any of these templates, just create a new file with the same name in the `app/Resources/TwigBundle/views/Exception` directory. This is the standard way of overriding any template that lives inside a bundle.

Customizing the 404 Page and other Error Pages

You can also customize specific error templates according to the HTTP status code. For instance, create a `app/Resources/TwigBundle/views/Exception/error404.html.twig` template to display a special page for 404 (page not found) errors.

Symfony uses the following algorithm to determine which template to use:

- First, it looks for a template for the given format and status code (like `error404.json.twig`);
- If it does not exist, it looks for a template for the given format (like `error.json.twig`);
- If it does not exist, it falls back to the HTML template (like `error.html.twig`).

Tip: To see the full list of default error templates, see the `Resources/views/Exception` directory of the `TwigBundle`. In a standard Symfony2 installation, the `TwigBundle` can be found at `vendor/symfony/src/Symfony/Bundle/TwigBundle`. Often, the easiest way to customize an error page is to copy it from the `TwigBundle` into `app/Resources/TwigBundle/views/Exception` and then modify it.

Note: The debug-friendly exception pages shown to the developer can even be customized in the same way by creating templates such as `exception.html.twig` for the standard HTML exception page or `exception.json.twig` for the JSON exception page.

3.1.4 How to define Controllers as Services

In the book, you've learned how easily a controller can be used when it extends the base `Symfony\Bundle\FrameworkBundle\Controller\Controller` class. While this works fine, controllers can also be specified as services.

To refer to a controller that's defined as a service, use the single colon (:) notation. For example, suppose we've defined a service called `my_controller` and we want to forward to a method called `indexAction()` inside the service:

```
$this->forward('my_controller:indexAction', array('foo' => $bar));
```

You need to use the same notation when defining the route `_controller` value:

```
my_controller:
    pattern:   /
    defaults:  { _controller: my_controller:indexAction }
```

To use a controller in this way, it must be defined in the service container configuration. For more information, see the Service Container chapter.

When using a controller defined as a service, it will most likely not extend the base `Controller` class. Instead of relying on its shortcut methods, you'll interact directly with the services that you need. Fortunately, this is usually pretty easy and the base `Controller` class itself is a great source on how to perform many common tasks.

Note: Specifying a controller as a service takes a little bit more work. The primary advantage is that the entire controller or any services passed to the controller can be modified via the service container configuration. This is especially useful when developing an open-source bundle or any bundle that will be used in many different projects. So, even if you don't specify your controllers as services, you'll likely see this done in some open-source Symfony2 bundles.

3.1.5 How to force routes to always use HTTPS or HTTP

Sometimes, you want to secure some routes and be sure that they are always accessed via the HTTPS protocol. The Routing component allows you to enforce the URI scheme via the `_scheme` requirement:

- *YAML*

```
secure:
    pattern:  /secure
    defaults: { _controller: AcmeDemoBundle:Main:secure }
    requirements:
        _scheme:  https
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="secure" pattern="/secure">
        <default key="_controller">AcmeDemoBundle:Main:secure</default>
        <requirement key="_scheme">https</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('secure', new Route('/secure', array(
    '_controller' => 'AcmeDemoBundle:Main:secure',
), array(
    '_scheme' => 'https',
)));

return $collection;
```

The above configuration forces the `secure` route to always use HTTPS.

When generating the `secure` URL, and if the current scheme is HTTP, Symfony will automatically generate an absolute URL with HTTPS as the scheme:

```
# If the current scheme is HTTPS
{{ path('secure') }}
# generates /secure

# If the current scheme is HTTP
{{ path('secure') }}
# generates https://example.com/secure
```

The requirement is also enforced for incoming requests. If you try to access the `/secure` path with HTTP, you will automatically be redirected to the same URL, but with the HTTPS scheme.

The above example uses `https` for the `_scheme`, but you can also force a URL to always use `http`.

Note: The Security component provides another way to enforce HTTP or HTTPs via the `requires_channel` setting. This alternative method is better suited to secure an "area" of your website (all URLs under `/admin`) or when you want to secure URLs defined in a third party bundle.

3.1.6 How to allow a "/" character in a route parameter

Sometimes, you need to compose URLs with parameters that can contain a slash `/`. For example, take the classic `/hello/{name}` route. By default, `/hello/Fabien` will match this route but not `/hello/Fabien/Kris`. This is because Symfony uses this character as separator between route parts.

This guide covers how you can modify a route so that `/hello/Fabien/Kris` matches the `/hello/{name}` route, where `{name}` equals `Fabien/Kris`.

Configure the Route

By default, the symfony routing components requires that the parameters match the following regex pattern: `[^/]+`. This means that all characters are allowed except `/`.

You must explicitly allow `/` to be part of your parameter by specifying a more permissive regex pattern.

- *YAML*

```
_hello:
    pattern: /hello/{name}
    defaults: { _controller: AcmeDemoBundle:Demo:hello }
    requirements:
        name: ".+"
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="_hello" pattern="/hello/{name}">
        <default key="_controller">AcmeDemoBundle:Demo:hello</default>
        <requirement key="name">.+</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('_hello', new Route('/hello/{name}', array(
    '_controller' => 'AcmeDemoBundle:Demo:hello',
), array(
    'name' => '.+',
)));

return $collection;
```

- *Annotations*

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class DemoController
{
    /**
     * @Route("/hello/{name}", name="_hello", requirements={"name" = ".+"})
     */
    public function helloAction($name)
    {
        // ...
    }
}
```

That's it! Now, the `{name}` parameter can contain the `/` character.

3.1.7 How to Use Assetic for Asset Management

Assetic combines two major ideas: assets and filters. The assets are files such as CSS, JavaScript and image files. The filters are things that can be applied to these files before they are served to the browser. This allows a separation between the asset files stored in the application and the files actually presented to the user.

Without Assetic, you just serve the files that are stored in the application directly:

- *Twig*

```
<script src="{{ asset('js/script.js') }}" type="text/javascript" />
```

- *PHP*

```
<script src="<?php echo $view['assets']->getUrl('js/script.js') ?>"
        type="text/javascript" />
```

But *with* Assetic, you can manipulate these assets however you want (or load them from anywhere) before serving them. These means you can:

- Minify and combine all of your CSS and JS files
- Run all (or just some) of your CSS or JS files through some sort of compiler, such as LESS, SASS or CoffeeScript
- Run image optimizations on your images

Assets

Using Assetic provides many advantages over directly serving the files. The files do not need to be stored where they are served from and can be drawn from various sources such as from within a bundle:

- *Twig*

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
%}
<script type="text/javascript" src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*')) as $url): ?>
<script type="text/javascript" src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Tip: To bring in CSS stylesheets, you can use the same methodologies seen in this entry, except with the *stylesheets* tag:

- *Twig*

```
{% stylesheets
    '@AcmeFooBundle/Resources/public/css/*'
%}
<link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}
```

- *PHP*

```
<?php foreach ($view['assetic']->stylesheets(
    array('@AcmeFooBundle/Resources/public/css/*')) as $url): ?>
<link rel="stylesheet" href="<?php echo $view->escape($url) ?>" />
<?php endforeach; ?>
```

In this example, all of the files in the `Resources/public/js/` directory of the `AcmeFooBundle` will be loaded and served from a different location. The actual rendered tag might simply look like:

```
<script src="/app_dev.php/js/abcd123.js"></script>
```

Note: This is a key point: once you let Assetic handle your assets, the files are served from a different location. This *can* cause problems with CSS files that reference images by their relative path. However, this can be fixed by using the `cssrewrite` filter, which updates paths in CSS files to reflect their new location.

Combining Assets

You can also combine several files into one. This helps to reduce the number of HTTP requests, which is great for front end performance. It also allows you to maintain the files more easily by splitting them into manageable parts. This can help with re-usability as you can easily split project-specific files from those which can be used in other applications, but still serve them as a single file:

- *Twig*

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    '@AcmeBarBundle/Resources/public/js/form.js'
    '@AcmeBarBundle/Resources/public/js/calendar.js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*',
          '@AcmeBarBundle/Resources/public/js/form.js',
          '@AcmeBarBundle/Resources/public/js/calendar.js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

In the *dev* environment, each file is still served individually, so that you can debug problems more easily. However, in the *prod* environment, this will be rendered as a single *script* tag.

Tip: If you're new to Assetic and try to use your application in the `prod` environment (by using the `app.php` controller), you'll likely see that all of your CSS and JS breaks. Don't worry! This is on purpose. For details on using Assetic in the *prod* environment, see *Dumping Asset Files*.

And combining files doesn't only apply to *your* files. You can also use Assetic to combine third party assets, such as jQuery, with your own into a single file:

- *Twig*

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/thirdparty/jquery.js'
    '@AcmeFooBundle/Resources/public/js/*'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/thirdparty/jquery.js',
          '@AcmeFooBundle/Resources/public/js/*')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Filters

Once they're managed by Assetic, you can apply filters to your assets before they are served. This includes filters that compress the output of your assets for smaller file sizes (and better front-end optimization). Other filters can compile JavaScript file from CoffeeScript files and process SASS into CSS. In fact, Assetic has a long list of available filters.

Many of the filters do not do the work directly, but use existing third-party libraries to do the heavy-lifting. This means that you'll often need to install a third-party library to use a filter. The great advantage of using Assetic to invoke these libraries (as opposed to using them directly) is that instead of having to run them manually after you work on the files, Assetic will take care of this for you and remove this step altogether from your development and deployment processes.

To use a filter, you first need to specify it in the Assetic configuration. Adding a filter here doesn't mean it's being used - it just means that it's available to use (we'll use the filter below).

For example to use the JavaScript YUI Compressor the following config should be added:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        yui_js:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="yui_js"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'yui_js' => array(
            'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
        ),
    ),
));
```

Now, to actually *use* the filter on a group of JavaScript files, add it into your template:

- *Twig*

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    filter='yui_js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('yui_js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

A more detailed guide about configuring and using Assetic filters as well as details of Assetic's debug mode can be found in How to Minify JavaScripts and Stylesheets with YUI Compressor.

Controlling the URL used

If you wish to, you can control the URLs that Assetic produces. This is done from the template and is relative to the public document root:

- *Twig*

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    output='js/compiled/main.js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array(),
    array('output' => 'js/compiled/main.js')
) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Note: Symfony also contains a method for cache *busting*, where the final URL generated by Assetic contains a query parameter that can be incremented via configuration on each deployment. For more information, see the *assets_version* configuration option.

Dumping Asset Files

In the `dev` environment, Assetic generates paths to CSS and JavaScript files that don't physically exist on your computer. But they render nonetheless because an internal Symfony controller opens the files and serves back the content (after running any filters).

This kind of dynamic serving of processed assets is great because it means that you can immediately see the new state of any asset files you change. It's also bad, because it can be quite slow. If you're using a lot of filters, it might be downright frustrating.

Fortunately, Assetic provides a way to dump your assets to real files, instead of being generated dynamically.

Dumping Asset Files in the `prod` environment

In the `prod` environment, your JS and CSS files are represented by a single tag each. In other words, instead of seeing each JavaScript file you're including in your source, you'll likely just see something like this:

```
<script src="/app_dev.php/js/abcd123.js"></script>
```

Moreover, that file does **not** actually exist, nor is it dynamically rendered by Symfony (as the asset files are in the `dev` environment). This is on purpose - letting Symfony generate these files dynamically in a production environment is just too slow.

Instead, each time you use your app in the `prod` environment (and therefore, each time you deploy), you should run the following task:

```
php app/console assetic:dump --env=prod --no-debug
```

This will physically generate and write each file that you need (e.g. `/js/abcd123.js`). If you update any of your assets, you'll need to run this again to regenerate the file.

Dumping Asset Files in the `dev` environment

By default, each asset path generated in the `dev` environment is handled dynamically by Symfony. This has no disadvantage (you can see your changes immediately), except that assets can load noticeably slow. If you feel like your assets are loading too slowly, follow this guide.

First, tell Symfony to stop trying to process these files dynamically. Make the following change in your `config_dev.yml` file:

- *YAML*

```
# app/config/config_dev.yml
assetic:
    use_controller: false
```

- *XML*

```
<!-- app/config/config_dev.xml -->
<assetic:config use-controller="false" />
```

- *PHP*

```
// app/config/config_dev.php
$container->loadFromExtension('assetic', array(
    'use_controller' => false,
));
```

Next, since Symfony is no longer generating these assets for you, you'll need to dump them manually. To do so, run the following:

```
php app/console assetic:dump
```

This physically writes all of the asset files you need for your `dev` environment. The big disadvantage is that you need to run this each time you update an asset. Fortunately, by passing the `--watch` option, the command will automatically regenerate assets *as they change*:

```
php app/console assetic:dump --watch
```

Since running this command in the `dev` environment may generate a bunch of files, it's usually a good idea to point your generated assets files to some isolated directory (e.g. `/js/compiled`), to keep things organized:

- *Twig*

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    output='js/compiled/main.js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array(),
    array('output' => 'js/compiled/main.js')
) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

3.1.8 How to Minify JavaScripts and Stylesheets with YUI Compressor

Yahoo! provides an excellent utility for minifying JavaScripts and stylesheets so they travel over the wire faster, the YUI Compressor. Thanks to Assetic, you can take advantage of this tool very easily.

Download the YUI Compressor JAR

The YUI Compressor is written in Java and distributed as a JAR. Download the JAR from the Yahoo! site and save it to `app/Resources/java/yuicompressor.jar`.

Configure the YUI Filters

Now you need to configure two Assetic filters in your application, one for minifying JavaScripts with the YUI Compressor and one for minifying stylesheets:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        yui_css:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
        yui_js:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="yui_css"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
    <assetic:filter
        name="yui_js"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'yui_css' => array(
            'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
        ),
        'yui_js' => array(
            'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
        ),
    ),
));
```

You now have access to two new Assetic filters in your application: `yui_css` and `yui_js`. These will use the YUI Compressor to minify stylesheets and JavaScripts, respectively.

Minify your Assets

You have YUI Compressor configured now, but nothing is going to happen until you apply one of these filters to an asset. Since your assets are a part of the view layer, this work is done in your templates:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('yui_js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Note: The above example assumes that you have a bundle called `AcmeFooBundle` and your JavaScript files are in the `Resources/public/js` directory under your bundle. This isn't important however - you can include your Javascript files no matter where they are.

With the addition of the `yui_js` filter to the asset tags above, you should now see minified JavaScripts coming over the wire much faster. The same process can be repeated to minify your stylesheets.

- *Twig*

```
{% stylesheets '@AcmeFooBundle/Resources/public/css/*' filter='yui_css' %}
<link rel="stylesheet" type="text/css" media="screen" href="{{ asset_url }}" />
{% endstylesheets %}
```

- *PHP*

```
<?php foreach ($view['assetic']->stylesheets(
    array('@AcmeFooBundle/Resources/public/css/*'),
    array('yui_css')) as $url): ?>
<link rel="stylesheet" type="text/css" media="screen" href="<?php echo $view->escape($url) ?>" /
<?php endforeach; ?>
```

Disable Minification in Debug Mode

Minified JavaScripts and Stylesheets are very difficult to read, let alone debug. Because of this, Assetic lets you disable a certain filter when your application is in debug mode. You can do this be prefixing the filter name in your template with a question mark: `?`. This tells Assetic to only apply this filter when debug mode is off.

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='?yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('?yui_js')) as $url): ?>
```

```
    <script src="<?php echo $view->escape($url) ?>"></script>
    <?php endforeach; ?>
```

3.1.9 How to Use Assetic For Image Optimization with Twig Functions

Amongst its many filters, Assetic has four filters which can be used for on-the-fly image optimization. This allows you to get the benefits of smaller file sizes without having to use an image editor to process each image. The results are cached and can be dumped for production so there is no performance hit for your end users.

Using Jpegoptim

Jpegoptim is a utility for optimizing JPEG files. To use it with Assetic, add the following to the Assetic config:

- *YAML*

```
    # app/config/config.yml
    assetic:
        filters:
            jpegoptim:
                bin: path/to/jpegoptim
```

- *XML*

```
    <!-- app/config/config.xml -->
    <assetic:config>
        <assetic:filter
            name="jpegoptim"
            bin="path/to/jpegoptim" />
    </assetic:config>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('assetic', array(
        'filters' => array(
            'jpegoptim' => array(
                'bin' => 'path/to/jpegoptim',
            ),
        ),
    ));
```

Note: Notice that to use jpegoptim, you must have it already installed on your system. The `bin` option points to the location of the compiled binary.

It can now be used from a template:

- *Twig*

```
    {% image '@AcmeFooBundle/Resources/public/images/example.jpg'
        filter='jpegoptim' output='/images/example.jpg'
    %}
    <img src="{{ asset_url }}" alt="Example"/>
    {% endimage %}
```

- *PHP*

```
<?php foreach ($view['assetic']->images(
    array('@AcmeFooBundle/Resources/public/images/example.jpg'),
    array('jpegoptim')) as $url): ?>
<img src="<?php echo $view->escape($url) ?>" alt="Example"/>
<?php endforeach; ?>
```

Removing all EXIF Data

By default, running this filter only removes some of the meta information stored in the file. Any EXIF data and comments are not removed, but you can remove these by using the `strip_all` option:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
            strip_all: true
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="path/to/jpegoptim"
        strip_all="true" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'path/to/jpegoptim',
            'strip_all' => 'true',
        ),
    ),
));
```

Lowering Maximum Quality

The quality level of the JPEG is not affected by default. You can gain further file size reductions by setting the max quality setting lower than the current level of the images. This will of course be at the expense of image quality:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
            max: 70
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="path/to/jpegoptim"
        max="70" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'path/to/jpegoptim',
            'max' => '70',
        ),
    ),
));
```

Shorter syntax: Twig Function

If you're using Twig, it's possible to achieve all of this with a shorter syntax by enabling and using a special Twig function. Start by adding the following config:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
    twig:
        functions:
            jpegoptim: ~
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="path/to/jpegoptim" />
    <assetic:twig>
        <assetic:twig_function
            name="jpegoptim" />
    </assetic:twig>
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'path/to/jpegoptim',
        ),
    ),
```

```
        'twig' => array(
            'functions' => array('jpegoptim'),
            ),
        ),
    ));
```

The Twig template can now be changed to the following:

```
<img src="{{ jpegoptim('@AcmeFooBundle/Resources/public/images/example.jpg') }}"
     alt="Example"/>
```

You can specify the output directory in the config in the following way:

- *YAML*

```
    # app/config/config.yml
    assetic:
        filters:
            jpegoptim:
                bin: path/to/jpegoptim
        twig:
            functions:
                jpegoptim: { output: images/*.jpg }
```

- *XML*

```
    <!-- app/config/config.xml -->
    <assetic:config>
        <assetic:filter
            name="jpegoptim"
            bin="path/to/jpegoptim" />
        <assetic:twig>
            <assetic:twig_function
                name="jpegoptim"
                output="images/*.jpg" />
        </assetic:twig>
    </assetic:config>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('assetic', array(
        'filters' => array(
            'jpegoptim' => array(
                'bin' => 'path/to/jpegoptim',
            ),
        ),
        'twig' => array(
            'functions' => array(
                'jpegoptim' => array(
                    output => 'images/*.jpg'
                ),
            ),
        ),
    ));
```

3.1.10 How to Apply an Assetic Filter to a Specific File Extension

Assetic filters can be applied to individual files, groups of files or even, as you'll see here, files that have a specific extension. To show you how to handle each option, let's suppose that you want to use Assetic's CoffeeScript filter, which compiles CoffeeScript files into Javascript.

The main configuration is just the paths to coffee and node. These default respectively to `/usr/bin/coffee` and `/usr/bin/node`:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="coffee"
        bin="/usr/bin/coffee"
        node="/usr/bin/node" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'coffee' => array(
            'bin' => '/usr/bin/coffee',
            'node' => '/usr/bin/node',
        ),
    ),
));
```

Filter a Single File

You can now serve up a single CoffeeScript file as JavaScript from within your templates:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
    filter='coffee'
%}
<script src="{{ asset_url }}" type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/example.coffee'),
    array('coffee')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

This is all that's needed to compile this CoffeeScript file and server it as the compiled JavaScript.

Filter Multiple Files

You can also combine multiple CoffeeScript files into a single output file:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
               '@AcmeFooBundle/Resources/public/js/another.coffee'
    filter='coffee'
%}
<script src="{{ asset_url }}" type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/example.coffee',
          '@AcmeFooBundle/Resources/public/js/another.coffee'),
    array('coffee')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

Both the files will now be served up as a single file compiled into regular JavaScript.

Filtering based on a File Extension

One of the great advantages of using Assetic is reducing the number of asset files to lower HTTP requests. In order to make full use of this, it would be good to combine *all* your JavaScript and CoffeeScript files together since they will ultimately all be served as JavaScript. Unfortunately just adding the JavaScript files to the files to be combined as above will not work as the regular JavaScript files will not survive the CoffeeScript compilation.

This problem can be avoided by using the `apply_to` option in the config, which allows you to specify that a filter should always be applied to particular file extensions. In this case you can specify that the Coffee filter is applied to all `.coffee` files:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
            apply_to: "\.coffee$"
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="coffee"
        bin="/usr/bin/coffee"
        node="/usr/bin/node"
        apply_to="\.coffee$" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'coffee' => array(
            'bin' => '/usr/bin/coffee',
            'node' => '/usr/bin/node',
            'apply_to' => '\.coffee$',
        ),
    ),
));
```

With this, you no longer need to specify the `coffee` filter in the template. You can also list regular JavaScript files, all of which will be combined and rendered as a single JavaScript file (with only the `.coffee` files being run through the CoffeeScript filter):

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
               '@AcmeFooBundle/Resources/public/js/another.coffee'
               '@AcmeFooBundle/Resources/public/js/regular.js'
%}
<script src="{{ asset_url }}" type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/example.coffee',
          '@AcmeFooBundle/Resources/public/js/another.coffee',
          '@AcmeFooBundle/Resources/public/js/regular.js'),
    as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

3.1.11 How to handle File Uploads with Doctrine

Handling file uploads with Doctrine entities is no different than handling any other file upload. In other words, you're free to move the file in your controller after handling a form submission. For examples of how to do this, see the file type reference page.

If you choose to, you can also integrate the file upload into your entity lifecycle (i.e. creation, update and removal). In this case, as your entity is created, updated, and removed from Doctrine, the file uploading and removal processing will take place automatically (without needing to do anything in your controller);

To make this work, you'll need to take care of a number of details, which will be covered in this cookbook entry.

Basic Setup

First, create a simple Doctrine Entity class to work with:

```
// src/Acme/DemoBundle/Entity/Document.php
namespace Acme\DemoBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
```

```
/**
 * @ORM\Entity
 */
class Document
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    public $id;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank
     */
    public $name;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    public $path;

    public function getAbsolutePath()
    {
        return null === $this->path ? null : $this->getUploadRootDir().'/'.$this->path;
    }

    public function getWebPath()
    {
        return null === $this->path ? null : $this->getUploadDir().'/'.$this->path;
    }

    protected function getUploadRootDir()
    {
        // the absolute directory path where uploaded documents should be saved
        return __DIR__.'/../../../../web/'.$this->getUploadDir();
    }

    protected function getUploadDir()
    {
        // get rid of the __DIR__ so it doesn't screw when displaying uploaded doc/image in the view
        return 'uploads/documents';
    }
}
```

The `Document` entity has a name and it is associated with a file. The `path` property stores the relative path to the file and is persisted to the database. The `getAbsolutePath()` is a convenience method that returns the absolute path to the file while the `getWebPath()` is a convenience method that returns the web path, which can be used in a template to link to the uploaded file.

Tip: If you have not done so already, you should probably read the file type documentation first to understand how the basic upload process works.

Note: If you're using annotations to specify your validation rules (as shown in this example), be sure that you've enabled validation by annotation (see *validation configuration*).

To handle the actual file upload in the form, use a "virtual" `file` field. For example, if you're building your form directly in a controller, it might look like this:

```
public function uploadAction()
{
    // ...

    $form = $this->createFormBuilder($document)
        ->add('name')
        ->add('file')
        ->getForm()
    ;

    // ...
}
```

Next, create this property on your `Document` class and add some validation rules:

```
// src/Acme/DemoBundle/Entity/Document.php

// ...
class Document
{
    /**
     * @Assert\File(maxSize="6000000")
     */
    public $file;

    // ...
}
```

Note: As you are using the `File` constraint, Symfony2 will automatically guess that the form field is a file upload input. That's why you did not have to set it explicitly when creating the form above (`->add('file')`).

The following controller shows you how to handle the entire process:

```
use Acme\DemoBundle\Entity\Document;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
// ...

/**
 * @Template()
 */
public function uploadAction()
{
    $document = new Document();
    $form = $this->createFormBuilder($document)
        ->add('name')
        ->add('file')
        ->getForm()
    ;

    if ($this->getRequest()->getMethod() === 'POST') {
        $form->bindRequest($this->getRequest());
        if ($form->isValid()) {
            $em = $this->getDoctrine()->getEntityManager();

            $em->persist($document);
            $em->flush();
```

```
            $this->redirect($this->generateUrl('...'));
        }
    }

    return array('form' => $form->createView());
}
```

Note: When writing the template, don't forget to set the `enctype` attribute:

```
<h1>Upload File</h1>

<form action="#" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" value="Upload Document" />
</form>
```

The previous controller will automatically persist the `Document` entity with the submitted name, but it will do nothing about the file and the `path` property will be blank.

An easy way to handle the file upload is to move it just before the entity is persisted and then set the `path` property accordingly. Start by calling a new `upload()` method on the `Document` class, which you'll create in a moment to handle the file upload:

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getEntityManager();

    $document->upload();

    $em->persist($document);
    $em->flush();

    $this->redirect('...');
}
```

The `upload()` method will take advantage of the `Symfony\Component\HttpFoundation\File\UploadedFile` object, which is what's returned after a `file` field is submitted:

```
public function upload()
{
    // the file property can be empty if the field is not required
    if (null === $this->file) {
        return;
    }

    // we use the original file name here but you should
    // sanitize it at least to avoid any security issues

    // move takes the target directory and then the target filename to move to
    $this->file->move($this->getUploadRootDir(), $this->file->getClientOriginalName());

    // set the path property to the filename where you've saved the file
    $this->path = $this->file->getClientOriginalName();

    // clean up the file property as you won't need it anymore
    $this->file = null;
}
```

Using Lifecycle Callbacks

Even if this implementation works, it suffers from a major flaw: What if there is a problem when the entity is persisted? The file would have already moved to its final location even though the entity's `path` property didn't persist correctly.

To avoid these issues, you should change the implementation so that the database operation and the moving of the file become atomic: if there is a problem persisting the entity or if the file cannot be moved, then *nothing* should happen.

To do this, you need to move the file right as Doctrine persists the entity to the database. This can be accomplished by hooking into an entity lifecycle callback:

```
/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
}
```

Next, refactor the `Document` class to take advantage of these callbacks:

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->file) {
            // do whatever you want to generate a unique name
            $this->path = uniqid().'.'.$this->file->guessExtension();
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->file) {
            return;
        }

        // if there is an error when moving the file, an exception will
        // be automatically thrown by move(). This will properly prevent
        // the entity from being persisted to the database on error
        $this->file->move($this->getUploadRootDir(), $this->path);

        unset($this->file);
    }
```

```
    /**
     * @ORM\PostRemove()
     */
    public function removeUpload()
    {
        if ($file = $this->getAbsolutePath()) {
            unlink($file);
        }
    }
}
```

The class now does everything you need: it generates a unique filename before persisting, moves the file after persisting, and removes the file if the entity is ever deleted.

Note: The `@ORM\PrePersist()` and `@ORM\PostPersist()` event callbacks are triggered before and after the entity is persisted to the database. On the other hand, the `@ORM\PreUpdate()` and `@ORM\PostUpdate()` event callbacks are called when the entity is updated.

Caution: The `PreUpdate` and `PostUpdate` callbacks are only triggered if there is a change in one of the entity's field that are persisted. This means that, by default, if you modify only the `$file` property, these events will not be triggered, as the property itself is not directly persisted via Doctrine. One solution would be to use an `updated` field that's persisted to Doctrine, and to modify it manually when changing the file.

Using the `id` as the filename

If you want to use the `id` as the name of the file, the implementation is slightly different as you need to save the extension under the `path` property, instead of the actual filename:

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->file) {
            $this->path = $this->file->guessExtension();
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->file) {
            return;
```

```
        }

        // you must throw an exception here if the file cannot be moved
        // so that the entity is not persisted to the database
        // which the UploadedFile move() method does
        $this->file->move($this->getUploadRootDir(), $this->id.'.'.$this->file->guessExtension());

        unset($this->file);
    }

    /**
     * @ORM\PostRemove()
     */
    public function removeUpload()
    {
        if ($file = $this->getAbsolutePath()) {
            unlink($file);
        }
    }

    public function getAbsolutePath()
    {
        return null === $this->path ? null : $this->getUploadRootDir().'/'.$this->id.'.'.$this->path;
    }
}
```

3.1.12 Doctrine Extensions: Timestampable: Sluggable, Translatable, etc.

Doctrine2 is very flexible, and the community has already created a series of useful Doctrine extensions to help you with tasks common entity-related tasks.

One bundle in particular - the DoctrineExtensionsBundle - provides integration with an extensions library that offers Sluggable, Translatable, Timestampable, Loggable, and Tree behaviors.

See the bundle for more details.

3.1.13 Registering Event Listeners and Subscribers

Doctrine packages a rich event system that fires events when almost anything happens inside the system. For you, this means that you can create arbitrary services and tell Doctrine to notify those objects whenever a certain action (e.g. `prePersist`) happens within Doctrine. This could be useful, for example, to create an independent search index whenever an object in your database is saved.

Doctrine defines two types of objects that can listen to Doctrine events: listeners and subscribers. Both are very similar, but listeners are a bit more straightforward. For more, see The Event System on Doctrine's website.

Configuring the Listener/Subscriber

To register a service to act as an event listener or subscriber you just have to *tag* it with the appropriate name. Depending on your use-case, you can hook a listener into every DBAL connection and ORM entity manager or just into one specific DBAL connection and all the entity managers that use this connection.

- *YAML*

```
    doctrine:
        dbal:
            default_connection: default
            connections:
                default:
                    driver: pdo_sqlite
                    memory: true

    services:
        my.listener:
            class: Acme\SearchBundle\Listener\SearchIndexer
            tags:
                - { name: doctrine.event_listener, event: postPersist }
        my.listener2:
            class: Acme\SearchBundle\Listener\SearchIndexer2
            tags:
                - { name: doctrine.event_listener, event: postPersist, connection: default }
        my.subscriber:
            class: Acme\SearchBundle\Listener\SearchIndexerSubscriber
            tags:
                - { name: doctrine.event_subscriber, connection: default }
```

- *XML*

```
    <?xml version="1.0" ?>
    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:doctrine="http://symfony.com/schema/dic/doctrine">

        <doctrine:config>
            <doctrine:dbal default-connection="default">
                <doctrine:connection driver="pdo_sqlite" memory="true" />
            </doctrine:dbal>
        </doctrine:config>

        <services>
            <service id="my.listener" class="Acme\SearchBundle\Listener\SearchIndexer">
                <tag name="doctrine.event_listener" event="postPersist" />
            </service>
            <service id="my.listener2" class="Acme\SearchBundle\Listener\SearchIndexer2">
                <tag name="doctrine.event_listener" event="postPersist" connection="default" />
            </service>
            <service id="my.subscriber" class="Acme\SearchBundle\Listener\SearchIndexerSubscriber">
                <tag name="doctrine.event_subscriber" connection="default" />
            </service>
        </services>
    </container>
```

Creating the Listener Class

In the previous example, a service `my.listener` was configured as a Doctrine listener on the event `postPersist`. That class behind that service must have a `postPersist` method, which will be called when the event is thrown:

```
// src/Acme/SearchBundle/Listener/SearchIndexer.php
namespace Acme\SearchBundle\Listener;

use Doctrine\ORM\Event\LifecycleEventArgs;
use Acme\StoreBundle\Entity\Product;
```

```
class SearchIndexer
{
    public function postPersist(LifecycleEventArgs $args)
    {
        $entity = $args->getEntity();
        $entityManager = $args->getEntityManager();

        // perhaps you only want to act on some "Product" entity
        if ($entity instanceof Product) {
            // do something with the Product
        }
    }
}
```

In each event, you have access to a `LifecycleEventArgs` object, which gives you access to both the entity object of the event and the entity manager itself.

One important thing to notice is that a listener will be listening for *all* entities in your application. So, if you're interested in only handling a specific type of entity (e.g. a `Product` entity but not a `BlogPost` entity), you should check for the class name of the entity in your method (as shown above).

3.1.14 How to generate Entities from an Existing Database

When starting work on a brand new project that uses a database, two different situations comes naturally. In most cases, the database model is designed and built from scratch. Sometimes, however, you'll start with an existing and probably unchangeable database model. Fortunately, Doctrine comes with a bunch of tools to help generate model classes from your existing database.

Note: As the Doctrine tools documentation says, reverse engineering is a one-time process to get started on a project. Doctrine is able to convert approximately 70-80% of the necessary mapping information based on fields, indexes and foreign key constraints. Doctrine can't discover inverse associations, inheritance types, entities with foreign keys as primary keys or semantical operations on associations such as cascade or lifecycle events. Some additional work on the generated entities will be necessary afterwards to design each to fit your domain model specificities.

This tutorial assumes you're using a simple blog application with the following two tables: `blog_post` and `blog_comment`. A comment record is linked to a post record thanks to a foreign key constraint.

```
CREATE TABLE `blog_post` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `title` varchar(100) COLLATE utf8_unicode_ci NOT NULL,
  `content` longtext COLLATE utf8_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL,
  PRIMARY KEY (`id`),
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

CREATE TABLE `blog_comment` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `post_id` bigint(20) NOT NULL,
  `author` varchar(20) COLLATE utf8_unicode_ci NOT NULL,
  `content` longtext COLLATE utf8_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL,
  PRIMARY KEY (`id`),
  KEY `blog_comment_post_id_idx` (`post_id`),
  CONSTRAINT `blog_post_id` FOREIGN KEY (`post_id`) REFERENCES `blog_post` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Before diving into the recipe, be sure your database connection parameters are correctly setup in the `app/config/parameters.yml` file (or wherever your database configuration is kept) and that you have initialized a bundle that will host your future entity class. In this tutorial, we will assume that an `AcmeBlogBundle` exists and is located under the `src/Acme/BlogBundle` folder.

The first step towards building entity classes from an existing database is to ask Doctrine to introspect the database and generate the corresponding metadata files. Metadata files describe the entity class to generate based on tables fields.

```
php app/console doctrine:mapping:convert xml ./src/Acme/BlogBundle/Resources/config/doctrine/metadata
```

This command line tool asks Doctrine to introspect the database and generate the XML metadata files under the `src/Acme/BlogBundle/Resources/config/doctrine/metadata/orm` folder of your bundle.

Tip: It's also possible to generate metadata class in YAML format by changing the first argument to *yml*.

The generated `BlogPost.dcm.xml` metadata file looks as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping>
  <entity name="BlogPost" table="blog_post">
    <change-tracking-policy>DEFERRED_IMPLICIT</change-tracking-policy>
    <id name="id" type="bigint" column="id">
      <generator strategy="IDENTITY"/>
    </id>
    <field name="title" type="string" column="title" length="100"/>
    <field name="content" type="text" column="content"/>
    <field name="isPublished" type="boolean" column="is_published"/>
    <field name="createdAt" type="datetime" column="created_at"/>
    <field name="updatedAt" type="datetime" column="updated_at"/>
    <field name="slug" type="string" column="slug" length="255"/>
    <lifecycle-callbacks/>
  </entity>
</doctrine-mapping>
```

Once the metadata files are generated, you can ask Doctrine to import the schema and build related entity classes by executing the following two commands.

```
php app/console doctrine:mapping:import AcmeBlogBundle annotation
php app/console doctrine:generate:entities AcmeBlogBundle
```

The first command generates entity classes with an annotations mapping, but you can of course change the `annotation` argument to `xml` or `yml`. The newly created `BlogComment` entity class looks as follow:

```
<?php

// src/Acme/BlogBundle/Entity/BlogComment.php
namespace Acme\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Acme\BlogBundle\Entity\BlogComment
 *
 * @ORM\Table(name="blog_comment")
 * @ORM\Entity
 */
class BlogComment
{
    /**
```

```
     * @var bigint $id
     *
     * @ORM\Column(name="id", type="bigint", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;

    /**
     * @var string $author
     *
     * @ORM\Column(name="author", type="string", length=100, nullable=false)
     */
    private $author;

    /**
     * @var text $content
     *
     * @ORM\Column(name="content", type="text", nullable=false)
     */
    private $content;

    /**
     * @var datetime $createdAt
     *
     * @ORM\Column(name="created_at", type="datetime", nullable=false)
     */
    private $createdAt;

    /**
     * @var BlogPost
     *
     * @ORM\ManyToOne(targetEntity="BlogPost")
     * @ORM\JoinColumn(name="post_id", referencedColumnName="id")
     */
    private $post;
}
```

As you can see, Doctrine converts all table fields to pure private and annotated class properties. The most impressive thing is that it also discovered the relationship with the `BlogPost` entity class based on the foreign key constraint. Consequently, you can find a private `$post` property mapped with a `BlogPost` entity in the `BlogComment` entity class.

The last command generated all getters and setters for your two `BlogPost` and `BlogComment` entity class properties. The generated entities are now ready to be used. Have fun!

3.1.15 How to use Doctrine's DBAL Layer

Note: This article is about Doctrine DBAL's layer. Typically, you'll work with the higher level Doctrine ORM layer, which simply uses the DBAL behind the scenes to actually communicate with the database. To read more about the Doctrine ORM, see "Databases and Doctrine ("The Model")".

The Doctrine Database Abstraction Layer (DBAL) is an abstraction layer that sits on top of PDO and offers an intuitive and flexible API for communicating with the most popular relational databases. In other words, the DBAL library makes it easy to execute queries and perform other database actions.

Tip: Read the official Doctrine DBAL Documentation to learn all the details and capabilities of Doctrine's DBAL library.

To get started, configure the database connection parameters:

- *YAML*

```
# app/config/config.yml
doctrine:
    dbal:
        driver:   pdo_mysql
        dbname:   Symfony2
        user:     root
        password: null
        charset:  UTF8
```

- *XML*

```
// app/config/config.xml
<doctrine:config>
    <doctrine:dbal
        name="default"
        dbname="Symfony2"
        user="root"
        password="null"
        driver="pdo_mysql"
    />
</doctrine:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('doctrine', array(
    'dbal' => array(
        'driver'    => 'pdo_mysql',
        'dbname'    => 'Symfony2',
        'user'      => 'root',
        'password'  => null,
    ),
));
```

For full DBAL configuration options, see *Doctrine DBAL Configuration*.

You can then access the Doctrine DBAL connection by accessing the `database_connection` service:

```
class UserController extends Controller
{
    public function indexAction()
    {
        $conn = $this->get('database_connection');
        $users = $conn->fetchAll('SELECT * FROM users');

        // ...
    }
}
```

Registering Custom Mapping Types

You can register custom mapping types through Symfony's configuration. They will be added to all configured connections. For more information on custom mapping types, read Doctrine's Custom Mapping Types section of their documentation.

- *YAML*

```
# app/config/config.yml
doctrine:
    dbal:
        types:
            custom_first: Acme\HelloBundle\Type\CustomFirst
            custom_second: Acme\HelloBundle\Type\CustomSecond
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

    <doctrine:config>
        <doctrine:dbal>
        <doctrine:dbal default-connection="default">
            <doctrine:connection>
                <doctrine:mapping-type name="enum">string</doctrine:mapping-type>
            </doctrine:connection>
        </doctrine:dbal>
    </doctrine:config>
</container>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('doctrine', array(
    'dbal' => array(
        'connections' => array(
            'default' => array(
                'mapping_types' => array(
                    'enum'  => 'string',
                ),
            ),
        ),
    ),
));
```

Registering Custom Mapping Types in the SchemaTool

The SchemaTool is used to inspect the database to compare the schema. To achieve this task, it needs to know which mapping type needs to be used for each database types. Registering new ones can be done through the configuration.

Let's map the ENUM type (not suppoorted by DBAL by default) to a the `string` mapping type:

- *YAML*

```
# app/config/config.yml
doctrine:
    dbal:
        connections:
            default:
                // Other connections parameters
                mapping_types:
                    enum: string
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

    <doctrine:config>
        <doctrine:dbal>
            <doctrine:type name="custom_first" class="Acme\HelloBundle\Type\CustomFirst" />
            <doctrine:type name="custom_second" class="Acme\HelloBundle\Type\CustomSecond" />
        </doctrine:dbal>
    </doctrine:config>
</container>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('doctrine', array(
    'dbal' => array(
        'types' => array(
            'custom_first'  => 'Acme\HelloBundle\Type\CustomFirst',
            'custom_second' => 'Acme\HelloBundle\Type\CustomSecond',
        ),
    ),
));
```

3.1.16 How to work with Multiple Entity Managers

You can use multiple entity managers in a Symfony2 application. This is necessary if you are using different databases or even vendors with entirely different sets of entities. In other words, one entity manager that connects to one database will handle some entities while another entity manager that connects to another database might handle the rest.

Note: Using multiple entity managers is pretty easy, but more advanced and not usually required. Be sure you actually need multiple entity managers before adding in this layer of complexity.

The following configuration code shows how you can configure two entity managers:

- *YAML*

```
doctrine:
    orm:
        default_entity_manager:   default
        entity_managers:
            default:
                connection:       default
```

```
                    mappings:
                        AcmeDemoBundle: ~
                        AcmeStoreBundle: ~
                customer:
                    connection:       customer
                    mappings:
                        AcmeCustomerBundle: ~
```

In this case, you've defined two entity managers and called them `default` and `customer`. The `default` entity manager manages entities in the `AcmeDemoBundle` and `AcmeStoreBundle`, while the `customer` entity manager manages entities in the `AcmeCustomerBundle`.

When working with multiple entity managers, you should be explicit about which entity manager you want. If you *do* omit the entity manager's name when asking for it, the default entity manager (i.e. `default`) is returned:

```
class UserController extends Controller
{
    public function indexAction()
    {
        // both return the "default" em
        $em = $this->get('doctrine')->getEntityManager();
        $em = $this->get('doctrine')->getEntityManager('default');

        $customerEm =  $this->get('doctrine')->getEntityManager('customer');
    }
}
```

You can now use Doctrine just as you did before - using the `default` entity manager to persist and fetch entities that it manages and the `customer` entity manager to persist and fetch its entities.

3.1.17 Registering Custom DQL Functions

Doctrine allows you to specify custom DQL functions. For more information on this topic, read Doctrine's cookbook article "DQL User Defined Functions".

In Symfony, you can register your custom DQL functions as follows:

- *YAML*

```
    # app/config/config.yml
    doctrine:
        orm:
            # ...
            entity_managers:
                default:
                    # ...
                    dql:
                        string_functions:
                            test_string: Acme\HelloBundle\DQL\StringFunction
                            second_string: Acme\HelloBundle\DQL\SecondStringFunction
                        numeric_functions:
                            test_numeric: Acme\HelloBundle\DQL\NumericFunction
                        datetime_functions:
                            test_datetime: Acme\HelloBundle\DQL\DatetimeFunction
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

    <doctrine:config>
        <doctrine:orm>
            <!-- ... -->
            <doctrine:entity-manager name="default">
                <!-- ... -->
                <doctrine:dql>
                    <doctrine:string-function name="test_string>Acme\HelloBundle\DQL\StringFunct
                    <doctrine:string-function name="second_string>Acme\HelloBundle\DQL\SecondStr
                    <doctrine:numeric-function name="test_numeric>Acme\HelloBundle\DQL\NumericFu
                    <doctrine:datetime-function name="test_datetime>Acme\HelloBundle\DQL\Datetim
                </doctrine:dql>
            </doctrine:entity-manager>
        </doctrine:orm>
    </doctrine:config>
</container>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('doctrine', array(
    'orm' => array(
        // ...
        'entity_managers' => array(
            'default' => array(
                // ...
                'dql' => array(
                    'string_functions' => array(
                        'test_string'   => 'Acme\HelloBundle\DQL\StringFunction',
                        'second_string' => 'Acme\HelloBundle\DQL\SecondStringFunction',
                    ),
                    'numeric_functions' => array(
                        'test_numeric' => 'Acme\HelloBundle\DQL\NumericFunction',
                    ),
                    'datetime_functions' => array(
                        'test_datetime' => 'Acme\HelloBundle\DQL\DatetimeFunction',
                    ),
                ),
            ),
        ),
    ),
));
```

3.1.18 How to customize Form Rendering

Symfony gives you a wide variety of ways to customize how a form is rendered. In this guide, you'll learn how to customize every possible part of your form with as little effort as possible whether you use Twig or PHP as your templating engine.

Form Rendering Basics

Recall that the label, error and HTML widget of a form field can easily be rendered by using the `form_row` Twig function or the `row` PHP helper method:

- *Twig*

```
{{ form_row(form.age) }}
```

- *PHP*

```
<?php echo $view['form']->row($form['age']) }} ?>
```

You can also render each of the three parts of the field individually:

- *Twig*

```
<div>
    {{ form_label(form.age) }}
    {{ form_errors(form.age) }}
    {{ form_widget(form.age) }}
</div>
```

- *PHP*

```
<div>
    <?php echo $view['form']->label($form['age']) }} ?>
    <?php echo $view['form']->errors($form['age']) }} ?>
    <?php echo $view['form']->widget($form['age']) }} ?>
</div>
```

In both cases, the form label, errors and HTML widget are rendered by using a set of markup that ships standard with Symfony. For example, both of the above templates would render:

```
<div>
    <label for="form_age">Age</label>
    <ul>
        <li>This field is required</li>
    </ul>
    <input type="number" id="form_age" name="form[age]" />
</div>
```

To quickly prototype and test a form, you can render the entire form with just one line:

- *Twig*

```
{{ form_widget(form) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form) }} ?>
```

The remainder of this recipe will explain how every part of the form's markup can be modified at several different levels. For more information about form rendering in general, see *Rendering a Form in a Template*.

What are Form Themes?

Symfony uses form fragments - a small piece of a template that renders just one part of a form - to render every part of a form - - field labels, errors, `input` text fields, `select` tags, etc

The fragments are defined as blocks in Twig and as template files in PHP.

A *theme* is nothing more than a set of fragments that you want to use when rendering a form. In other words, if you want to customize one portion of how a form is rendered, you'll import a *theme* which contains a customization of the appropriate form fragments.

Symfony comes with a default theme (form_div_layout.html.twig in Twig and `FrameworkBundle:Form` in PHP) that defines each and every fragment needed to render every part of a form.

In the next section you will learn how to customize a theme by overriding some or all of its fragments.

For example, when the widget of a `integer` type field is rendered, an `input number` field is generated

- *Twig*

```
{{ form_widget(form.age) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['age']) ?>
```

renders:

```
<input type="number" id="form_age" name="form[age]" required="required" value="33" />
```

Internally, Symfony uses the `integer_widget` fragment to render the field. This is because the field type is `integer` and you're rendering its `widget` (as opposed to its `label` or `errors`).

In Twig that would default to the block `integer_widget` from the form_div_layout.html.twig template.

In PHP it would rather be the `integer_widget.html.php` file located in `FrameworkBundle/Resources/views/Form` folder.

The default implementation of the `integer_widget` fragment looks like this:

- *Twig*

```
{% block integer_widget %}
    {% set type = type|default('number') %}
    {{ block('field_widget') }}
{% endblock integer_widget %}
```

- *PHP*

```
<!-- integer_widget.html.php -->

<?php echo $view['form']->renderBlock('field_widget', array('type' => isset($type) ? $type : "nu
```

As you can see, this fragment itself renders another fragment - `field_widget`:

- *Twig*

```
{% block field_widget %}
    {% set type = type|default('text') %}
    <input type="{{ type }}" {{ block('widget_attributes') }} value="{{ value }}" />
{% endblock field_widget %}
```

- *PHP*

```
<!-- FrameworkBundle/Resources/views/Form/field_widget.html.php -->

<input
    type="<?php echo isset($type) ? $view->escape($type) : "text" ?>"
    value="<?php echo $view->escape($value) ?>"
    <?php echo $view['form']->renderBlock('attributes') ?>
/>
```

The point is, the fragments dictate the HTML output of each part of a form. To customize the form output, you just need to identify and override the correct fragment. A set of these form fragment customizations is known as a form "theme". When rendering a form, you can choose which form theme(s) you want to apply.

In Twig a theme is a single template file and the fragments are the blocks defined in this file.

In PHP a theme is a folder and the the fragments are individual template files in this folder.

Knowing which block to customize

In this example, the customized fragment name is `integer_widget` because you want to override the HTML `widget` for all `integer` field types. If you need to customize textarea fields, you would customize `textarea_widget`.
As you can see, the fragment name is a combination of the field type and which part of the field is being rendered (e.g. `widget`, `label`, `errors`, `row`). As such, to customize how errors are rendered for just input `text` fields, you should customize the `text_errors` fragment.
More commonly, however, you'll want to customize how errors are displayed across *all* fields. You can do this by customizing the `field_errors` fragment. This takes advantage of field type inheritance. Specifically, since the `text` type extends from the `field` type, the form component will first look for the type-specific fragment (e.g. `text_errors`) before falling back to its parent fragment name if it doesn't exist (e.g. `field_errors`).
For more information on this topic, see *Form Fragment Naming*.

Form Theming

To see the power of form theming, suppose you want to wrap every input `number` field with a `div` tag. The key to doing this is to customize the `integer_widget` fragment.

Form Theming in Twig

When customizing the form field block in Twig, you have two options on *where* the customized form block can live:

Method	Pros	Cons
Inside the same template as the form	Quick and easy	Can't be reused in other templates
Inside a separate template	Can be reused by many templates	Requires an extra template to be created

Both methods have the same effect but are better in different situations.

Method 1: Inside the same Template as the Form

The easiest way to customize the `integer_widget` block is to customize it directly in the template that's actually rendering the form.

```
{% extends '::base.html.twig' %}

{% form_theme form _self %}

{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('field_widget') }}
    </div>
```

```
{% endblock %}

{% block content %}
    {# render the form #}

    {{ form_row(form.age) }}
{% endblock %}
```

By using the special `{% form_theme form _self %}` tag, Twig looks inside the same template for any overridden form blocks. Assuming the `form.age` field is an `integer` type field, when its widget is rendered, the customized `integer_widget` block will be used.

The disadvantage of this method is that the customized form block can't be reused when rendering other forms in other templates. In other words, this method is most useful when making form customizations that are specific to a single form in your application. If you want to reuse a form customization across several (or all) forms in your application, read on to the next section.

Method 2: Inside a Separate Template

You can also choose to put the customized `integer_widget` form block in a separate template entirely. The code and end-result are the same, but you can now re-use the form customization across many templates:

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}

{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('field_widget') }}
    </div>
{% endblock %}
```

Now that you've created the customized form block, you need to tell Symfony to use it. Inside the template where you're actually rendering your form, tell Symfony to use the template via the `form_theme` tag:

```
{% form_theme form 'AcmeDemoBundle:Form:fields.html.twig' %}

{{ form_widget(form.age) }}
```

When the `form.age` widget is rendered, Symfony will use the `integer_widget` block from the new template and the `input` tag will be wrapped in the `div` element specified in the customized block.

Form Theming in PHP

When using PHP as a templating engine, the only method to customize a fragment is to create a new template file - this is similar to the second method used by Twig.

The template file must be named after the fragment. You must create a `integer_widget.html.php` file in order to customize the `integer_widget` fragment.

```
<!-- src/Acme/DemoBundle/Resources/views/Form/integer_widget.html.php -->

<div class="integer_widget">
    <?php echo $view['form']->renderBlock('field_widget', array('type' => isset($type) ? $type : "nur
</div>
```

Now that you've created the customized form template, you need to tell Symfony to use it. Inside the template where you're actually rendering your form, tell Symfony to use the theme via the `setTheme` helper method:

```
<?php $view['form']->setTheme($form, array('AcmeDemoBundle:Form')) ;?>

<?php $view['form']->widget($form['age']) ?>
```

When the `form.age` widget is rendered, Symfony will use the customized `integer_widget.html.php` template and the `input` tag will be wrapped in the `div` element.

Referencing Base Form Blocks (Twig specific)

So far, to override a particular form block, the best method is to copy the default block from form_div_layout.html.twig, paste it into a different template, and the customize it. In many cases, you can avoid doing this by referencing the base block when customizing it.

This is easy to do, but varies slightly depending on if your form block customizations are in the same template as the form or a separate template.

Referencing Blocks from inside the same Template as the Form

Import the blocks by adding a `use` tag in the template where you're rendering the form:

```
{% use 'form_div_layout.html.twig' with integer_widget as base_integer_widget %}
```

Now, when the blocks from form_div_layout.html.twig are imported, the `integer_widget` block is called `base_integer_widget`. This means that when you redefine the `integer_widget` block, you can reference the default markup via `base_integer_widget`:

```
{% block integer_widget %}
    <div class="integer_widget">
        {{ block('base_integer_widget') }}
    </div>
{% endblock %}
```

Referencing Base Blocks from an External Template

If your form customizations live inside an external template, you can reference the base block by using the `parent()` Twig function:

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}

{% extends 'form_div_layout.html.twig' %}

{% block integer_widget %}
    <div class="integer_widget">
        {{ parent() }}
    </div>
{% endblock %}
```

Note: It is not possible to reference the base block when using PHP as the templating engine. You have to manually copy the content from the base block to your new template file.

Making Application-wide Customizations

If you'd like a certain form customization to be global to your application, you can accomplish this by making the form customizations in an external template and then importing it inside your application configuration:

Twig

By using the following configuration, any customized form blocks inside the `AcmeDemoBundle:Form:fields.html.twig` template will be used globally when a form is rendered.

- *YAML*

```
# app/config/config.yml

twig:
    form:
        resources:
            - 'AcmeDemoBundle:Form:fields.html.twig'
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->

<twig:config ...>
        <twig:form>
            <resource>AcmeDemoBundle:Form:fields.html.twig</resource>
        </twig:form>
        <!-- ... -->
</twig:config>
```

- *PHP*

```
// app/config/config.php

$container->loadFromExtension('twig', array(
    'form' => array('resources' => array(
        'AcmeDemoBundle:Form:fields.html.twig',
     ))
    // ...
));
```

By default, Twig uses a *div* layout when rendering forms. Some people, however, may prefer to render forms in a *table* layout. Use the `form_table_layout.html.twig` resource to use such a layout:

- *YAML*

```
# app/config/config.yml

twig:
    form:
        resources: ['form_table_layout.html.twig']
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->

<twig:config ...>
```

```
            <twig:form>
                <resource>form_table_layout.html.twig</resource>
            </twig:form>
            <!-- ... -->
    </twig:config>
```

- *PHP*

```
    // app/config/config.php

    $container->loadFromExtension('twig', array(
        'form' => array('resources' => array(
            'form_table_layout.html.twig',
         ))
        // ...
    ));
```

If you only want to make the change in one template, add the following line to your template file rather than adding the template as a resource:

```
{% form_theme form 'form_table_layout.html.twig' %}
```

Note that the `form` variable in the above code is the form view variable that you passed to your template.

PHP

By using the following configuration, any customized form fragments inside the `src/Acme/DemoBundle/Resources/views/Form` folder will be used globally when a form is rendered.

- *YAML*

```
    # app/config/config.yml

    framework:
        templating:
            form:
                resources:
                    - 'AcmeDemoBundle:Form'
        # ...
```

- *XML*

```
    <!-- app/config/config.xml -->

    <framework:config ...>
        <framework:templating>
            <framework:form>
                <resource>AcmeDemoBundle:Form</resource>
            </framework:form>
        </framework:templating>
        <!-- ... -->
    </framework:config>
```

- *PHP*

```
    // app/config/config.php

    // PHP
    $container->loadFromExtension('framework', array(
```

```
        'templating' => array('form' =>
            array('resources' => array(
                'AcmeDemoBundle:Form',
         )))
        // ...
    ));
```

By default, the PHP engine uses a *div* layout when rendering forms. Some people, however, may prefer to render forms in a *table* layout. Use the `FrameworkBundle:FormTable` resource to use such a layout:

- *YAML*

```
# app/config/config.yml

framework:
    templating:
        form:
            resources:
                - 'FrameworkBundle:FormTable'
```

- *XML*

```
<!-- app/config/config.xml -->

<framework:config ...>
    <framework:templating>
        <framework:form>
            <resource>FrameworkBundle:FormTable</resource>
        </framework:form>
    </framework:templating>
    <!-- ... -->
</framework:config>
```

- *PHP*

```
// app/config/config.php

$container->loadFromExtension('framework', array(
    'templating' => array('form' =>
        array('resources' => array(
            'FrameworkBundle:FormTable',
     )))
    // ...
));
```

If you only want to make the change in one template, add the following line to your template file rather than adding the template as a resource:

```
<?php $view['form']->setTheme($form, array('FrameworkBundle:FormTable')); ?>
```

Note that the `$form` variable in the above code is the form view variable that you passed to your template.

How to customize an Individual field

So far, you've seen the different ways you can customize the widget output of all text field types. You can also customize individual fields. For example, suppose you have two `text` fields - `first_name` and `last_name` - but you only want to customize one of the fields. This can be accomplished by customizing a fragment whose name is a combination of the field id attribute and which part of the field is being customized. For example:

- *Twig*

```
{% form_theme form _self %}

{% block _product_name_widget %}
    <div class="text_widget">
        {{ block('field_widget') }}
    </div>
{% endblock %}

{{ form_widget(form.name) }}
```

- *PHP*

```
<!-- Main template -->

<?php echo $view['form']->setTheme($form, array('AcmeDemoBundle:Form')); ?>

<?php echo $view['form']->widget($form['name']); ?>

<!-- src/Acme/DemoBundle/Resources/views/Form/_product_name_widget.html.php -->

<div class="text_widget">
      echo $view['form']->renderBlock('field_widget') ?>
</div>
```

Here, the `_product_name_widget` fragment defines the template to use for the field whose *id* is `product_name` (and name is `product[name]`).

Tip: The `product` portion of the field is the form name, which may be set manually or generated automatically based on your form type name (e.g. `ProductType` equates to `product`). If you're not sure what your form name is, just view the source of your generated form.

You can also override the markup for an entire field row using the same method:

- *Twig*

```
{% form_theme form _self %}

{% block _product_name_row %}
    <div class="name_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock %}
```

- *PHP*

```
<!-- _product_name_row.html.php -->

<div class="name_row">
    <?php echo $view['form']->label($form) ?>
    <?php echo $view['form']->errors($form) ?>
    <?php echo $view['form']->widget($form) ?>
</div>
```

Other Common Customizations

So far, this recipe has shown you several different ways to customize a single piece of how a form is rendered. The key is to customize a specific fragment that corresponds to the portion of the form you want to control (see *naming form blocks*).

In the next sections, you'll see how you can make several common form customizations. To apply these customizations, use one of the methods described in the *Form Theming* section.

Customizing Error Output

Note: The form component only handles *how* the validation errors are rendered, and not the actual validation error messages. The error messages themselves are determined by the validation constraints you apply to your objects. For more information, see the chapter on validation.

There are many different ways to customize how errors are rendered when a form is submitted with errors. The error messages for a field are rendered when you use the `form_errors` helper:

- *Twig*

```
{{ form_errors(form.age) }}
```

- *PHP*

```
<?php echo $view['form']->errors($form['age']); ?>
```

By default, the errors are rendered inside an unordered list:

```
<ul>
    <li>This field is required</li>
</ul>
```

To override how errors are rendered for *all* fields, simply copy, paste and customize the `field_errors` fragment.

- *Twig*

```
{% block field_errors %}
{% spaceless %}
    {% if errors|length > 0 %}
    <ul class="error_list">
        {% for error in errors %}
            <li>{{ error.messageTemplate|trans(error.messageParameters, 'validators') }}</li>
        {% endfor %}
    </ul>
    {% endif %}
{% endspaceless %}
{% endblock field_errors %}
```

- *PHP*

```
<!-- fields_errors.html.php -->

<?php if (count($errors) > 0): ?>
    <ul class="error_list">
        <?php foreach ($errors as $error): ?>
            <li><?php echo $view['translator']->trans(
                $error->getMessageTemplate(),
                $error->getMessageParameters(),
```

```
                    'validators'
                ) ?></li>
            <?php endforeach; ?>
        </ul>
    <?php endif ?>
```

Tip: See *Form Theming* for how to apply this customization.

You can also customize the error output for just one specific field type. For example, certain errors that are more global to your form (i.e. not specific to just one field) are rendered separately, usually at the top of your form:

- *Twig*

```
{{ form_errors(form) }}
```

- *PHP*

```
<?php echo $view['form']->render($form); ?>
```

To customize *only* the markup used for these errors, follow the same directions as above, but now call the block `form_errors` (Twig) / the file `form_errors.html.php` (PHP). Now, when errors for the `form` type are rendered, your customized fragment will be used instead of the default `field_errors`.

Customizing the "Form Row"

When you can manage it, the easiest way to render a form field is via the `form_row` function, which renders the label, errors and HTML widget of a field. To customize the markup used for rendering *all* form field rows, override the `field_row` fragment. For example, suppose you want to add a class to the `div` element around each row:

- *Twig*

```
{% block field_row %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock field_row %}
```

- *PHP*

```
<!-- field_row.html.php -->

<div class="form_row">
    <?php echo $view['form']->label($form) ?>
    <?php echo $view['form']->errors($form) ?>
    <?php echo $view['form']->widget($form) ?>
</div>
```

Tip: See *Form Theming* for how to apply this customization.

Adding a "Required" Asterisk to Field Labels

If you want to denote all of your required fields with a required asterisk (`*`), you can do this by customizing the `field_label` fragment.

In Twig, if you're making the form customization inside the same template as your form, modify the `use` tag and add the following:

```
{% use 'form_div_layout.html.twig' with field_label as base_field_label %}

{% block field_label %}
    {{ block('base_field_label') }}

    {% if required %}
        <span class="required" title="This field is required">*</span>
    {% endif %}
{% endblock %}
```

In Twig, if you're making the form customization inside a separate template, use the following:

```
{% extends 'form_div_layout.html.twig' %}

{% block field_label %}
    {{ parent() }}

    {% if required %}
        <span class="required" title="This field is required">*</span>
    {% endif %}
{% endblock %}
```

When using PHP as a templating engine you have to copy the content from the original template:

```
<!-- field_label.html.php -->

<!-- original content -->
<label for="<?php echo $view->escape($id) ?>" <?php foreach($attr as $k => $v) { printf('%s="%s" ', $

<!-- customization -->
<?php if ($required) : ?>
    <span class="required" title="This field is required">*</span>
<?php endif ?>
```

Tip: See *Form Theming* for how to apply this customization.

Adding "help" messages

You can also customize your form widgets to have an optional "help" message.

In Twig, If you're making the form customization inside the same template as your form, modify the `use` tag and add the following:

```
{% use 'form_div_layout.html.twig' with field_widget as base_field_widget %}

{% block field_widget %}
    {{ block('base_field_widget') }}

    {% if help is defined %}
        <span class="help">{{ help }}</span>
    {% endif %}
{% endblock %}
```

In twig, If you're making the form customization inside a separate template, use the following:

```
{% extends 'form_div_layout.html.twig' %}

{% block field_widget %}
    {{ parent() }}

    {% if help is defined %}
        <span class="help">{{ help }}</span>
    {% endif %}
{% endblock %}
```

When using PHP as a templating engine you have to copy the content from the original template:

```
<!-- field_widget.html.php -->

<!-- Original content -->
<input
    type="<?php echo isset($type) ? $view->escape($type) : "text" ?>"
    value="<?php echo $view->escape($value) ?>"
    <?php echo $view['form']->renderBlock('attributes') ?>
/>

<!-- Customization -->
<?php if (isset($help)) : ?>
    <span class="help"><?php echo $view->escape($help) ?></span>
<?php endif ?>
```

To render a help message below a field, pass in a `help` variable:

- *Twig*

```
    {{ form_widget(form.title, { 'help': 'foobar' }) }}
```

- *PHP*

```
    <?php echo $view['form']->widget($form['title'], array('help' => 'foobar')) ?>
```

Tip: See *Form Theming* for how to apply this customization.

3.1.19 Using Data Transformers

You'll often find the need to transform the data the user entered in a form into something else for use in your program. You could easily do this manually in your controller, but what if you want to use this specific form in different places?

Say you have a one-to-one relation of Task to Issue, e.g. a Task optionally has an issue linked to it. Adding a listbox with all possible issues can eventually lead to a really long listbox in which it is impossible to find something. You'll rather want to add a textbox, in which the user can simply enter the number of the issue. In the controller you can convert this issue number to an actual task, and eventually add errors to the form if it was not found, but of course this is not really clean.

It would be better if this issue was automatically looked up and converted to an Issue object, for use in your action. This is where Data Transformers come into play.

First, create a custom form type which has a Data Transformer attached to it, which returns the Issue by number: the issue selector type. Eventually this will simply be a text field, as we configure the fields' parent to be a "text" field, in which you will enter the issue number. The field will display an error if a non existing number was entered:

```
// src/Acme/TaskBundle/Form/IssueSelectorType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;
use Acme\TaskBundle\Form\DataTransformer\IssueToNumberTransformer;
use Doctrine\Common\Persistence\ObjectManager;

class IssueSelectorType extends AbstractType
{
    private $om;

    public function __construct(ObjectManager $om)
    {
        $this->om = $om;
    }

    public function buildForm(FormBuilder $builder, array $options)
    {
        $transformer = new IssueToNumberTransformer($this->om);
        $builder->appendClientTransformer($transformer);
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'invalid_message'=>'The selected issue does not exist'
        );
    }

    public function getParent(array $options)
    {
        return 'text';
    }

    public function getName()
    {
        return 'issue_selector';
    }
}
```

Tip: You can also use transformers without creating a new custom form type by calling `appendClientTransformer` on any field builder:

```
use Acme\TaskBundle\Form\DataTransformer\IssueToNumberTransformer;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        // ...

        // this assumes that the entity manager was passed in as an option
        $entityManager = $options['em'];
        $transformer = new IssueToNumberTransformer($entityManager);

        // use a normal text field, but transform the text into an issue object
        $builder
```

```
            ->add('issue', 'text')
            ->appendClientTransformer($transformer)
        ;
    }

    // ...
}
```

Next, we create the data transformer, which does the actual conversion:

```
// src/Acme/TaskBundle/Form/DataTransformer/IssueToNumberTransformer.php
namespace Acme\TaskBundle\Form\DataTransformer;

use Symfony\Component\Form\Exception\TransformationFailedException;
use Symfony\Component\Form\DataTransformerInterface;
use Doctrine\Common\Persistence\ObjectManager;

class IssueToNumberTransformer implements DataTransformerInterface
{
    private $om;

    public function __construct(ObjectManager $om)
    {
        $this->om = $om;
    }

    // transforms the Issue object to a string
    public function transform($val)
    {
        if (null === $val) {
            return '';
        }

        return $val->getNumber();
    }

    // transforms the issue number into an Issue object
    public function reverseTransform($val)
    {
        if (!$val) {
            return null;
        }

        $issue = $this->om->getRepository('AcmeTaskBundle:Issue')->findOneBy(array('number' => $val))

        if (null === $issue) {
            throw new TransformationFailedException(sprintf('An issue with number %s does not exist!'

        }

        return $issue;
    }
}
```

Finally, since we've decided to create a custom form type that uses the data transformer, register the Type in the service container, so that the entity manager can be automatically injected:

- *YAML*

```
    services:
        acme_demo.type.issue_selector:
            class: Acme\TaskBundle\Form\IssueSelectorType
            arguments: ["@doctrine.orm.entity_manager"]
            tags:
                - { name: form.type, alias: issue_selector }
```

- *XML*

```
    <service id="acme_demo.type.issue_selector" class="Acme\TaskBundle\Form\IssueSelectorType">
        <argument type="service" id="doctrine.orm.entity_manager"/>
        <tag name="form.type" alias="issue_selector" />
    </service>
```

You can now add the type to your form by its alias as follows:

```
// src/Acme/TaskBundle/Form/Type/TaskType.php

namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('task');
        $builder->add('dueDate', null, array('widget' => 'single_text'));
        $builder->add('issue', 'issue_selector');
    }

    public function getName()
    {
        return 'task';
    }
}
```

Now it will be very easy at any random place in your application to use this selector type to select an issue by number. No logic has to be added to your Controller at all.

If you want a new issue to be created when an unknown number is entered, you can instantiate it rather than throwing the TransformationFailedException, and even persist it to your entity manager if the task has no cascading options for the issue.

3.1.20 How to Dynamically Generate Forms Using Form Events

Before jumping right into dynamic form generation, let's have a quick review of what a bare form class looks like:

```
//src/Acme/DemoBundle/Form/ProductType.php
namespace Acme\DemoBundle\Form

use Symfony\Component\Form\AbstractType
use Symfony\Component\Form\FormBuilder;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
```

```
    {
        $builder->add('name');
        $builder->add('price');
    }

    public function getName()
    {
        return 'product';
    }
}
```

Note: If this particular section of code isn't already familiar to you, you probably need to take a step back and first review the Forms chapter before proceeding.

Let's assume for a moment that this form utilizes an imaginary "Product" class that has only two relevant properties ("name" and "price"). The form generated from this class will look the exact same regardless of a new Product is being created or if an existing product is being edited (e.g. a product fetched from the database).

Suppose now, that you don't want the user to be able to change the *name* value once the object has been created. To do this, you can rely on Symfony's *Event Dispatcher* system to analyze the data on the object and modify the form based on the Product object's data. In this entry, you'll learn how to add this level of flexibility to your forms.

Adding An Event Subscriber To A Form Class

So, instead of directly adding that "name" widget via our ProductType form class, let's delegate the responsibility of creating that particular field to an Event Subscriber:

```
//src/Acme/DemoBundle/Form/ProductType.php
namespace Acme\DemoBundle\Form

use Symfony\Component\Form\AbstractType
use Symfony\Component\Form\FormBuilder;
use Acme\DemoBundle\Form\EventListener\AddNameFieldSubscriber;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $subscriber = new AddNameFieldSubscriber($builder->getFormFactory());
        $builder->addEventSubscriber($subscriber);
        $builder->add('price');
    }

    public function getName()
    {
        return 'product';
    }
}
```

The event subscriber is passed the FormFactory object in its constructor so that our new subscriber is capable of creating the form widget once it is notified of the dispatched event during form creation.

Inside the Event Subscriber Class

The goal is to create a "name" field *only* if the underlying Product object is new (e.g. hasn't been persisted to the database). Based on that, the subscriber might look like the following:

```
// src/Acme/DemoBundle/Form/EventListener/AddNameFieldSubscriber.php
namespace Acme\DemoBundle\Form\EventListener;

use Symfony\Component\Form\Event\DataEvent;
use Symfony\Component\Form\FormFactoryInterface;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Form\FormEvents;

class AddNameFieldSubscriber implements EventSubscriberInterface
{
    private $factory;

    public function __construct(FormFactoryInterface $factory)
    {
        $this->factory = $factory;
    }

    public static function getSubscribedEvents()
    {
        // Tells the dispatcher that we want to listen on the form.pre_set_data
        // event and that the preSetData method should be called.
        return array(FormEvents::PRE_SET_DATA => 'preSetData');
    }

    public function preSetData(DataEvent $event)
    {
        $data = $event->getData();
        $form = $event->getForm();

        // During form creation setData() is called with null as an argument
        // by the FormBuilder constructor. We're only concerned with when
        // setData is called with an actual Entity object in it (whether new,
        // or fetched with Doctrine). This if statement let's us skip right
        // over the null condition.
        if (null === $data) {
            return;
        }

        // check if the product object is "new"
        if (!$data->getId()) {
            $form->add($this->factory->createNamed('text', 'name'));
        }
    }
}
```

Caution: It is easy to misunderstand the purpose of the `if (null === $data)` segment of this event subscriber. To fully understand its role, you might consider also taking a look at the Form class and paying special attention to where setData() is called at the end of the constructor, as well as the setData() method itself.

The `FormEvents::PRE_SET_DATA` line actually resolves to the string `form.pre_set_data`. The FormEvents class serves an organizational purpose. It is a centralized location in which you can find all of the various form events available.

While this example could have used the `form.set_data` event or even the `form.post_set_data` events just as effectively, by using `form.pre_set_data` we guarantee that the data being retrieved from the `Event` object has in no way been modified by any other subscribers or listeners. This is because `form.pre_set_data` passes a DataEvent object instead of the FilterDataEvent object passed by the `form.set_data` event. DataEvent, unlike its child FilterDataEvent, lacks a setData() method.

Note: You may view the full list of form events via the FormEvents class, found in the form bundle.

3.1.21 How to Embed a Collection of Forms

In this entry, you'll learn how to create a form that embeds a collection of many other forms. This could be useful, for example, if you had a `Task` class and you wanted to edit/create/remove many `Tag` objects related to that Task, right inside the same form.

Note: In this entry, we'll loosely assume that you're using Doctrine as your database store. But if you're not using Doctrine (e.g. Propel or just a database connection), it's all pretty much the same.

If you *are* using Doctrine, you'll need to add the Doctrine metadata, including the `ManyToMany` on the Task's `tags` property.

Let's start there: suppose that each `Task` belongs to multiple `Tags` objects. Start by creating a simple `Task` class:

```
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

class Task
{
    protected $description;

    protected $tags;

    public function __construct()
    {
        $this->tags = new ArrayCollection();
    }

    public function getDescription()
    {
        return $this->description;
    }

    public function setDescription($description)
    {
        $this->description = $description;
    }

    public function getTags()
    {
        return $this->tags;
    }

    public function setTags(ArrayCollection $tags)
    {
        $this->tags = $tags;
```

```
    }
}
```

Note: The `ArrayCollection` is specific to Doctrine and is basically the same as using an `array` (but it must be an `ArrayCollection`) if you're using Doctrine.

Now, create a `Tag` class. As you saw above, a `Task` can have many `Tag` objects:

```
// src/Acme/TaskBundle/Entity/Tag.php
namespace Acme\TaskBundle\Entity;

class Tag
{
    public $name;
}
```

Tip: The `name` property is public here, but it can just as easily be protected or private (but then it would need `getName` and `setName` methods).

Now let's get to the forms. Create a form class so that a `Tag` object can be modified by the user:

```
// src/Acme/TaskBundle/Form/Type/TagType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TagType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('name');
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class' => 'Acme\TaskBundle\Entity\Tag',
        );
    }

    public function getName()
    {
        return 'tag';
    }
}
```

With this, we have enough to render a tag form by itself. But since the end goal is to allow the tags of a `Task` to be modified right inside the task form itself, create a form for the `Task` class.

Notice that we embed a collection of `TagType` forms using the collection field type:

```
// src/Acme/TaskBundle/Form/Type/TaskType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;
```

```
class TaskType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('description');

        $builder->add('tags', 'collection', array('type' => new TagType()));
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class' => 'Acme\TaskBundle\Entity\Task',
        );
    }

    public function getName()
    {
        return 'task';
    }
}
```

In your controller, you'll now initialize a new instance of `TaskType`:

```
// src/Acme/TaskBundle/Controller/TaskController.php
namespace Acme\TaskBundle\Controller;

use Acme\TaskBundle\Entity\Task;
use Acme\TaskBundle\Entity\Tag;
use Acme\TaskBundle\Form\TaskType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class TaskController extends Controller
{
    public function newAction(Request $request)
    {
        $task = new Task();

        // dummy code - this is here just so that the Task has some tags
        // otherwise, this isn't an interesting example
        $tag1 = new Tag()
        $tag1->name = 'tag1';
        $task->getTags()->add($tag1);
        $tag2 = new Tag()
        $tag2->name = 'tag2';
        $task->getTags()->add($tag2);
        // end dummy code

        $form = $this->createForm(new TaskType(), $task);

        // maybe do some form process here in a POST request

        return $this->render('AcmeTaskBundle:Task:new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

The corresponding template is now able to render both the `description` field for the task form as well as all the `TagType` forms for any tags that are already related to this `Task`. In the above controller, I added some dummy code so that you can see this in action (since a `Task` has zero tags when first created).

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Task/new.html.twig #}
{# ... #}

{# render the task's only field: description #}
{{ form_row(form.description) }}

<h3>Tags</h3>
<ul class="tags">
    {# iterate over each existing tag and render its only field: name #}
                {% for tag in form.tags %}
        <li>{{ form_row(tag.name) }}</li>
                {% endfor %}
</ul>

{{ form_rest(form) }}
{# ... #}
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Task/new.html.php -->
<!-- ... -->

<h3>Tags</h3>
<ul class="tags">
                <?php foreach($form['tags'] as $tag): ?>
        <li><?php echo $view['form']->row($tag['name']) ?></li>
                <?php endforeach; ?>
</ul>

<?php echo $view['form']->rest($form) ?>
<!-- ... -->
```

When the user submits the form, the submitted data for the `Tags` fields are used to construct an ArrayCollection of `Tag` objects, which is then set on the `tag` field of the `Task` instance.

The `Tags` collection is accessible naturally via `$task->getTags()` and can be persisted to the database or used however you need.

So far, this works great, but this doesn't allow you to dynamically add new todos or delete existing todos. So, while editing existing todos will work great, your user can't actually add any new todos yet.

Allowing "new" todos with the "prototype"

Allowing the user to dynamically add new todos means that we'll need to use some JavaScript. Previously we added two tags to our form in the controller. Now we need to let the user add as many tag forms as he needs directly in the browser. This will be done through a bit of JavaScript.

The first thing we need to do is to tell the form collection know that it will receive an unknown number of tags. So far we've added two tags and the form type expects to receive exactly two, otherwise an error will be thrown: `This form should not contain extra fields`. To make this flexible, we add the `allow_add` option to our collection field:

```
// ...

public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('description');

    $builder->add('tags', 'collection', array(
        'type' => new TagType(),
        'allow_add' => true,
        'by_reference' => false,
    ));
}
```

Note that we also added `'by_reference' => false`. This is because we are not sending a reference to an existing tag but rather creating a new tag at the time we save the todo and its tags together.

The `allow_add` option also does one more thing. It will add a `data-prototype` property to the `div` containing the tag collection. This property contains html to add a Tag form element to our page like this:

```
<div data-prototype="&lt;div&gt;&lt;label class=&quot; required&quot;&gt;$$name$$&lt;/label&gt;&lt;di
</div>
```

We will get this property from our javascript and use it to display new Tag forms. To make things simple, we will embed jQuery in our page as it allows for easy cross-browser manipulation of the page.

First let's add a link on the `new` form with a class `add_tag_link`. Each time this is clicked by the user, we will add an empty tag for him:

```
$('.record_action').append('<li><a href="#" class="add_tag_link">Add a tag</a></li>');
```

We also include a template containing the javascript needed to add the form elements when the link is clicked.

Our script can be as simple as this:

```
function addTagForm() {
    // Get the div that holds the collection of tags
    var collectionHolder = $('#task_tags');
    // Get the data-prototype we explained earlier
    var prototype = collectionHolder.attr('data-prototype');
    // Replace '$$name$$' in the prototype's HTML to
    // instead be a number based on the current collection's length.
    form = prototype.replace(/\$\$name\$\$/g, collectionHolder.children().length);
    // Display the form in the page
    collectionHolder.append(form);
}
// Add the link to add tags
$('.record_action').append('<li><a href="#" class="add_tag_link">Add a tag</a></li>');
// When the link is clicked we add the field to input another tag
$('a.jslink').click(function(event){
    addTagForm();
});
```

Now, each time a user clicks the `Add a tag` link, a new sub form will appear on the page. The server side form component is aware it should not expect any specific size for the `Tag` collection. And all the tags we add while creating the new `Todo` will be saved together with it.

For more details, see the collection form type reference.

Allowing todos to be removed

This section has not been written yet, but will soon. If you're interested in writing this entry, see Contributing to the Documentation.

3.1.22 How to Create a Custom Form Field Type

Symfony comes with a bunch of core field types available for building forms. However there are situations where we want to create a custom form field type for a specific purpose. This recipe assumes we need a field definition that holds a person's gender, based on the existing choice field. This section explains how the field is defined, how we can customize its layout and finally, how we can register it for use in our application.

Defining the Field Type

In order to create the custom field type, first we have to create the class representing the field. In our situation the class holding the field type will be called *GenderType* and the file will be stored in the default location for form fields, which is `<BundleName>\Form\Type`. Make sure the field extends `Symfony\Component\Form\AbstractType`:

```
# src/Acme/DemoBundle/Form/Type/GenderType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class GenderType extends AbstractType
{
    public function getDefaultOptions(array $options)
    {
        return array(
            'choices' => array(
                'm' => 'Male',
                'f' => 'Female',
            )
        );
    }

    public function getParent(array $options)
    {
        return 'choice';
    }

    public function getName()
    {
        return 'gender';
    }
}
```

Tip: The location of this file is not important - the `Form\Type` directory is just a convention.

Here, the return value of the `getParent` function indicates that we're extending the `choice` field type. This means that, by default, we inherit all of the logic and rendering of that field type. To see some of the logic, check out the ChoiceType class. There are three methods that are particularly important:

- `buildForm()` - Each field type has a `buildForm` method, which is where you configure and build any field(s). Notice that this is the same method you use to setup *your* forms, and it works the same here.

- `buildView()` - This method is used to set any extra variables you'll need when rendering your field in a template. For example, in ChoiceType, a `multiple` variable is set and used in the template to set (or not set) the `multiple` attribute on the `select` field. See *Creating a Template for the Field* for more details.
- `getDefaultOptions()` - This defines options for your form type that can be used in `buildForm()` and `buildView()`. There are a lot of options common to all fields (see FieldType), but you can create any others that you need here.

Tip: If you're creating a field that consists of many fields, then be sure to set your "parent" type as `form` or something that extends `form`. Also, if you need to modify the "view" of any of your child types from your parent type, use the `buildViewBottomUp()` method.

The `getName()` method returns an identifier which should be unique in your application. This is used in various places, such as when customizing how your form type will be rendered.

The goal of our field was to extend the choice type to enable selection of a gender. This is achieved by fixing the `choices` to a list of possible genders.

Creating a Template for the Field

Each field type is rendered by a template fragment, which is determined in part by the value of your `getName()` method. For more information, see *What are Form Themes?*.

In this case, since our parent field is `choice`, we don't *need* to do any work as our custom field type will automatically be rendered like a `choice` type. But for the sake of this example, let's suppose that when our field is "expanded" (i.e. radio buttons or checkboxes, instead of a select field), we want to always render it in a `ul` element. In your form theme template (see above link for details), create a `gender_widget` block to handle this:

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}

{% block gender_widget %}
{% spaceless %}
    {% if expanded %}
        <ul {{ block('widget_container_attributes') }}>
        {% for child in form %}
            <li>
                {{ form_widget(child) }}
                {{ form_label(child) }}
            </li>
        {% endfor %}
        </ul>
    {% else %}
        {# just let the choice widget render the select tag #}
        {{ block('choice_widget') }}
    {% endif %}
{% endspaceless %}
{% endblock %}
```

Note: Make sure the correct widget prefix is used. In this example the name should be `gender_widget`, according to the value returned by `getName`. Further, the main config file should point to the custom form template so that it's used when rendering all forms.

```
# app/config/config.yml

twig:
    form:
```

```
        resources:
            - 'AcmeDemoBundle:Form:fields.html.twig'
```

Using the Field Type

You can now use your custom field type immediately, simply by creating a new instance of the type in one of your forms:

```
// src/Acme/DemoBundle/Form/Type/AuthorType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class AuthorType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('gender_code', new GenderType(), array(
            'empty_value' => 'Choose a gender',
        ));
    }
}
```

But this only works because the `GenderType()` is very simple. What if the gender codes were stored in configuration or in a database? The next section explains how more complex field types solve this problem.

Creating your Field Type as a Service

So far, this entry has assumed that you have a very simple custom field type. But if you need access to configuration, a database connection, or some other service, then you'll want to register your custom type as a service. For example, suppose that we're storing the gender parameters in configuration:

- *YAML*

```
    # app/config/config.yml
    parameters:
        genders:
            m: Male
            f: Female
```

- *XML*

```
    <!-- app/config/config.xml -->
    <parameters>
        <parameter key="genders" type="collection">
            <parameter key="m">Male</parameter>
            <parameter key="f">Female</parameter>
        </parameter>
    </parameters>
```

To use the parameter, we'll define our custom field type as a service, injecting the `genders` parameter value as the first argument to its to-be-created `__construct` function:

- *YAML*

```
    # src/Acme/DemoBundle/Resources/config/services.yml
    services:
        form.type.gender:
            class: Acme\DemoBundle\Form\Type\GenderType
            arguments:
                - "%genders%"
            tags:
                - { name: form.type, alias: gender }
```

- *XML*

```
    <!-- src/Acme/DemoBundle/Resources/config/services.xml -->
    <service id="form.type.gender" class="Acme\DemoBundle\Form\Type\GenderType">
        <argument>%genders%</argument>
        <tag name="form.type" alias="gender" />
    </service>
```

Tip: Make sure the services file is being imported. See *Importing Configuration with imports* for details.

Be sure that the `alias` attribute of the tag corresponds with the value returned by the `getName` method defined earlier. We'll see the importance of this in a moment when we use the custom field type. But first, add a `__construct` argument to `GenderType`, which receives the gender configuration:

```
# src/Acme/DemoBundle/Form/Type/GenderType.php
namespace Acme\DemoBundle\Form\Type;
// ...

class GenderType extends AbstractType
{
    private $genderChoices;

    public function __construct(array $genderChoices)
    {
        $this->genderChoices = $genderChoices;
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'choices' => $this->genderChoices,
        );
    }

    // ...
}
```

Great! The `GenderType` is now fueled by the configuration parameters and registered as a service. And because we used the `form.type` alias in its configuration, using the field is now much easier:

```
// src/Acme/DemoBundle/Form/Type/AuthorType.php
namespace Acme\DemoBundle\Form\Type;
// ...

class AuthorType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('gender_code', 'gender', array(
```

```
            'empty_value' => 'Choose a gender',
        ));
    }
}
```

Notice that instead of instantiating a new instance, we can just refer to it by the alias used in our service configuration, `gender`. Have fun!

3.1.23 How to create a Custom Validation Constraint

You can create a custom constraint by extending the base constraint class, `Symfony\Component\Validator\Constraint`. Options for your constraint are represented as public properties on the constraint class. For example, the Url constraint includes the `message` and `protocols` properties:

```
namespace Symfony\Component\Validator\Constraints;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class Url extends Constraint
{
    public $message = 'This value is not a valid URL';
    public $protocols = array('http', 'https', 'ftp', 'ftps');
}
```

Note: The `@Annotation` annotation is necessary for this new constraint in order to make it available for use in classes via annotations.

As you can see, a constraint class is fairly minimal. The actual validation is performed by a another "constraint validator" class. The constraint validator class is specified by the constraint's `validatedBy()` method, which includes some simple default logic:

```
// in the base Symfony\Component\Validator\Constraint class
public function validatedBy()
{
    return get_class($this).'Validator';
}
```

In other words, if you create a custom `Constraint` (e.g. `MyConstraint`), Symfony2 will automatically look for another class, `MyConstraintValidator` when actually performing the validation.

The validator class is also simple, and only has one required method: `isValid`. Take the `NotBlankValidator` as an example:

```
class NotBlankValidator extends ConstraintValidator
{
    public function isValid($value, Constraint $constraint)
    {
        if (null === $value || '' === $value) {
            $this->setMessage($constraint->message);

            return false;
        }
```

```
        return true;
    }
}
```

Constraint Validators with Dependencies

If your constraint validator has dependencies, such as a database connection, it will need to be configured as a service in the dependency injection container. This service must include the `validator.constraint_validator` tag and an `alias` attribute:

- *YAML*

```
    services:
        validator.unique.your_validator_name:
            class: Fully\Qualified\Validator\Class\Name
            tags:
                - { name: validator.constraint_validator, alias: alias_name }
```

- *XML*

```
    <service id="validator.unique.your_validator_name" class="Fully\Qualified\Validator\Class\Name">
        <argument type="service" id="doctrine.orm.default_entity_manager" />
        <tag name="validator.constraint_validator" alias="alias_name" />
    </service>
```

- *PHP*

```
    $container
        ->register('validator.unique.your_validator_name', 'Fully\Qualified\Validator\Class\Name')
        ->addTag('validator.constraint_validator', array('alias' => 'alias_name'))
    ;
```

Your constraint class should now use this alias to reference the appropriate validator:

```
public function validatedBy()
{
    return 'alias_name';
}
```

As mentioned above, Symfony2 will automatically look for a class named after the constraint, with `Validator` appended. If your constraint validator is defined as a service, it's important that you override the `validatedBy()` method to return the alias used when defining your service, otherwise Symfony2 won't use the constraint validator service, and will instantiate the class instead, without any dependencies injected.

3.1.24 How to Master and Create new Environments

Every application is the combination of code and a set of configuration that dictates how that code should function. The configuration may define the database being used, whether or not something should be cached, or how verbose logging should be. In Symfony2, the idea of "environments" is the idea that the same codebase can be run using multiple different configurations. For example, the `dev` environment should use configuration that makes development easy and friendly, while the `prod` environment should use a set of configuration optimized for speed.

Different Environments, Different Configuration Files

A typical Symfony2 application begins with three environments: `dev`, `prod`, and `test`. As discussed, each "environment" simply represents a way to execute the same codebase with different configuration. It should be no surprise

then that each environment loads its own individual configuration file. If you're using the YAML configuration format, the following files are used:

- for the `dev` environment: `app/config/config_dev.yml`
- for the `prod` environment: `app/config/config_prod.yml`
- for the `test` environment: `app/config/config_test.yml`

This works via a simple standard that's used by default inside the `AppKernel` class:

```
// app/AppKernel.php
// ...

class AppKernel extends Kernel
{
    // ...

    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
    }
}
```

As you can see, when Symfony2 is loaded, it uses the given environment to determine which configuration file to load. This accomplishes the goal of multiple environments in an elegant, powerful and transparent way.

Of course, in reality, each environment differs only somewhat from others. Generally, all environments will share a large base of common configuration. Opening the "dev" configuration file, you can see how this is accomplished easily and transparently:

- *YAML*

```
imports:
    - { resource: config.yml }

# ...
```

- *XML*

```
<imports>
    <import resource="config.xml" />
</imports>

<!-- ... -->
```

- *PHP*

```
$loader->import('config.php');

// ...
```

To share common configuration, each environment's configuration file simply first imports from a central configuration file (`config.yml`). The remainder of the file can then deviate from the default configuration by overriding individual parameters. For example, by default, the `web_profiler` toolbar is disabled. However, in the `dev` environment, the toolbar is activated by modifying the default value in the `dev` configuration file:

- *YAML*

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }
```

```
    web_profiler:
        toolbar: true
        # ...
```

- *XML*

```
    <!-- app/config/config_dev.xml -->
    <imports>
        <import resource="config.xml" />
    </imports>

    <webprofiler:config
        toolbar="true"
        # ...
    />
```

- *PHP*

```
    // app/config/config_dev.php
    $loader->import('config.php');

    $container->loadFromExtension('web_profiler', array(
        'toolbar' => true,
        // ..
    ));
```

Executing an Application in Different Environments

To execute the application in each environment, load up the application using either the `app.php` (for the `prod` environment) or the `app_dev.php` (for the `dev` environment) front controller:

```
http://localhost/app.php      -> *prod* environment
http://localhost/app_dev.php  -> *dev* environment
```

Note: The given URLs assume that your web server is configured to use the `web/` directory of the application as its root. Read more in Installing Symfony2.

If you open up one of these files, you'll quickly see that the environment used by each is explicitly set:

```
1  <?php
2
3  require_once __DIR__.'/../app/bootstrap_cache.php';
4  require_once __DIR__.'/../app/AppCache.php';
5
6  use Symfony\Component\HttpFoundation\Request;
7
8  $kernel = new AppCache(new AppKernel('prod', false));
9  $kernel->handle(Request::createFromGlobals())->send();
```

As you can see, the `prod` key specifies that this environment will run in the `prod` environment. A Symfony2 application can be executed in any environment by using this code and changing the environment string.

Note: The `test` environment is used when writing functional tests and is not accessible in the browser directly via a front controller. In other words, unlike the other environments, there is no `app_test.php` front controller file.

***Debug* Mode**

Important, but unrelated to the topic of *environments* is the `false` key on line 8 of the front controller above. This specifies whether or not the application should run in "debug mode". Regardless of the environment, a Symfony2 application can be run with debug mode set to `true` or `false`. This affects many things in the application, such as whether or not errors should be displayed or if cache files are dynamically rebuilt on each request. Though not a requirement, debug mode is generally set to `true` for the `dev` and `test` environments and `false` for the `prod` environment.
Internally, the value of the debug mode becomes the `kernel.debug` parameter used inside the service container. If you look inside the application configuration file, you'll see the parameter used, for example, to turn logging on or off when using the Doctrine DBAL:

- *YAML*

```
doctrine:
    dbal:
        logging:  %kernel.debug%
        # ...
```

- *XML*

```
<doctrine:dbal logging="%kernel.debug%" ... />
```

- *PHP*

```
$container->loadFromExtension('doctrine', array(
    'dbal' => array(
        'logging'  => '%kernel.debug%',
        // ...
    ),
    // ...
));
```

Creating a New Environment

By default, a Symfony2 application has three environments that handle most cases. Of course, since an environment is nothing more than a string that corresponds to a set of configuration, creating a new environment is quite easy.

Suppose, for example, that before deployment, you need to benchmark your application. One way to benchmark the application is to use near-production settings, but with Symfony2's `web_profiler` enabled. This allows Symfony2 to record information about your application while benchmarking.

The best way to accomplish this is via a new environment called, for example, `benchmark`. Start by creating a new configuration file:

- *YAML*

```
# app/config/config_benchmark.yml

imports:
    - { resource: config_prod.yml }

framework:
    profiler: { only_exceptions: false }
```

- *XML*

```
<!-- app/config/config_benchmark.xml -->
```

```
    <imports>
        <import resource="config_prod.xml" />
    </imports>

    <framework:config>
        <framework:profiler only-exceptions="false" />
    </framework:config>
```

- *PHP*

```
    // app/config/config_benchmark.php

    $loader->import('config_prod.php')

    $container->loadFromExtension('framework', array(
        'profiler' => array('only-exceptions' => false),
    ));
```

And with this simple addition, the application now supports a new environment called `benchmark`.

This new configuration file imports the configuration from the `prod` environment and modifies it. This guarantees that the new environment is identical to the `prod` environment, except for any changes explicitly made here.

Because you'll want this environment to be accessible via a browser, you should also create a front controller for it. Copy the `web/app.php` file to `web/app_benchmark.php` and edit the environment to be `benchmark`:

```
<?php

require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('benchmark', false);
$kernel->handle(Request::createFromGlobals())->send();
```

The new environment is now accessible via:

```
http://localhost/app_benchmark.php
```

Note: Some environments, like the `dev` environment, are never meant to be accessed on any deployed server by the general public. This is because certain environments, for debugging purposes, may give too much information about the application or underlying infrastructure. To be sure these environments aren't accessible, the front controller is usually protected from external IP addresses via the following code at the top of the controller:

```
    if (!in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', '::1'))) {
        die('You are not allowed to access this file. Check '.basename(__FILE__).' for more informat
    }
```

Environments and the Cache Directory

Symfony2 takes advantage of caching in many ways: the application configuration, routing configuration, Twig templates and more are cached to PHP objects stored in files on the filesystem.

By default, these cached files are largely stored in the `app/cache` directory. However, each environment caches its own set of files:

```
app/cache/dev   - cache directory for the *dev* environment
app/cache/prod  - cache directory for the *prod* environment
```

Sometimes, when debugging, it may be helpful to inspect a cached file to understand how something is working. When doing so, remember to look in the directory of the environment you're using (most commonly `dev` while developing and debugging). While it can vary, the `app/cache/dev` directory includes the following:

- `appDevDebugProjectContainer.php` - the cached "service container" that represents the cached application configuration;
- `appdevUrlGenerator.php` - the PHP class generated from the routing configuration and used when generating URLs;
- `appdevUrlMatcher.php` - the PHP class used for route matching - look here to see the compiled regular expression logic used to match incoming URLs to different routes;
- `twig/` - this directory contains all the cached Twig templates.

Going Further

Read the article on How to Set External Parameters in the Service Container.

3.1.25 How to Set External Parameters in the Service Container

In the chapter How to Master and Create new Environments, you learned how to manage your application configuration. At times, it may benefit your application to store certain credentials outside of your project code. Database configuration is one such example. The flexibility of the symfony service container allows you to easily do this.

Environment Variables

Symfony will grab any environment variable prefixed with `SYMFONY__` and set it as a parameter in the service container. Double underscores are replaced with a period, as a period is not a valid character in an environment variable name.

For example, if you're using Apache, environment variables can be set using the following `VirtualHost` configuration:

```
<VirtualHost *:80>
    ServerName      Symfony2
    DocumentRoot    "/path/to/symfony_2_app/web"
    DirectoryIndex  index.php index.html
    SetEnv          SYMFONY__DATABASE__USER user
    SetEnv          SYMFONY__DATABASE__PASSWORD secret

    <Directory "/path/to/symfony_2_app/web">
        AllowOverride All
        Allow from All
    </Directory>
</VirtualHost>
```

Note: The example above is for an Apache configuration, using the SetEnv directive. However, this will work for any web server which supports the setting of environment variables.

Also, in order for your console to work (which does not use Apache), you must export these as shell variables. On a Unix system, you can run the following:

```
export SYMFONY__DATABASE__USER=user
export SYMFONY__DATABASE__PASSWORD=secret
```

Now that you have declared an environment variable, it will be present in the PHP `$_SERVER` global variable. Symfony then automatically sets all `$_SERVER` variables prefixed with `SYMFONY__` as parameters in the service container.

You can now reference these parameters wherever you need them.

- *YAML*

```
doctrine:
    dbal:
        driver    pdo_mysql
        dbname:   symfony2_project
        user:     %database.user%
        password: %database.password%
```

- *XML*

```
<!-- xmlns:doctrine="http://symfony.com/schema/dic/doctrine" -->
<!-- xsi:schemaLocation="http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/do

<doctrine:config>
    <doctrine:dbal
        driver="pdo_mysql"
        dbname="symfony2_project"
        user="%database.user%"
        password="%database.password%"
    />
</doctrine:config>
```

- *PHP*

```
$container->loadFromExtension('doctrine', array('dbal' => array(
    'driver'   => 'pdo_mysql',
    'dbname'   => 'symfony2_project',
    'user'     => '%database.user%',
    'password' => '%database.password%',
));
```

Constants

The container also has support for setting PHP constants as parameters. To take advantage of this feature, map the name of your constant to a parameter key, and define the type as `constant`.

```
<?xml version="1.0" encoding="UTF-8"?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>

    <parameters>
        <parameter key="global.constant.value" type="constant">GLOBAL_CONSTANT</parameter>
        <parameter key="my_class.constant.value" type="constant">My_Class::CONSTANT_NAME</parame
    </parameters>
</container>
```

Note: This only works for XML configuration. If you're *not* using XML, simply import an XML file to take advantage of this functionality:

```
// app/config/config.yml
imports:
    - { resource: parameters.xml }
```

Miscellaneous Configuration

The `imports` directive can be used to pull in parameters stored elsewhere. Importing a PHP file gives you the flexibility to add whatever is needed in the container. The following imports a file named `parameters.php`.

- *YAML*

```
# app/config/config.yml
imports:
    - { resource: parameters.php }
```

- *XML*

```
<!-- app/config/config.xml -->
<imports>
    <import resource="parameters.php" />
</imports>
```

- *PHP*

```
// app/config/config.php
$loader->import('parameters.php');
```

Note: A resource file can be one of many types. PHP, XML, YAML, INI, and closure resources are all supported by the `imports` directive.

In `parameters.php`, tell the service container the parameters that you wish to set. This is useful when important configuration is in a nonstandard format. The example below includes a Drupal database's configuration in the symfony service container.

```
// app/config/parameters.php

include_once('/path/to/drupal/sites/default/settings.php');
$container->setParameter('drupal.database.url', $db_url);
```

3.1.26 How to Use a Factory to Create Services

Symfony2's Service Container provides a powerful way of controlling the creation of objects, allowing you to specify arguments passed to the constructor as well as calling methods and setting parameters. Sometimes, however, this will not provide you with everything you need to construct your objects. For this situation, you can use a factory to create the object and tell the service container to call a method on the factory rather than directly instantiating the object.

Suppose you have a factory that configures and returns a new NewsletterManager object:

```
namespace Acme\HelloBundle\Newsletter;

class NewsletterFactory
{
```

```
    public function get()
    {
        $newsletterManager = new NewsletterManager();

        // ...

        return $newsletterManager;
    }
}
```

To make the `NewsletterManager` object available as a service, you can configure the service container to use the `NewsletterFactory` factory class:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager
    newsletter_factory.class: Acme\HelloBundle\Newsletter\NewsletterFactory
services:
    newsletter_manager:
        class:          %newsletter_manager.class%
        factory_class:  %newsletter_factory.class%
        factory_method: get
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager</par
    <parameter key="newsletter_factory.class">Acme\HelloBundle\Newsletter\NewsletterFactory</par
</parameters>

<services>
    <service id="newsletter_manager"
             class="%newsletter_manager.class%"
             factory-class="%newsletter_factory.class%"
             factory-method="get"
    />
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana
$container->setParameter('newsletter_factory.class', 'Acme\HelloBundle\Newsletter\NewsletterFact

$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%'
))->setFactoryClass(
    '%newsletter_factory.class%'
)->setFactoryMethod(
    'get'
);
```

When you specify the class to use for the factory (via `factory_class`) the method will be called statically. If the factory itself should be instantiated and the resulting object's method called (as in this example), configure the factory itself as a service:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager
    newsletter_factory.class: Acme\HelloBundle\Newsletter\NewsletterFactory
services:
    newsletter_factory:
        class:            %newsletter_factory.class%
    newsletter_manager:
        class:            %newsletter_manager.class%
        factory_service:  newsletter_factory
        factory_method:   get
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager</par
    <parameter key="newsletter_factory.class">Acme\HelloBundle\Newsletter\NewsletterFactory</par
</parameters>

<services>
    <service id="newsletter_factory" class="%newsletter_factory.class%"/>
    <service id="newsletter_manager"
             class="%newsletter_manager.class%"
             factory-service="newsletter_factory"
             factory-method="get"
    />
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana
$container->setParameter('newsletter_factory.class', 'Acme\HelloBundle\Newsletter\NewsletterFact

$container->setDefinition('newsletter_factory', new Definition(
    '%newsletter_factory.class%'
))
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%'
))->setFactoryService(
    'newsletter_factory'
)->setFactoryMethod(
    'get'
);
```

Note: The factory service is specified by its id name and not a reference to the service itself. So, you do not need to use the @ syntax.

Passing Arguments to the Factory Method

If you need to pass arguments to the factory method, you can use the `arguments` options inside the service container. For example, suppose the `get` method in the previous example takes the `templating` service as an argument:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager
    newsletter_factory.class: Acme\HelloBundle\Newsletter\NewsletterFactory
services:
    newsletter_factory:
        class:            %newsletter_factory.class%
    newsletter_manager:
        class:            %newsletter_manager.class%
        factory_service:  newsletter_factory
        factory_method:   get
        arguments:
            -             @templating
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager</par
    <parameter key="newsletter_factory.class">Acme\HelloBundle\Newsletter\NewsletterFactory</par
</parameters>

<services>
    <service id="newsletter_factory" class="%newsletter_factory.class%"/>
    <service id="newsletter_manager"
             class="%newsletter_manager.class%"
             factory-service="newsletter_factory"
             factory-method="get"
    >
        <argument type="service" id="templating" />
    </service>
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana
$container->setParameter('newsletter_factory.class', 'Acme\HelloBundle\Newsletter\NewsletterFact

$container->setDefinition('newsletter_factory', new Definition(
    '%newsletter_factory.class%'
))
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%',
    array(new Reference('templating'))
))->setFactoryService(
    'newsletter_factory'
)->setFactoryMethod(
```

```
        'get'
    );
```

3.1.27 How to Manage Common Dependencies with Parent Services

As you add more functionality to your application, you may well start to have related classes that share some of the same dependencies. For example you may have a Newsletter Manager which uses setter injection to set its dependencies:

```
namespace Acme\HelloBundle\Mail;

use Acme\HelloBundle\Mailer;
use Acme\HelloBundle\EmailFormatter;

class NewsletterManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }
    // ...
}
```

and also a Greeting Card class which shares the same dependencies:

```
namespace Acme\HelloBundle\Mail;

use Acme\HelloBundle\Mailer;
use Acme\HelloBundle\EmailFormatter;

class GreetingCardManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }
    // ...
}
```

The service config for these classes would look something like this:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Mail\NewsletterManager
    greeting_card_manager.class: Acme\HelloBundle\Mail\GreetingCardManager
services:
    my_mailer:
        # ...
    my_email_formatter:
        # ...
    newsletter_manager:
        class:     %newsletter_manager.class%
        calls:
            - [ setMailer, [ @my_mailer ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]

    greeting_card_manager:
        class:     %greeting_card_manager.class%
        calls:
            - [ setMailer, [ @my_mailer ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Mail\NewsletterManager</parameter
    <parameter key="greeting_card_manager.class">Acme\HelloBundle\Mail\GreetingCardManager</para
</parameters>

<services>
    <service id="my_mailer" ... >
      <!-- ... -->
    </service>
    <service id="my_email_formatter" ... >
      <!-- ... -->
    </service>
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <call method="setMailer">
            <argument type="service" id="my_mailer" />
        </call>
        <call method="setEmailFormatter">
            <argument type="service" id="my_email_formatter" />
        </call>
    </service>
    <service id="greeting_card_manager" class="%greeting_card_manager.class%">
        <call method="setMailer">
            <argument type="service" id="my_mailer" />
        </call>
        <call method="setEmailFormatter">
            <argument type="service" id="my_email_formatter" />
        </call>
    </service>
</services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    // ...
    $container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Mail\NewsletterManager');
    $container->setParameter('greeting_card_manager.class', 'Acme\HelloBundle\Mail\GreetingCardManag

    $container->setDefinition('my_mailer', ... );
    $container->setDefinition('my_email_formatter', ... );
    $container->setDefinition('newsletter_manager', new Definition(
        '%newsletter_manager.class%'
    ))->addMethodCall('setMailer', array(
        new Reference('my_mailer')
    ))->addMethodCall('setEmailFormatter', array(
        new Reference('my_email_formatter')
    ));
    $container->setDefinition('greeting_card_manager', new Definition(
        '%greeting_card_manager.class%'
    ))->addMethodCall('setMailer', array(
        new Reference('my_mailer')
    ))->addMethodCall('setEmailFormatter', array(
        new Reference('my_email_formatter')
    ));
```

There is a lot of repetition in both the classes and the configuration. This means that if you changed, for example, the `Mailer` of `EmailFormatter` classes to be injected via the constructor, you would need to update the config in two places. Likewise if you needed to make changes to the setter methods you would need to do this in both classes. The typical way to deal with the common methods of these related classes would be to extract them to a super class:

```
namespace Acme\HelloBundle\Mail;

use Acme\HelloBundle\Mailer;
use Acme\HelloBundle\EmailFormatter;

abstract class MailManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }
    // ...
}
```

The `NewsletterManager` and `GreetingCardManager` can then extend this super class:

```
namespace Acme\HelloBundle\Mail;

class NewsletterManager extends MailManager
{
    // ...
```

```
}
```

and:

```
namespace Acme\HelloBundle\Mail;

class GreetingCardManager extends MailManager
{
    // ...
}
```

In a similar fashion, the Symfony2 service container also supports extending services in the configuration so you can also reduce the repetition by specifying a parent for a service.

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Mail\NewsletterManager
    greeting_card_manager.class: Acme\HelloBundle\Mail\GreetingCardManager
    mail_manager.class: Acme\HelloBundle\Mail\MailManager
services:
    my_mailer:
        # ...
    my_email_formatter:
        # ...
    mail_manager:
        class:     %mail_manager.class%
        abstract:  true
        calls:
            - [ setMailer, [ @my_mailer ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]

    newsletter_manager:
        class:     %newsletter_manager.class%
        parent: mail_manager

    greeting_card_manager:
        class:     %greeting_card_manager.class%
        parent: mail_manager
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Mail\NewsletterManager</parameter
    <parameter key="greeting_card_manager.class">Acme\HelloBundle\Mail\GreetingCardManager</para
    <parameter key="mail_manager.class">Acme\HelloBundle\Mail\MailManager</parameter>
</parameters>

<services>
    <service id="my_mailer" ... >
      <!-- ... -->
    </service>
    <service id="my_email_formatter" ... >
      <!-- ... -->
    </service>
```

```
        <service id="mail_manager" class="%mail_manager.class%" abstract="true">
            <call method="setMailer">
                <argument type="service" id="my_mailer" />
            </call>
            <call method="setEmailFormatter">
                <argument type="service" id="my_email_formatter" />
            </call>
        </service>
        <service id="newsletter_manager" class="%newsletter_manager.class%" parent="mail_manager"/>
        <service id="greeting_card_manager" class="%greeting_card_manager.class%" parent="mail_manag
    </services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    // ...
    $container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Mail\NewsletterManager');
    $container->setParameter('greeting_card_manager.class', 'Acme\HelloBundle\Mail\GreetingCardManag
    $container->setParameter('mail_manager.class', 'Acme\HelloBundle\Mail\MailManager');

    $container->setDefinition('my_mailer', ... );
    $container->setDefinition('my_email_formatter', ... );
    $container->setDefinition('mail_manager', new Definition(
        '%mail_manager.class%'
    ))->SetAbstract(
        true
    )->addMethodCall('setMailer', array(
        new Reference('my_mailer')
    ))->addMethodCall('setEmailFormatter', array(
        new Reference('my_email_formatter')
    ));
    $container->setDefinition('newsletter_manager', new DefinitionDecorator(
        'mail_manager'
    ))->setClass(
        '%newsletter_manager.class%'
    );
    $container->setDefinition('greeting_card_manager', new DefinitionDecorator(
        'mail_manager'
    ))->setClass(
        '%greeting_card_manager.class%'
    );
```

In this context, having a `parent` service implies that the arguments and method calls of the parent service should be used for the child services. Specifically, the setter methods defined for the parent service will be called when the child services are instantiated.

Note: If you remove the `parent` config key, the services will still be instantiated and they will still of course extend the `MailManager` class. The difference is that omitting the `parent` config key will mean that the `calls` defined on the `mail_manager` service will not be executed when the child services are instantiated.

The parent class is abstract as it should not be directly instantiated. Setting it to abstract in the config file as has been done above will mean that it can only be used as a parent service and cannot be used directly as a service to inject and will be removed at compile time. In other words, it exists merely as a "template" that other services can use.

Overriding Parent Dependencies

There may be times where you want to override what class is passed in for a dependency of one child service only. Fortunately, by adding the method call config for the child service, the dependencies set by the parent class will be overridden. So if you needed to pass a different dependency just to the `NewsletterManager` class, the config would look like this:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Mail\NewsletterManager
    greeting_card_manager.class: Acme\HelloBundle\Mail\GreetingCardManager
    mail_manager.class: Acme\HelloBundle\Mail\MailManager
services:
    my_mailer:
        # ...
    my_alternative_mailer:
        # ...
    my_email_formatter:
        # ...
    mail_manager:
        class:     %mail_manager.class%
        abstract:  true
        calls:
            - [ setMailer, [ @my_mailer ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]

    newsletter_manager:
        class:     %newsletter_manager.class%
        parent: mail_manager
        calls:
            - [ setMailer, [ @my_alternative_mailer ] ]

    greeting_card_manager:
        class:     %greeting_card_manager.class%
        parent: mail_manager
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Mail\NewsletterManager</parameter
    <parameter key="greeting_card_manager.class">Acme\HelloBundle\Mail\GreetingCardManager</para
    <parameter key="mail_manager.class">Acme\HelloBundle\Mail\MailManager</parameter>
</parameters>

<services>
    <service id="my_mailer" ... >
      <!-- ... -->
    </service>
    <service id="my_alternative_mailer" ... >
      <!-- ... -->
    </service>
    <service id="my_email_formatter" ... >
      <!-- ... -->
    </service>
```

```
        <service id="mail_manager" class="%mail_manager.class%" abstract="true">
            <call method="setMailer">
                <argument type="service" id="my_mailer" />
            </call>
            <call method="setEmailFormatter">
                <argument type="service" id="my_email_formatter" />
            </call>
        </service>
        <service id="newsletter_manager" class="%newsletter_manager.class%" parent="mail_manager">
             <call method="setMailer">
                <argument type="service" id="my_alternative_mailer" />
            </call>
        </service>
        <service id="greeting_card_manager" class="%greeting_card_manager.class%" parent="mail_manag
    </services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    // ...
    $container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Mail\NewsletterManager');
    $container->setParameter('greeting_card_manager.class', 'Acme\HelloBundle\Mail\GreetingCardManag
    $container->setParameter('mail_manager.class', 'Acme\HelloBundle\Mail\MailManager');

    $container->setDefinition('my_mailer', ... );
    $container->setDefinition('my_alternative_mailer', ... );
    $container->setDefinition('my_email_formatter', ... );
    $container->setDefinition('mail_manager', new Definition(
        '%mail_manager.class%'
    ))->SetAbstract(
        true
    )->addMethodCall('setMailer', array(
        new Reference('my_mailer')
    ))->addMethodCall('setEmailFormatter', array(
        new Reference('my_email_formatter')
    ));
    $container->setDefinition('newsletter_manager', new DefinitionDecorator(
        'mail_manager'
    ))->setClass(
        '%newsletter_manager.class%'
    )->addMethodCall('setMailer', array(
        new Reference('my_alternative_mailer')
    ));
    $container->setDefinition('greeting_card_manager', new DefinitionDecorator(
        'mail_manager'
    ))->setClass(
        '%greeting_card_manager.class%'
    );
```

The `GreetingCardManager` will receive the same dependencies as before, but the `NewsletterManager` will be passed the `my_alternative_mailer` instead of the `my_mailer` service.

Collections of Dependencies

It should be noted that the overridden setter method in the previous example is actually called twice - once per the parent definition and once per the child definition. In the previous example, that was fine, since the second `setMailer` call replaces mailer object set by the first call.

In some cases, however, this can be a problem. For example, if the overridden method call involves adding something to a collection, then two objects will be added to that collection. The following shows such a case, if the parent class looks like this:

```
namespace Acme\HelloBundle\Mail;

use Acme\HelloBundle\Mailer;
use Acme\HelloBundle\EmailFormatter;

abstract class MailManager
{
    protected $filters;

    public function setFilter($filter)
    {
        $this->filters[] = $filter;
    }
    // ...
}
```

If you had the following config:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Mail\NewsletterManager
    mail_manager.class: Acme\HelloBundle\Mail\MailManager
services:
    my_filter:
        # ...
    another_filter:
        # ...
    mail_manager:
        class:     %mail_manager.class%
        abstract:  true
        calls:
            - [ setFilter, [ @my_filter ] ]

    newsletter_manager:
        class:     %newsletter_manager.class%
        parent: mail_manager
        calls:
            - [ setFilter, [ @another_filter ] ]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Mail\NewsletterManager</parameter
    <parameter key="mail_manager.class">Acme\HelloBundle\Mail\MailManager</parameter>
</parameters>
```

```
<services>
    <service id="my_filter" ... >
      <!-- ... -->
    </service>
    <service id="another_filter" ... >
      <!-- ... -->
    </service>
    <service id="mail_manager" class="%mail_manager.class%" abstract="true">
        <call method="setFilter">
             <argument type="service" id="my_filter" />
        </call>
    </service>
    <service id="newsletter_manager" class="%newsletter_manager.class%" parent="mail_manager">
         <call method="setFilter">
             <argument type="service" id="another_filter" />
        </call>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Mail\NewsletterManager');
$container->setParameter('mail_manager.class', 'Acme\HelloBundle\Mail\MailManager');

$container->setDefinition('my_filter', ... );
$container->setDefinition('another_filter', ... );
$container->setDefinition('mail_manager', new Definition(
    '%mail_manager.class%'
))->SetAbstract(
    true
)->addMethodCall('setFilter', array(
    new Reference('my_filter')
));
$container->setDefinition('newsletter_manager', new DefinitionDecorator(
    'mail_manager'
))->setClass(
    '%newsletter_manager.class%'
)->addMethodCall('setFilter', array(
    new Reference('another_filter')
));
```

In this example, the `setFilter` of the `newsletter_manager` service will be called twice, resulting in the `$filters` array containing both `my_filter` and `another_filter` objects. This is great if you just want to add additional filters to the subclasses. If you want to replace the filters passed to the subclass, removing the parent setting from the config will prevent the base class from calling to `setFilter`.

3.1.28 How to work with Scopes

This entry is all about scopes, a somewhat advanced topic related to the Service Container. If you've ever gotten an error mentioning "scopes" when creating services, or need to create a service that depends on the *request* service, then this entry is for you.

Understanding Scopes

The scope of a service controls how long an instance of a service is used by the container. The Dependency Injection component provides two generic scopes:

- *container* (the default one): The same instance is used each time you request it from this container.
- *prototype*: A new instance is created each time you request the service.

The FrameworkBundle also defines a third scope: *request*. This scopes is tied to the request, meaning a new instance is created for each subrequest and is unavailable outside the request (for instance in the CLI).

Scopes add a constraint on the dependencies of a service: a service cannot depend on services from a narrower scope. For example, if you create a generic *my_foo* service, but try to inject the *request* component, you'll receive a `Symfony\Component\DependencyInjection\Exception\ScopeWideningInjectionException` when compiling the container. Read the sidebar below for more details.

Scopes and Dependencies

Imagine you've configured a *my_mailer* service. You haven't configured the scope of the service, so it defaults to *container*. In other words, everytime you ask the container for the *my_mailer* service, you get the same object back. This is usually how you want your services to work.
Imagine, however, that you need the *request* service in your *my_mailer* service, maybe because you're reading the URL of the current request. So, you add it as a constructor argument. Let's look at why this presents a problem:

- When requesting *my_mailer*, an instance of *my_mailer* (let's call it *MailerA*) is created and the *request* service (let's call it *RequestA*) is passed to it. Life is good!
- You've now made a subrequest in Symfony, which is a fancy way of saying that you've called, for example, the *{% render ... %}* Twig function, which executes another controller. Internally, the old *request* service (*RequestA*) is actually replaced by a new request instance (*RequestB*). This happens in the background, and it's totally normal.
- In your embedded controller, you once again ask for the *my_mailer* service. Since your service is in the *container* scope, the same instance (*MailerA*) is just re-used. But here's the problem: the *MailerA* instance still contains the old *RequestA* object, which is now **not** the correct request object to have (*RequestB* is now the current *request* service). This is subtle, but the mis-match could cause major problems, which is why it's not allowed.
  So, that's the reason *why* scopes exists, and how they can cause problems. Keep reading to find out the common solutions.

Note: A service can of course depend on a service from a wider scope without any issue.

Setting the Scope in the Definition

The scope of a service is defined in the definition of the service:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
services:
    greeting_card_manager:
        class: Acme\HelloBundle\Mail\GreetingCardManager
        scope: request
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/services.xml -->
    <services>
        <service id="greeting_card_manager" class="Acme\HelloBundle\Mail\GreetingCardManager" scope=
    </services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;

    $container->setDefinition(
        'greeting_card_manager',
        new Definition('Acme\HelloBundle\Mail\GreetingCardManager')
    )->setScope('request');
```

If you don't specify the scope, it defaults to *container*, which is what you want most of the time. Unless your service depends on another service that's scoped to a narrower scope (most commonly, the *request* service), you probably don't need to set the scope.

Using a Service from a narrower Scope

If your service depends on a scoped service, the best solution is to put it in the same scope (or a narrower one). Usually, this means putting your new service in the *request* scope.

But this is not always possible (for instance, a twig extension must be in the *container* scope as the Twig environment needs it as a dependency). In these cases, you should pass the entire container into your service and retrieve your dependency from the container each time we need it to be sure you have the right instance:

```
namespace Acme\HelloBundle\Mail;

use Symfony\Component\DependencyInjection\ContainerInterface;

class Mailer
{
    protected $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }

    public function sendEmail()
    {
        $request = $this->container->get('request');
        // Do something using the request here
    }
}
```

Caution: Take care not to store the request in a property of the object for a future call of the service as it would be the same issue described in the first section (except that symfony cannot detect that you are wrong).

The service config for this class would look something like this:

- *YAML*

```
    # src/Acme/HelloBundle/Resources/config/services.yml
    parameters:
```

```
        # ...
        my_mailer.class: Acme\HelloBundle\Mail\Mailer
    services:
        my_mailer:
            class:     %my_mailer.class%
            arguments:
                - "@service_container"
            # scope: container can be omitted as it is the default
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/services.xml -->
    <parameters>
        <!-- ... -->
        <parameter key="my_mailer.class">Acme\HelloBundle\Mail\Mailer</parameter>
    </parameters>

    <services>
        <service id="my_mailer" class="%my_mailer.class%">
             <argument type="service" id="service_container" />
        </service>
    </services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    // ...
    $container->setParameter('my_mailer.class', 'Acme\HelloBundle\Mail\Mailer');

    $container->setDefinition('my_mailer', new Definition(
        '%my_mailer.class%',
        array(new Reference('service_container'))
    ));
```

Note: Injecting the whole container into a service is generally not a good idea (only inject what you need). In some rare cases, it's necessary when you have a service in the `container` scope that needs a service in the `request` scope.

If you define a controller as a service then you can get the `Request` object without injecting the container by having it passed in as an argument of your action method. See *The Request as a Controller Argument* for details.

3.1.29 How to make your Services use Tags

Several of Symfony2's core services depend on tags to recognize which services should be loaded, notified of events, or handled in some other special way. For example, Twig uses the tag `twig.extension` to load extra extensions.

But you can also use tags in your own bundles. For example in case your service handles a collection of some kind, or implements a "chain", in which several alternative strategies are tried until one of them is successful. In this article I will use the example of a "transport chain", which is a collection of classes implementing `\Swift_Transport`. Using the chain, the Swift mailer may try several ways of transport, until one succeeds. This post focuses mainly on the dependency injection part of the story.

To begin with, define the `TransportChain` class:

```
namespace Acme\MailerBundle;

class TransportChain
{
    private $transports;

    public function __construct()
    {
        $this->transports = array();
    }

    public function addTransport(\Swift_Transport  $transport)
    {
        $this->transports[] = $transport;
    }
}
```

Then, define the chain as a service:

- *YAML*

```
# src/Acme/MailerBundle/Resources/config/services.yml
parameters:
    acme_mailer.transport_chain.class: Acme\MailerBundle\TransportChain

services:
    acme_mailer.transport_chain:
        class: %acme_mailer.transport_chain.class%
```

- *XML*

```
<!-- src/Acme/MailerBundle/Resources/config/services.xml -->

<parameters>
    <parameter key="acme_mailer.transport_chain.class">Acme\MailerBundle\TransportChain</paramet
</parameters>

<services>
    <service id="acme_mailer.transport_chain" class="%acme_mailer.transport_chain.class%" />
</services>
```

- *PHP*

```
// src/Acme/MailerBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('acme_mailer.transport_chain.class', 'Acme\MailerBundle\TransportChain'

$container->setDefinition('acme_mailer.transport_chain', new Definition('%acme_mailer.transport_
```

Define Services with a Custom Tag

Now we want several of the `\Swift_Transport` classes to be instantiated and added to the chain automatically using the `addTransport()` method. As an example we add the following transports as services:

- *YAML*

```
    # src/Acme/MailerBundle/Resources/config/services.yml
    services:
        acme_mailer.transport.smtp:
            class: \Swift_SmtpTransport
            arguments:
                - %mailer_host%
            tags:
                -  { name: acme_mailer.transport }
        acme_mailer.transport.sendmail:
            class: \Swift_SendmailTransport
            tags:
                -  { name: acme_mailer.transport }
```

- *XML*

```
    <!-- src/Acme/MailerBundle/Resources/config/services.xml -->
    <service id="acme_mailer.transport.smtp" class="\Swift_SmtpTransport">
        <argument>%mailer_host%</argument>
        <tag name="acme_mailer.transport" />
    </service>

    <service id="acme_mailer.transport.sendmail" class="\Swift_SendmailTransport">
        <tag name="acme_mailer.transport" />
    </service>
```

- *PHP*

```
    // src/Acme/MailerBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;

    $definitionSmtp = new Definition('\Swift_SmtpTransport', array('%mailer_host%'));
    $definitionSmtp->addTag('acme_mailer.transport');
    $container->setDefinition('acme_mailer.transport.smtp', $definitionSmtp);

    $definitionSendmail = new Definition('\Swift_SendmailTransport');
    $definitionSendmail->addTag('acme_mailer.transport');
    $container->setDefinition('acme_mailer.transport.sendmail', $definitionSendmail);
```

Notice the tags named "acme_mailer.transport". We want the bundle to recognize these transports and add them to the chain all by itself. In order to achieve this, we need to add a `build()` method to the `AcmeMailerBundle` class:

```
namespace Acme\MailerBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
use Symfony\Component\DependencyInjection\ContainerBuilder;

use Acme\MailerBundle\DependencyInjection\Compiler\TransportCompilerPass;

class AcmeMailerBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);

        $container->addCompilerPass(new TransportCompilerPass());
    }
}
```

Create a `CompilerPass`

You will have spotted a reference to the not yet existing `TransportCompilerPass` class. This class will make sure that all services with a tag `acme_mailer.transport` will be added to the `TransportChain` class by calling the `addTransport()` method. The `TransportCompilerPass` should look like this:

```
namespace Acme\MailerBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\Reference;

class TransportCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        if (false === $container->hasDefinition('acme_mailer.transport_chain')) {
            return;
        }

        $definition = $container->getDefinition('acme_mailer.transport_chain');

        foreach ($container->findTaggedServiceIds('acme_mailer.transport') as $id => $attributes) {
            $definition->addMethodCall('addTransport', array(new Reference($id)));
        }
    }
}
```

The `process()` method checks for the existence of the `acme_mailer.transport_chain` service, then looks for all services tagged `acme_mailer.transport`. It adds to the definition of the `acme_mailer.transport_chain` service a call to `addTransport()` for each "acme_mailer.transport" service it has found. The first argument of each of these calls will be the mailer transport service itself.

Note: By convention, tag names consist of the name of the bundle (lowercase, underscores as separators), followed by a dot, and finally the "real" name, so the tag "transport" in the AcmeMailerBundle should be: `acme_mailer.transport`.

The Compiled Service Definition

Adding the compiler pass will result in the automatic generation of the following lines of code in the compiled service container. In case you are working in the "dev" environment, open the file `/cache/dev/appDevDebugProjectContainer.php` and look for the method `getTransportChainService()`. It should look like this:

```
protected function getAcmeMailer_TransportChainService()
{
    $this->services['acme_mailer.transport_chain'] = $instance = new \Acme\MailerBundle\TransportCha

    $instance->addTransport($this->get('acme_mailer.transport.smtp'));
    $instance->addTransport($this->get('acme_mailer.transport.sendmail'));

    return $instance;
}
```

3.1.30 How to use PdoSessionStorage to store Sessions in the Database

The default session storage of Symfony2 writes the session information to file(s). Most medium to large websites use a database to store the session values instead of files, because databases are easier to use and scale in a multi-webserver environment.

Symfony2 has a built-in solution for database session storage called `Symfony\Component\HttpFoundation\SessionStorag`
To use it, you just need to change some parameters in `config.yml` (or the configuration format of your choice):

- *YAML*

```
# app/config/config.yml
framework:
    session:
        # ...
        storage_id:     session.storage.pdo

parameters:
    pdo.db_options:
        db_table:    session
        db_id_col:   session_id
        db_data_col: session_value
        db_time_col: session_time

services:
    pdo:
        class: PDO
        arguments:
            dsn:      "mysql:dbname=mydatabase"
            user:     myuser
            password: mypassword

    session.storage.pdo:
        class:     Symfony\Component\HttpFoundation\SessionStorage\PdoSessionStorage
        arguments: [@pdo, %session.storage.options%, %pdo.db_options%]
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:session storage-id="session.storage.pdo" lifetime="3600" auto-start="true"/>
</framework:config>

<parameters>
    <parameter key="pdo.db_options" type="collection">
        <parameter key="db_table">session</parameter>
        <parameter key="db_id_col">session_id</parameter>
        <parameter key="db_data_col">session_value</parameter>
        <parameter key="db_time_col">session_time</parameter>
    </parameter>
</parameters>

<services>
    <service id="pdo" class="PDO">
        <argument>mysql:dbname=mydatabase</argument>
        <argument>myuser</argument>
        <argument>mypassword</argument>
    </service>
```

```
        <service id="session.storage.pdo" class="Symfony\Component\HttpFoundation\SessionStorage\Pdo
            <argument type="service" id="pdo" />
            <argument>%session.storage.options%</argument>
            <argument>%pdo.db_options%</argument>
        </service>
    </services>
```

- *PHP*

```
    // app/config/config.yml
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    $container->loadFromExtension('framework', array(
        // ...
        'session' => array(
            // ...
            'storage_id' => 'session.storage.pdo',
        ),
    ));

    $container->setParameter('pdo.db_options', array(
        'db_table'      => 'session',
        'db_id_col'     => 'session_id',
        'db_data_col'   => 'session_value',
        'db_time_col'   => 'session_time',
    ));

    $pdoDefinition = new Definition('PDO', array(
        'mysql:dbname=mydatabase',
        'myuser',
        'mypassword',
    ));
    $container->setDefinition('pdo', $pdoDefinition);

    $storageDefinition = new Definition('Symfony\Component\HttpFoundation\SessionStorage\PdoSessionS
        new Reference('pdo'),
        '%session.storage.options%',
        '%pdo.db_options%',
    ));
    $container->setDefinition('session.storage.pdo', $storageDefinition);
```

- `db_table`: The name of the session table in your database
- `db_id_col`: The name of the id column in your session table (VARCHAR(255) or larger)
- `db_data_col`: The name of the value column in your session table (TEXT or CLOB)
- `db_time_col`: The name of the time column in your session table (INTEGER)

Sharing your Database Connection Information

With the given configuration, the database connection settings are defined for the session storage connection only. This is OK when you use a separate database for the session data.

But if you'd like to store the session data in the same database as the rest of your project's data, you can use the connection settings from the parameter.ini by referencing the database-related parameters defined there:

- *YAML*

```
    pdo:
        class: PDO
        arguments:
            - "mysql:dbname=%database_name%"
            - %database_user%
            - %database_password%
```

- *XML*

```
    <service id="pdo" class="PDO">
        <argument>mysql:dbname=%database_name%</argument>
        <argument>%database_user%</argument>
        <argument>%database_password%</argument>
    </service>
```

- *XML*

```
    $pdoDefinition = new Definition('PDO', array(
        'mysql:dbname=%database_name%',
        '%database_user%',
        '%database_password%',
    ));
```

Example SQL Statements

MySQL

The SQL statement for creating the needed database table might look like the following (MySQL):

```
CREATE TABLE `session` (
    `session_id` varchar(255) NOT NULL,
    `session_value` text NOT NULL,
    `session_time` int(11) NOT NULL,
    PRIMARY KEY (`session_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

PostgreSQL

For PostgreSQL, the statement should look like this:

```
CREATE TABLE session (
    session_id character varying(255) NOT NULL,
    session_value text NOT NULL,
    session_time integer NOT NULL,
    CONSTRAINT session_pkey PRIMARY KEY (session_id),
);
```

3.1.31 Bundle Structure and Best Practices

A bundle is a directory that has a well-defined structure and can host anything from classes to controllers and web resources. Even if bundles are very flexible, you should follow some best practices if you want to distribute them.

Bundle Name

A bundle is also a PHP namespace. The namespace must follow the technical interoperability standards for PHP 5.3 namespaces and class names: it starts with a vendor segment, followed by zero or more category segments, and it ends with the namespace short name, which must end with a `Bundle` suffix.

A namespace becomes a bundle as soon as you add a bundle class to it. The bundle class name must follow these simple rules:

- Use only alphanumeric characters and underscores;
- Use a CamelCased name;
- Use a descriptive and short name (no more than 2 words);
- Prefix the name with the concatenation of the vendor (and optionally the category namespaces);
- Suffix the name with `Bundle`.

Here are some valid bundle namespaces and class names:

Namespace	Bundle Class Name
`Acme\Bundle\BlogBundle`	`AcmeBlogBundle`
`Acme\Bundle\Social\BlogBundle`	`AcmeSocialBlogBundle`
`Acme\BlogBundle`	`AcmeBlogBundle`

By convention, the `getName()` method of the bundle class should return the class name.

Note: If you share your bundle publicly, you must use the bundle class name as the name of the repository (`AcmeBlogBundle` and not `BlogBundle` for instance).

Note: Symfony2 core Bundles do not prefix the Bundle class with `Symfony` and always add a `Bundle` subnamespace; for example: `Symfony\Bundle\FrameworkBundle\FrameworkBundle`.

Each bundle has an alias, which is the lower-cased short version of the bundle name using underscores (`acme_hello` for `AcmeHelloBundle`, or `acme_social_blog` for `Acme\Social\BlogBundle` for instance). This alias is used to enforce uniqueness within a bundle (see below for some usage examples).

Directory Structure

The basic directory structure of a `HelloBundle` bundle must read as follows:

```
XXX/...
    HelloBundle/
        HelloBundle.php
        Controller/
        Resources/
            meta/
                LICENSE
            config/
            doc/
                index.rst
            translations/
            views/
            public/
        Tests/
```

The `XXX` directory(ies) reflects the namespace structure of the bundle.

The following files are mandatory:

- `HelloBundle.php`;
- `Resources/meta/LICENSE`: The full license for the code;
- `Resources/doc/index.rst`: The root file for the Bundle documentation.

Note: These conventions ensure that automated tools can rely on this default structure to work.

The depth of sub-directories should be kept to the minimal for most used classes and files (2 levels at a maximum). More levels can be defined for non-strategic, less-used files.

The bundle directory is read-only. If you need to write temporary files, store them under the `cache/` or `log/` directory of the host application. Tools can generate files in the bundle directory structure, but only if the generated files are going to be part of the repository.

The following classes and files have specific emplacements:

Type	Directory
Commands	`Command/`
Controllers	`Controller/`
Service Container Extensions	`DependencyInjection/`
Event Listeners	`EventListener/`
Configuration	`Resources/config/`
Web Resources	`Resources/public/`
Translation files	`Resources/translations/`
Templates	`Resources/views/`
Unit and Functional Tests	`Tests/`

Classes

The bundle directory structure is used as the namespace hierarchy. For instance, a `HelloController` controller is stored in `Bundle/HelloBundle/Controller/HelloController.php` and the fully qualified class name is `Bundle\HelloBundle\Controller\HelloController`.

All classes and files must follow the Symfony2 coding standards.

Some classes should be seen as facades and should be as short as possible, like Commands, Helpers, Listeners, and Controllers.

Classes that connect to the Event Dispatcher should be suffixed with `Listener`.

Exceptions classes should be stored in an `Exception` sub-namespace.

Vendors

A bundle must not embed third-party PHP libraries. It should rely on the standard Symfony2 autoloading instead.

A bundle should not embed third-party libraries written in JavaScript, CSS, or any other language.

Tests

A bundle should come with a test suite written with PHPUnit and stored under the `Tests/` directory. Tests should follow the following principles:

- The test suite must be executable with a simple `phpunit` command run from a sample application;

- The functional tests should only be used to test the response output and some profiling information if you have some;
- The code coverage should at least covers 95% of the code base.

Note: A test suite must not contain `AllTests.php` scripts, but must rely on the existence of a `phpunit.xml.dist` file.

Documentation

All classes and functions must come with full PHPDoc.

Extensive documentation should also be provided in the reStructuredText format, under the `Resources/doc/` directory; the `Resources/doc/index.rst` file is the only mandatory file and must be the entry point for the documentation.

Controllers

As a best practice, controllers in a bundle that's meant to be distributed to others must not extend the `Symfony\Bundle\FrameworkBundle\Controller\Controller` base class. They can implement `Symfony\Component\DependencyInjection\ContainerAwareInterface` or extend `Symfony\Component\DependencyInjection\ContainerAware` instead.

Note: If you have a look at `Symfony\Bundle\FrameworkBundle\Controller\Controller` methods, you will see that they are only nice shortcuts to ease the learning curve.

Routing

If the bundle provides routes, they must be prefixed with the bundle alias. For an AcmeBlogBundle for instance, all routes must be prefixed with `acme_blog_`.

Templates

If a bundle provides templates, they must use Twig. A bundle must not provide a main layout, except if it provides a full working application.

Translation Files

If a bundle provides message translations, they must be defined in the XLIFF format; the domain should be named after the bundle name (`bundle.hello`).

A bundle must not override existing messages from another bundle.

Configuration

To provide more flexibility, a bundle can provide configurable settings by using the Symfony2 built-in mechanisms.

For simple configuration settings, rely on the default `parameters` entry of the Symfony2 configuration. Symfony2 parameters are simple key/value pairs; a value being any valid PHP value. Each parameter name should start with the bundle alias, though this is just a best-practice suggestion. The rest of the parameter name will use a period (`.`) to separate different parts (e.g. `acme_hello.email.from`).

The end user can provide values in any configuration file:

- *YAML*

```
# app/config/config.yml
parameters:
    acme_hello.email.from: fabien@example.com
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
    <parameter key="acme_hello.email.from">fabien@example.com</parameter>
</parameters>
```

- *PHP*

```
// app/config/config.php
$container->setParameter('acme_hello.email.from', 'fabien@example.com');
```

- *INI*

```
[parameters]
acme_hello.email.from = fabien@example.com
```

Retrieve the configuration parameters in your code from the container:

```
$container->getParameter('acme_hello.email.from');
```

Even if this mechanism is simple enough, you are highly encouraged to use the semantic configuration described in the cookbook.

Note: If you are defining services, they should also be prefixed with the bundle alias.

Learn more from the Cookbook

- How to expose a Semantic Configuration for a Bundle

3.1.32 How to use Bundle Inheritance to Override parts of a Bundle

When working with third-party bundles, you'll probably come across a situation where you want to override a file in that third-party bundle with a file in one of your own bundles. Symfony gives you a very convenient way to override things like controllers, templates, and other files in a bundle's `Resources/` directory.

For example, suppose that you're installing the FOSUserBundle, but you want to override its base `layout.html.twig` template, as well as one of its controllers. Suppose also that you have your own `AcmeUserBundle` where you want the overridden files to live. Start by registering the `FOSUserBundle` as the "parent" of your bundle:

```
// src/Acme/UserBundle/AcmeUserBundle.php
namespace Acme\UserBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class AcmeUserBundle extends Bundle
{
    public function getParent()
```

```
    {
        return 'FOSUserBundle';
    }
}
```

By making this simple change, you can now override several parts of the `FOSUserBundle` simply by creating a file with the same name.

Overriding Controllers

Suppose you want to add some functionality to the `registerAction` of a `RegistrationController` that lives inside `FOSUserBundle`. To do so, just create your own `RegistrationController.php` file, override the bundle's original method, and change its functionality:

```
// src/Acme/UserBundle/Controller/RegistrationController.php
namespace Acme\UserBundle\Controller;

use FOS\UserBundle\Controller\RegistrationController as BaseController;

class RegistrationController extends BaseController
{
    public function registerAction()
    {
        $response = parent::registerAction();

        // do custom stuff

        return $response;
    }
}
```

Tip: Depending on how severely you need to change the behavior, you might call `parent::registerAction()` or completely replace its logic with your own.

Note: Overriding controllers in this way only works if the bundle refers to the controller using the standard `FOSUserBundle:Registration:register` syntax in routes and templates. This is the best practice.

Overriding Resources: Templates, Routing, Validation, etc

Most resources can also be overridden, simply by creating a file in the same location as your parent bundle.

For example, it's very common to need to override the `FOSUserBundle`'s `layout.html.twig` template so that it uses your application's base layout. Since the file lives at `Resources/views/layout.html.twig` in the `FOSUserBundle`, you can create your own file in the same location of `AcmeUserBundle`. Symfony will ignore the file that lives inside the `FOSUserBundle` entirely, and use your file instead.

The same goes for routing files, validation configuration and other resources.

Note: The overriding of resources only works when you refer to resources with the `@FosUserBundle/Resources/config/routing/security.xml` method. If you refer to resources without using the @BundleName shortcut, they can't be overridden in this way.

Caution: Translation files do not work in the same way as described above. All translation files are accumulated into a set of "pools" (one for each) domain. Symfony loads translation files from bundles first (in the order that the bundles are initialized) and then from your `app/Resources` directory. If the same translation is specified in two resources, the translation from the resource that's loaded last will win.

3.1.33 How to Override any Part of a Bundle

This article has not been written yet, but will soon. If you're interested in writing this entry, see Contributing to the Documentation.

This topic is meant to show how you can override each and every part of a bundle, both from your application and from other bundles. This may include:

- Templates
- Routing
- Controllers
- Services & Configuration
- Entities & Entity mapping
- Forms
- Validation metadata

In some cases, this may talk about the best practices that a bundle must use in order for certain pieces to be overridable (or easily overridable). We may also talk about how certain pieces *aren't* really overridable, but your best approach at solving your problems anyways.

3.1.34 How to expose a Semantic Configuration for a Bundle

If you open your application configuration file (usually `app/config/config.yml`), you'll see a number of different configuration "namespaces", such as `framework`, `twig`, and `doctrine`. Each of these configures a specific bundle, allowing you to configure things at a high level and then let the bundle make all the low-level, complex changes that result.

For example, the following tells the `FrameworkBundle` to enable the form integration, which involves the defining of quite a few services as well as integration of other related components:

- *YAML*

```
framework:
    # ...
    form:             true
```

- *XML*

```
<framework:config>
    <framework:form />
</framework:config>
```

- *PHP*

```
$container->loadFromExtension('framework', array(
    // ...
    'form'            => true,
```

```
        // ...
    ));
```

When you create a bundle, you have two choices on how to handle configuration:

1. **Normal Service Configuration** (*easy*):

   You can specify your services in a configuration file (e.g. `services.yml`) that lives in your bundle and then import it from your main application configuration. This is really easy, quick and totally effective. If you make use of *parameters*, then you still have the flexibility to customize your bundle from your application configuration. See "*Importing Configuration with imports*" for more details.

2. **Exposing Semantic Configuration** (*advanced*):

   This is the way configuration is done with the core bundles (as described above). The basic idea is that, instead of having the user override individual parameters, you let the user configure just a few, specifically created options. As the bundle developer, you then parse through that configuration and load services inside an "Extension" class. With this method, you won't need to import any configuration resources from your main application configuration: the Extension class can handle all of this.

The second option - which you'll learn about in this article - is much more flexible, but also requires more time to setup. If you're wondering which method you should use, it's probably a good idea to start with method #1, and then change to #2 later if you need to.

The second method has several specific advantages:

- Much more powerful than simply defining parameters: a specific option value might trigger the creation of many service definitions;
- Ability to have configuration hierarchy
- Smart merging when several configuration files (e.g. `config_dev.yml` and `config.yml`) override each other's configuration;
- Configuration validation (if you use a *Configuration Class*);
- IDE auto-completion when you create an XSD and developers use XML.

Overriding bundle parameters

If a Bundle provides an Extension class, then you should generally *not* override any service container parameters from that bundle. The idea is that if an Extension class is present, every setting that should be configurable should be present in the configuration made available by that class. In other words the extension class defines all the publicly supported configuration settings for which backward compatibility will be maintained.

Creating an Extension Class

If you do choose to expose a semantic configuration for your bundle, you'll first need to create a new "Extension" class, which will handle the process. This class should live in the `DependencyInjection` directory of your bundle and its name should be constructed by replacing the `Bundle` suffix of the Bundle class name with `Extension`. For example, the Extension class of `AcmeHelloBundle` would be called `AcmeHelloExtension`:

```
// Acme/HelloBundle/DependencyInjection/AcmeHelloExtension.php
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class AcmeHelloExtension extends Extension
```

```
{
    public function load(array $configs, ContainerBuilder $container)
    {
        // where all of the heavy logic is done
    }

    public function getXsdValidationBasePath()
    {
        return __DIR__.'/../Resources/config/';
    }

    public function getNamespace()
    {
        return 'http://www.example.com/symfony/schema/';
    }
}
```

Note: The `getXsdValidationBasePath` and `getNamespace` methods are only required if the bundle provides optional XSD's for the configuration.

The presence of the previous class means that you can now define an `acme_hello` configuration namespace in any configuration file. The namespace `acme_hello` is constructed from the extension's class name by removing the word `Extension` and then lowercasing and underscoring the rest of the name. In other words, `AcmeHelloExtension` becomes `acme_hello`.

You can begin specifying configuration under this namespace immediately:

- *YAML*

```
# app/config/config.yml
acme_hello: ~
```

- *XML*

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:acme_hello="http://www.example.com/symfony/schema/"
    xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/symfony/sc

   <acme_hello:config />
   ...

</container>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('acme_hello', array());
```

Tip: If you follow the naming conventions laid out above, then the `load()` method of your extension code is always called as long as your bundle is registered in the Kernel. In other words, even if the user does not provide any configuration (i.e. the `acme_hello` entry doesn't even appear), the `load()` method will be called and passed an empty `$configs` array. You can still provide some sensible defaults for your bundle if you want.

Parsing the `$configs` Array

Whenever a user includes the `acme_hello` namespace in a configuration file, the configuration under it is added to an array of configurations and passed to the `load()` method of your extension (Symfony2 automatically converts XML and YAML to an array).

Take the following configuration:

- *YAML*

```
# app/config/config.yml
acme_hello:
    foo: fooValue
    bar: barValue
```

- *XML*

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:acme_hello="http://www.example.com/symfony/schema/"
    xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/symfony/sc

    <acme_hello:config foo="fooValue">
        <acme_hello:bar>barValue</acme_hello:bar>
    </acme_hello:config>

</container>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('acme_hello', array(
    'foo' => 'fooValue',
    'bar' => 'barValue',
));
```

The array passed to your `load()` method will look like this:

```
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    )
)
```

Notice that this is an *array of arrays*, not just a single flat array of the configuration values. This is intentional. For example, if `acme_hello` appears in another configuration file - say `config_dev.yml` - with different values beneath it, then the incoming array might look like this:

```
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    ),
    array(
        'foo' => 'fooDevValue',
        'baz' => 'newConfigEntry',
```

```
    ),
)
```

The order of the two arrays depends on which one is set first.

It's your job, then, to decide how these configurations should be merged together. You might, for example, have later values override previous values or somehow merge them together.

Later, in the *Configuration Class* section, you'll learn of a truly robust way to handle this. But for now, you might just merge them manually:

```
public function load(array $configs, ContainerBuilder $container)
{
    $config = array();
    foreach ($configs as $subConfig) {
        $config = array_merge($config, $subConfig);
    }

    // now use the flat $config array
}
```

Caution: Make sure the above merging technique makes sense for your bundle. This is just an example, and you should be careful to not use it blindly.

Using the `load()` Method

Within `load()`, the `$container` variable refers to a container that only knows about this namespace configuration (i.e. it doesn't contain service information loaded from other bundles). The goal of the `load()` method is to manipulate the container, adding and configuring any methods or services needed by your bundle.

Loading External Configuration Resources

One common thing to do is to load an external configuration file that may contain the bulk of the services needed by your bundle. For example, suppose you have a `services.xml` file that holds much of your bundle's service configuration:

```
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
use Symfony\Component\Config\FileLocator;

public function load(array $configs, ContainerBuilder $container)
{
    // prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
    $loader->load('services.xml');
}
```

You might even do this conditionally, based on one of the configuration values. For example, suppose you only want to load a set of services if an `enabled` option is passed and set to true:

```
public function load(array $configs, ContainerBuilder $container)
{
    // prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
```

```
    if (isset($config['enabled']) && $config['enabled']) {
        $loader->load('services.xml');
    }
}
```

Configuring Services and Setting Parameters

Once you've loaded some service configuration, you may need to modify the configuration based on some of the input values. For example, suppose you have a service whose first argument is some string "type" that it will use internally. You'd like this to be easily configured by the bundle user, so in your service configuration file (e.g. `services.xml`), you define this service and use a blank parameter - `acme_hello.my_service_type` - as its first argument:

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services

    <parameters>
        <parameter key="acme_hello.my_service_type" />
    </parameters>

    <services>
        <service id="acme_hello.my_service" class="Acme\HelloBundle\MyService">
            <argument>%acme_hello.my_service_type%</argument>
        </service>
    </services>
</container>
```

But why would you define an empty parameter and then pass it to your service? The answer is that you'll set this parameter in your extension class, based on the incoming configuration values. Suppose, for example, that you want to allow the user to define this *type* option under a key called `my_type`. Add the following to the `load()` method to do this:

```
public function load(array $configs, ContainerBuilder $container)
{
    // prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
    $loader->load('services.xml');

    if (!isset($config['my_type'])) {
        throw new \InvalidArgumentException('The "my_type" option must be set');
    }

    $container->setParameter('acme_hello.my_service_type', $config['my_type']);
}
```

Now, the user can effectively configure the service by specifying the `my_type` configuration value:

- *YAML*

```
    # app/config/config.yml
    acme_hello:
        my_type: foo
        # ...
```

- *XML*

```xml
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:acme_hello="http://www.example.com/symfony/schema/"
    xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/symfony/sc

    <acme_hello:config my_type="foo">
        <!-- ... -->
    </acme_hello:config>

</container>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('acme_hello', array(
    'my_type' => 'foo',
    // ...
));
```

Global Parameters

When you're configuring the container, be aware that you have the following global parameters available to use:

- `kernel.name`
- `kernel.environment`
- `kernel.debug`
- `kernel.root_dir`
- `kernel.cache_dir`
- `kernel.logs_dir`
- `kernel.bundle_dirs`
- `kernel.bundles`
- `kernel.charset`

Caution: All parameter and service names starting with a _ are reserved for the framework, and new ones must not be defined by bundles.

Validation and Merging with a Configuration Class

So far, you've done the merging of your configuration arrays by hand and are checking for the presence of config values manually using the `isset()` PHP function. An optional *Configuration* system is also available which can help with merging, validation, default values, and format normalization.

Note: Format normalization refers to the fact that certain formats - largely XML - result in slightly different configuration arrays and that these arrays need to be "normalized" to match everything else.

To take advantage of this system, you'll create a `Configuration` class and build a tree that defines your configuration in that class:

```
// src/Acme/HelloBundle/DependencyExtension/Configuration.php
namespace Acme\HelloBundle\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('acme_hello');

        $rootNode
            ->children()
                ->scalarNode('my_type')->defaultValue('bar')->end()
            ->end()
        ;

        return $treeBuilder;
    }
```

This is a *very* simple example, but you can now use this class in your `load()` method to merge your configuration and force validation. If any options other than `my_type` are passed, the user will be notified with an exception that an unsupported option was passed:

```
use Symfony\Component\Config\Definition\Processor;
// ...

public function load(array $configs, ContainerBuilder $container)
{
    $processor = new Processor();
    $configuration = new Configuration();
    $config = $processor->processConfiguration($configuration, $configs);

    // ...
}
```

The `processConfiguration()` method uses the configuration tree you've defined in the `Configuration` class to validate, normalize and merge all of the configuration arrays together.

The `Configuration` class can be much more complicated than shown here, supporting array nodes, "prototype" nodes, advanced validation, XML-specific normalization and advanced merging. The best way to see this in action is to checkout out some of the core Configuration classes, such as the one from the FrameworkBundle Configuration or the TwigBundle Configuration.

Default Configuration Dump

New in version 2.1: The `config:dump-reference` command was added in Symfony 2.1

The `config:dump-reference` command allows a bundle's default configuration to be output to the console in yaml.

As long as your bundle's configuration is located in the standard location (`YourBundle\DependencyInjection\Configuration`) and does not have a `__constructor()` it will work automatically. If you have a something different your `Extension` class will have to override the `Extension::getConfiguration()` method. Have it return an instance of your `Configuration`.

Comments and examples can be added to your configuration nodes using the `->setInfo()` and `->setExample()` methods:

```
// src/Acme/HelloBundle/DependencyExtension/Configuration.php
namespace Acme\HelloBundle\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('acme_hello');

        $rootNode
            ->children()
                ->scalarNode('my_type')
                    ->defaultValue('bar')
                    ->setInfo('what my_type configures')
                    ->setExample('example setting')
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
```

This text appears as yaml comments in the output of the `config:dump-reference` command.

Extension Conventions

When creating an extension, follow these simple conventions:

- The extension must be stored in the `DependencyInjection` sub-namespace;
- The extension must be named after the bundle name and suffixed with `Extension` (`AcmeHelloExtension` for `AcmeHelloBundle`);
- The extension should provide an XSD schema.

If you follow these simple conventions, your extensions will be registered automatically by Symfony2. If not, override the Bundle **:method:‘Symfony\\Component\\HttpKernel\\Bundle\\Bundle::build‘** method in your bundle:

```
use Acme\HelloBundle\DependencyInjection\UnconventionalExtensionClass;

class AcmeHelloBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);

        // register extensions that do not follow the conventions manually
        $container->registerExtension(new UnconventionalExtensionClass());
    }
}
```

In this case, the extension class must also implement a `getAlias()` method and return a unique alias named after the bundle (e.g. `acme_hello`). This is required because the class name doesn't follow the standards by ending in `Extension`.

Additionally, the `load()` method of your extension will *only* be called if the user specifies the `acme_hello` alias in at least one configuration file. Once again, this is because the Extension class doesn't follow the standards set out above, so nothing happens automatically.

3.1.35 How to send an Email

Sending emails is a classic task for any web application and one that has special complications and potential pitfalls. Instead of recreating the wheel, one solution to send emails is to use the `SwiftmailerBundle`, which leverages the power of the Swiftmailer library.

Note: Don't forget to enable the bundle in your kernel before using it:

```
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
    );

    // ...
}
```

Configuration

Before using Swiftmailer, be sure to include its configuration. The only mandatory configuration parameter is `transport`:

- *YAML*

```
# app/config/config.yml
swiftmailer:
    transport:  smtp
    encryption: ssl
    auth_mode:  login
    host:       smtp.gmail.com
    username:   your_username
    password:   your_password
```

- *XML*

```
<!-- app/config/config.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    transport="smtp"
    encryption="ssl"
    auth-mode="login"
    host="smtp.gmail.com"
```

```
        username="your_username"
        password="your_password" />
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('swiftmailer', array(
        'transport'  => "smtp",
        'encryption' => "ssl",
        'auth_mode'  => "login",
        'host'       => "smtp.gmail.com",
        'username'   => "your_username",
        'password'   => "your_password",
    ));
```

The majority of the Swiftmailer configuration deals with how the messages themselves should be delivered.

The following configuration attributes are available:

- `transport` (`smtp`, `mail`, `sendmail`, or `gmail`)
- `username`
- `password`
- `host`
- `port`
- `encryption` (`tls`, or `ssl`)
- `auth_mode` (`plain`, `login`, or `cram-md5`)
- `spool`
  - `type` (how to queue the messages, only `file` is supported currently)
  - `path` (where to store the messages)
- `delivery_address` (an email address where to send ALL emails)
- `disable_delivery` (set to true to disable delivery completely)

Sending Emails

The Swiftmailer library works by creating, configuring and then sending `Swift_Message` objects. The "mailer" is responsible for the actual delivery of the message and is accessible via the `mailer` service. Overall, sending an email is pretty straightforward:

```
public function indexAction($name)
{
    $message = \Swift_Message::newInstance()
        ->setSubject('Hello Email')
        ->setFrom('send@example.com')
        ->setTo('recipient@example.com')
        ->setBody($this->renderView('HelloBundle:Hello:email.txt.twig', array('name' => $name)))
    ;
    $this->get('mailer')->send($message);

    return $this->render(...);
}
```

To keep things decoupled, the email body has been stored in a template and rendered with the `renderView()` method.

The `$message` object supports many more options, such as including attachments, adding HTML content, and much more. Fortunately, Swiftmailer covers the topic of Creating Messages in great detail in its documentation.

Tip: Several other cookbook articles are available related to sending emails in Symfony2:

- How to use Gmail to send Emails
- How to Work with Emails During Development
- How to Spool Email

3.1.36 How to use Gmail to send Emails

During development, instead of using a regular SMTP server to send emails, you might find using Gmail easier and more practical. The Swiftmailer bundle makes it really easy.

Tip: Instead of using your regular Gmail account, it's of course recommended that you create a special account.

In the development configuration file, change the `transport` setting to `gmail` and set the `username` and `password` to the Google credentials:

- *YAML*

```
# app/config/config_dev.yml
swiftmailer:
    transport: gmail
    username:  your_gmail_username
    password:  your_gmail_password
```

- *XML*

```
<!-- app/config/config_dev.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    transport="gmail"
    username="your_gmail_username"
    password="your_gmail_password" />
```

- *PHP*

```
// app/config/config_dev.php
$container->loadFromExtension('swiftmailer', array(
    'transport' => "gmail",
    'username'  => "your_gmail_username",
    'password'  => "your_gmail_password",
));
```

You're done!

Note: The `gmail` transport is simply a shortcut that uses the `smtp` transport and sets `encryption`, `auth_mode`

and `host` to work with Gmail.

3.1.37 How to Work with Emails During Development

When developing an application which sends email, you will often not want to actually send the email to the specified recipient during development. If you are using the `SwiftmailerBundle` with Symfony2, you can easily achieve this through configuration settings without having to make any changes to your application's code at all. There are two main choices when it comes to handling email during development: (a) disabling the sending of email altogether or (b) sending all email to a specific address.

Disabling Sending

You can disable sending email by setting the `disable_delivery` option to `true`. This is the default in the `test` environment in the Standard distribution. If you do this in the `test` specific config then email will not be sent when you run tests, but will continue to be sent in the `prod` and `dev` environments:

- *YAML*

```
# app/config/config_test.yml
swiftmailer:
    disable_delivery:  true
```

- *XML*

```
<!-- app/config/config_test.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    disable-delivery="true" />
```

- *PHP*

```
// app/config/config_test.php
$container->loadFromExtension('swiftmailer', array(
    'disable_delivery'  => "true",
));
```

If you'd also like to disable deliver in the `dev` environment, simply add this same configuration to the `config_dev.yml` file.

Sending to a Specified Address

You can also choose to have all email sent to a specific address, instead of the address actually specified when sending the message. This can be done via the `delivery_address` option:

- *YAML*

```
# app/config/config_dev.yml
swiftmailer:
    delivery_address:  dev@example.com
```

- *XML*

```
<!-- app/config/config_dev.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    delivery-address="dev@example.com" />
```

- *PHP*

```
// app/config/config_dev.php
$container->loadFromExtension('swiftmailer', array(
    'delivery_address'  => "dev@example.com",
));
```

Now, suppose you're sending an email to `recipient@example.com`.

```
public function indexAction($name)
{
    $message = \Swift_Message::newInstance()
        ->setSubject('Hello Email')
        ->setFrom('send@example.com')
        ->setTo('recipient@example.com')
        ->setBody($this->renderView('HelloBundle:Hello:email.txt.twig', array('name' => $name)))
    ;
    $this->get('mailer')->send($message);

    return $this->render(...);
}
```

In the `dev` environment, the email will instead be sent to `dev@example.com`. Swiftmailer will add an extra header to the email, `X-Swift-To`, containing the replaced address, so you can still see who it would have been sent to.

Note: In addition to the `to` addresses, this will also stop the email being sent to any `CC` and `BCC` addresses set for it. Swiftmailer will add additional headers to the email with the overridden addresses in them. These are `X-Swift-Cc` and `X-Swift-Bcc` for the `CC` and `BCC` addresses respectively.

Viewing from the Web Debug Toolbar

You can view any email sent during a single response when you are in the `dev` environment using the Web Debug Toolbar. The email icon in the toolbar will show how many emails were sent. If you click it, a report will open showing the details of the sent emails.

If you're sending an email and then immediately redirecting to another page, the web debug toolbar will not display an email icon or a report on the next page.

Instead, you can set the `intercept_redirects` option to `true` in the `config_dev.yml` file, which will cause the redirect to stop and allow you to open the report with details of the sent emails.

Tip: Alternatively, you can open the profiler after the redirect and search by the submit URL used on previous request (e.g. `/contact/handle`). The profiler's search feature allows you to load the profiler information for any past requests.

- *YAML*

```yaml
# app/config/config_dev.yml
web_profiler:
    intercept_redirects: true
```

- *XML*

```xml
<!-- app/config/config_dev.xml -->

<!-- xmlns:webprofiler="http://symfony.com/schema/dic/webprofiler" -->
<!-- xsi:schemaLocation="http://symfony.com/schema/dic/webprofiler http://symfony.com/schema/dic

<webprofiler:config
    intercept-redirects="true"
/>
```

- *PHP*

```php
// app/config/config_dev.php
$container->loadFromExtension('web_profiler', array(
    'intercept_redirects' => 'true',
));
```

3.1.38 How to Spool Email

When you are using the `SwiftmailerBundle` to send an email from a Symfony2 application, it will default to sending the email immediately. You may, however, want to avoid the performance hit of the communication between `Swiftmailer` and the email transport, which could cause the user to wait for the next page to load while the email is sending. This can be avoided by choosing to "spool" the emails instead of sending them directly. This means that `Swiftmailer` does not attempt to send the email but instead saves the message to somewhere such as a file. Another process can then read from the spool and take care of sending the emails in the spool. Currently only spooling to file is supported by `Swiftmailer`.

In order to use the spool, use the following configuration:

- *YAML*

```yaml
# app/config/config.yml
swiftmailer:
    # ...
    spool:
        type: file
        path: /path/to/spool
```

- *XML*

```xml
<!-- app/config/config.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config>
     <swiftmailer:spool
         type="file"
         path="/path/to/spool" />
</swiftmailer:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('swiftmailer', array(
     // ...
    'spool' => array(
        'type' => 'file',
        'path' => '/path/to/spool',
    )
));
```

Tip: If you want to store the spool somewhere with your project directory, remember that you can use the *%kernel.root_dir%* parameter to reference the project's root:

```
path: %kernel.root_dir%/spool
```

Now, when your app sends an email, it will not actually be sent but instead added to the spool. Sending the messages from the spool is done separately. There is a console command to send the messages in the spool:

```
php app/console swiftmailer:spool:send
```

It has an option to limit the number of messages to be sent:

```
php app/console swiftmailer:spool:send --message-limit=10
```

You can also set the time limit in seconds:

```
php app/console swiftmailer:spool:send --time-limit=10
```

Of course you will not want to run this manually in reality. Instead, the console command should be triggered by a cron job or scheduled task and run at a regular interval.

3.1.39 How to simulate HTTP Authentication in a Functional Test

If your application needs HTTP authentication, pass the username and password as server variables to `createClient()`:

```
$client = static::createClient(array(), array(
    'PHP_AUTH_USER' => 'username',
    'PHP_AUTH_PW'   => 'pa$$word',
));
```

You can also override it on a per request basis:

```
$client->request('DELETE', '/post/12', array(), array(
    'PHP_AUTH_USER' => 'username',
    'PHP_AUTH_PW'   => 'pa$$word',
));
```

3.1.40 How to test the Interaction of several Clients

If you need to simulate an interaction between different Clients (think of a chat for instance), create several Clients:

```
$harry = static::createClient();
$sally = static::createClient();
```

```
$harry->request('POST', '/say/sally/Hello');
$sally->request('GET', '/messages');

$this->assertEquals(201, $harry->getResponse()->getStatusCode());
$this->assertRegExp('/Hello/', $sally->getResponse()->getContent());
```

This works except when your code maintains a global state or if it depends on third-party libraries that has some kind of global state. In such a case, you can insulate your clients:

```
$harry = static::createClient();
$sally = static::createClient();

$harry->insulate();
$sally->insulate();

$harry->request('POST', '/say/sally/Hello');
$sally->request('GET', '/messages');

$this->assertEquals(201, $harry->getResponse()->getStatusCode());
$this->assertRegExp('/Hello/', $sally->getResponse()->getContent());
```

Insulated clients transparently execute their requests in a dedicated and clean PHP process, thus avoiding any side-effects.

Tip: As an insulated client is slower, you can keep one client in the main process, and insulate the other ones.

3.1.41 How to use the Profiler in a Functional Test

It's highly recommended that a functional test only tests the Response. But if you write functional tests that monitor your production servers, you might want to write tests on the profiling data as it gives you a great way to check various things and enforce some metrics.

The Symfony2 `Profiler` gathers a lot of data for each request. Use this data to check the number of database calls, the time spent in the framework, ... But before writing assertions, always check that the profiler is indeed available (it is enabled by default in the `test` environment):

```
class HelloControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();
        $crawler = $client->request('GET', '/hello/Fabien');

        // Write some assertions about the Response
        // ...

        // Check that the profiler is enabled
        if ($profile = $client->getProfile()) {
            // check the number of requests
            $this->assertTrue($profile->getCollector('db')->getQueryCount() < 10);

            // check the time spent in the framework
            $this->assertTrue( $profile->getCollector('timer')->getTime() < 0.5);
        }
    }
}
```

If a test fails because of profiling data (too many DB queries for instance), you might want to use the Web Profiler to analyze the request after the tests finish. It's easy to achieve if you embed the token in the error message:

```
$this->assertTrue(
    $profile->get('db')->getQueryCount() < 30,
    sprintf('Checks that query count is less than 30 (token %s)', $profile->getToken())
);
```

Caution: The profiler store can be different depending on the environment (especially if you use the SQLite store, which is the default configured one).

Note: The profiler information is available even if you insulate the client or if you use an HTTP layer for your tests.

Tip: Read the API for built-in data collectors to learn more about their interfaces.

3.1.42 How to test Doctrine Repositories

Unit testing Doctrine repositories in a Symfony project is not a straightforward task. Indeed, to load a repository you need to load your entities, an entity manager, and some other stuff like a connection.

To test your repository, you have two different options:

1. **Functional test**: This includes using a real database connection with real database objects. It's easy to setup and can test anything, but is slower to execute. See *Functional Testing*.
2. **Unit test**: Unit testing is faster to run and more precise in how you test. It does require a little bit more setup, which is covered in this document. It can also only test methods that, for example, build queries, not methods that actually execute them.

Unit Testing

As Symfony and Doctrine share the same testing framework, it's quite easy to implement unit tests in your Symfony project. The ORM comes with its own set of tools to ease the unit testing and mocking of everything you need, such as a connection, an entity manager, etc. By using the testing components provided by Doctrine - along with some basic setup - you can leverage Doctrine's tools to unit test your repositories.

Keep in mind that if you want to test the actual execution of your queries, you'll need a functional test (see *Functional Testing*). Unit testing is only possible when testing a method that builds a query.

Setup

First, you need to add the DoctrineTests namespace to your autoloader:

```
// app/autoload.php
$loader->registerNamespaces(array(
    //...
    'Doctrine\\Tests'               => __DIR__.'/../vendor/doctrine/tests',
));
```

Next, you will need to setup an entity manager in each test so that Doctrine will be able to load your entities and repositories for you.

As Doctrine is not able by default to load annotation metadata from your entities, you'll need to configure the annotation reader to be able to parse and load the entities:

```
// src/Acme/ProductBundle/Tests/Entity/ProductRepositoryTest.php
namespace Acme\ProductBundle\Tests\Entity;

use Doctrine\Tests\OrmTestCase;
use Doctrine\Common\Annotations\AnnotationReader;
use Doctrine\ORM\Mapping\Driver\DriverChain;
use Doctrine\ORM\Mapping\Driver\AnnotationDriver;

class ProductRepositoryTest extends OrmTestCase
{
    private $_em;

    protected function setUp()
    {
        $reader = new AnnotationReader();
        $reader->setIgnoreNotImportedAnnotations(true);
        $reader->setEnableParsePhpImports(true);

        $metadataDriver = new AnnotationDriver(
            $reader,
            // provide the namespace of the entities you want to tests
            'Acme\\ProductBundle\\Entity'
        );

        $this->_em = $this->_getTestEntityManager();

        $this->_em->getConfiguration()
            ->setMetadataDriverImpl($metadataDriver);

        // allows you to use the AcmeProductBundle:Product syntax
        $this->_em->getConfiguration()->setEntityNamespaces(array(
            'AcmeProductBundle' => 'Acme\\ProductBundle\\Entity'
        ));
    }
}
```

If you look at the code, you can notice:

- You extend from `\Doctrine\Tests\OrmTestCase`, which provide useful methods for unit testing;
- You need to setup the `AnnotationReader` to be able to parse and load the entities;
- You create the entity manager by calling `_getTestEntityManager`, which returns a mocked entity manager with a mocked connection.

That's it! You're ready to write units tests for your Doctrine repositories.

Writing your Unit Test

Remember that Doctrine repository methods can only be tested if they are building and returning a query (but not actually executing a query). Take the following example:

```
// src/Acme/StoreBundle/Entity/ProductRepository
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
```

```
{
    public function createSearchByNameQueryBuilder($name)
    {
        return $this->createQueryBuilder('p')
            ->where('p.name LIKE :name')
            ->setParameter('name', $name);
    }
}
```

In this example, the method is returning a `QueryBuilder` instance. You can test the result of this method in a variety of ways:

```
class ProductRepositoryTest extends \Doctrine\Tests\OrmTestCase
{
    /* ... */

    public function testCreateSearchByNameQueryBuilder()
    {
        $queryBuilder = $this->_em->getRepository('AcmeProductBundle:Product')
            ->createSearchByNameQueryBuilder('foo');

        $this->assertEquals('p.name LIKE :name', (string) $queryBuilder->getDqlPart('where'));
        $this->assertEquals(array('name' => 'foo'), $queryBuilder->getParameters());
    }
 }
```

In this test, you dissect the `QueryBuilder` object, looking that each part is as you'd expect. If you were adding other things to the query builder, you might check the dql parts: `select`, `from`, `join`, `set`, `groupBy`, `having`, or `orderBy`.

If you only have a raw `Query` object or prefer to test the actual query, you can test the DQL query string directly:

```
public function testCreateSearchByNameQueryBuilder()
{
    $queryBuilder = $this->_em->getRepository('AcmeProductBundle:Product')
        ->createSearchByNameQueryBuilder('foo');

    $query = $queryBuilder->getQuery();

    // test DQL
    $this->assertEquals(
        'SELECT p FROM Acme\ProductBundle\Entity\Product p WHERE p.name LIKE :name',
        $query->getDql()
    );
}
```

Functional Testing

If you need to actually execute a query, you will need to boot the kernel to get a valid connection. In this case, you'll extend the `WebTestCase`, which makes all of this quite easy:

```
// src/Acme/ProductBundle/Tests/Entity/ProductRepositoryFunctionalTest.php
namespace Acme\ProductBundle\Tests\Entity;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class ProductRepositoryFunctionalTest extends WebTestCase
{
```

```
    /**
     * @var \Doctrine\ORM\EntityManager
     */
    private $_em;

    public function setUp()
    {
            $kernel = static::createKernel();
            $kernel->boot();
        $this->_em = $kernel->getContainer()
            ->get('doctrine.orm.entity_manager');
    }

    public function testProductByCategoryName()
    {
        $results = $this->_em->getRepository('AcmeProductBundle:Product')
            ->searchProductsByNameQuery('foo')
            ->getResult();

        $this->assertEquals(count($results), 1);
    }
}
```

3.1.43 How to add "Remember Me" Login Functionality

Once a user is authenticated, their credentials are typically stored in the session. This means that when the session ends they will be logged out and have to provide their login details again next time they wish to access the application. You can allow users to choose to stay logged in for longer than the session lasts using a cookie with the `remember_me` firewall option. The firewall needs to have a secret key configured, which is used to encrypt the cookie's content. It also has several options with default values which are shown here:

- *YAML*

```
    # app/config/security.yml
    firewalls:
        main:
            remember_me:
                key:      aSecretKey
                lifetime: 3600
                path:     /
                domain:   ~ # Defaults to the current domain from $_SERVER
```

- *XML*

```
    <!-- app/config/security.xml -->
    <config>
        <firewall>
            <remember-me
                key="aSecretKey"
                lifetime="3600"
                path="/"
                domain="" <!-- Defaults to the current domain from $_SERVER -->
            />
        </firewall>
    </config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('remember_me' => array(
            'key'                   => 'aSecretKey',
            'lifetime'              => 3600,
            'path'                  => '/',
            'domain'                => '', // Defaults to the current domain from $_SERVER
        )),
    ),
));
```

It's a good idea to provide the user with the option to use or not use the remember me functionality, as it will not always be appropriate. The usual way of doing this is to add a checkbox to the login form. By giving the checkbox the name `_remember_me`, the cookie will automatically be set when the checkbox is checked and the user successfully logs in. So, your specific login form might ultimately look like this:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="checkbox" id="remember_me" name="_remember_me" checked />
    <label for="remember_me">Keep me logged in</label>

    <input type="submit" name="login" />
</form>
```

- *PHP*

```
<?php // src/Acme/SecurityBundle/Resources/views/Security/login.html.php ?>
<?php if ($error): ?>
    <div><?php echo $error->getMessage() ?></div>
<?php endif; ?>

<form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username"
           name="_username" value="<?php echo $last_username ?>" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="checkbox" id="remember_me" name="_remember_me" checked />
    <label for="remember_me">Keep me logged in</label>

    <input type="submit" name="login" />
</form>
```

The user will then automatically be logged in on subsequent visits while the cookie remains valid.

Forcing the User to Re-authenticate before accessing certain Resources

When the user returns to your site, he/she is authenticated automatically based on the information stored in the remember me cookie. This allows the user to access protected resources as if the user had actually authenticated upon visiting the site.

In some cases, however, you may want to force the user to actually re-authenticate before accessing certain resources. For example, you might allow a "remember me" user to see basic account information, but then require them to actually re-authenticate before modifying that information.

The security component provides an easy way to do this. In addition to roles explicitly assigned to them, users are automatically given one of the following roles depending on how they are authenticated:

- `IS_AUTHENTICATED_ANONYMOUSLY` - automatically assigned to a user who is in a firewall protected part of the site but who has not actually logged in. This is only possible if anonymous access has been allowed.
- `IS_AUTHENTICATED_REMEMBERED` - automatically assigned to a user who was authenticated via a remember me cookie.
- `IS_AUTHENTICATED_FULLY` - automatically assigned to a user that has provided their login details during the current session.

You can use these to control access beyond the explicitly assigned roles.

Note: If you have the `IS_AUTHENTICATED_REMEMBERED` role, then you also have the `IS_AUTHENTICATED_ANONYMOUSLY` role. If you have the `IS_AUTHENTICATED_FULLY` role, then you also have the other two roles. In other words, these roles represent three levels of increasing "strength" of authentication.

You can use these additional roles for finer grained control over access to parts of a site. For example, you may want you user to be able to view their account at `/account` when authenticated by cookie but to have to provide their login details to be able to edit the account details. You can do this by securing specific controller actions using these roles. The edit action in the controller could be secured using the service context.

In the following example, the action is only allowed if the user has the `IS_AUTHENTICATED_FULLY` role.

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException
// ...

public function editAction()
{
    if (false === $this->get('security.context')->isGranted(
        'IS_AUTHENTICATED_FULLY'
    )) {
        throw new AccessDeniedException();
    }

    // ...
}
```

You can also choose to install and use the optional JMSSecurityExtraBundle, which can secure your controller using annotations:

```
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Secure(roles="IS_AUTHENTICATED_FULLY")
 */
public function editAction($name)
{
```

```
    // ...
}
```

Tip: If you also had an access control in your security configuration that required the user to have a `ROLE_USER` role in order to access any of the account area, then you'd have the following situation:

- If a non-authenticated (or anonymously authenticated user) tries to access the account area, the user will be asked to authenticate.
- Once the user has entered his username and password, assuming the user receives the `ROLE_USER` role per your configuration, the user will have the `IS_AUTHENTICATED_FULLY` role and be able to access any page in the account section, including the `editAction` controller.
- If the user's session ends, when the user returns to the site, he will be able to access every account page - except for the edit page - without being forced to re-authenticate. However, when he tries to access the `editAction` controller, he will be forced to re-authenticate, since he is not, yet, fully authenticated.

For more information on securing services or methods in this way, see How to secure any Service or Method in your Application.

3.1.44 How to implement your own Voter to blacklist IP Addresses

The Symfony2 security component provides several layers to authenticate users. One of the layers is called a *voter*. A voter is a dedicated class that checks if the user has the rights to be connected to the application. For instance, Symfony2 provides a layer that checks if the user is fully authenticated or if it has some expected roles.

It is sometimes useful to create a custom voter to handle a specific case not handled by the framework. In this section, you'll learn how to create a voter that will allow you to blacklist users by their IP.

The Voter Interface

A custom voter must implement `Symfony\Component\Security\Core\Authorization\Voter\VoterInterface`, which requires the following three methods:

```
interface VoterInterface
{
    function supportsAttribute($attribute);
    function supportsClass($class);
    function vote(TokenInterface $token, $object, array $attributes);
}
```

The `supportsAttribute()` method is used to check if the voter supports the given user attribute (i.e: a role, an acl, etc.).

The `supportsClass()` method is used to check if the voter supports the current user token class.

The `vote()` method must implement the business logic that verifies whether or not the user is granted access. This method must return one of the following values:

- `VoterInterface::ACCESS_GRANTED`: The user is allowed to access the application
- `VoterInterface::ACCESS_ABSTAIN`: The voter cannot decide if the user is granted or not
- `VoterInterface::ACCESS_DENIED`: The user is not allowed to access the application

In this example, we will check if the user's IP address matches against a list of blacklisted addresses. If the user's IP is blacklisted, we will return `VoterInterface::ACCESS_DENIED`, otherwise we will return `VoterInterface::ACCESS_ABSTAIN` as this voter's purpose is only to deny access, not to grant access.

Creating a Custom Voter

To blacklist a user based on its IP, we can use the `request` service and compare the IP address against a set of blacklisted IP addresses:

```
namespace Acme\DemoBundle\Security\Authorization\Voter;

use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\Security\Core\Authorization\Voter\VoterInterface;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;

class ClientIpVoter implements VoterInterface
{
    public function __construct(ContainerInterface $container, array $blacklistedIp = array())
    {
        $this->container     = $container;
        $this->blacklistedIp = $blacklistedIp;
    }

    public function supportsAttribute($attribute)
    {
        // we won't check against a user attribute, so we return true
        return true;
    }

    public function supportsClass($class)
    {
        // our voter supports all type of token classes, so we return true
        return true;
    }

    function vote(TokenInterface $token, $object, array $attributes)
    {
        $request = $this->container->get('request');
        if (in_array($request->getClientIp(), $this->blacklistedIp)) {
            return VoterInterface::ACCESS_DENIED;
        }

        return VoterInterface::ACCESS_ABSTAIN;
    }
}
```

That's it! The voter is done. The next step is to inject the voter into the security layer. This can be done easily through the service container.

Declaring the Voter as a Service

To inject the voter into the security layer, we must declare it as a service, and tag it as a "security.voter":

- *YAML*

```
    # src/Acme/AcmeBundle/Resources/config/services.yml

    services:
        security.access.blacklist_voter:
            class:      Acme\DemoBundle\Security\Authorization\Voter\ClientIpVoter
            arguments:  [@service_container, [123.123.123.123, 171.171.171.171]]
            public:     false
```

```
            tags:
                -       { name: security.voter }
```

- *XML*

```
    <!-- src/Acme/AcmeBundle/Resources/config/services.xml -->

    <service id="security.access.blacklist_voter"
             class="Acme\DemoBundle\Security\Authorization\Voter\ClientIpVoter" public="false">
        <argument type="service" id="service_container" strict="false" />
        <argument type="collection">
            <argument>123.123.123.123</argument>
            <argument>171.171.171.171</argument>
        </argument>
        <tag name="security.voter" />
    </service>
```

- *PHP*

```
    // src/Acme/AcmeBundle/Resources/config/services.php

    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    $definition = new Definition(
        'Acme\DemoBundle\Security\Authorization\Voter\ClientIpVoter',
        array(
            new Reference('service_container'),
            array('123.123.123.123', '171.171.171.171'),
        ),
    );
    $definition->addTag('security.voter');
    $definition->setPublic(false);

    $container->setDefinition('security.access.blacklist_voter', $definition);
```

Tip: Be sure to import this configuration file from your main application configuration file (e.g. `app/config/config.yml`). For more information see *Importing Configuration with imports*. To read more about defining services in general, see the Service Container chapter.

Changing the Access Decision Strategy

In order for the new voter to take effect, we need to change the default access decision strategy, which, by default, grants access if *any* voter grants access.

In our case, we will choose the `unanimous` strategy. Unlike the `affirmative` strategy (the default), with the `unanimous` strategy, if only one voter denies access (e.g. the `ClientIpVoter`), access is not granted to the end user.

To do that, override the default `access_decision_manager` section of your application configuration file with the following code.

- *YAML*

```
    # app/config/security.yml
    security:
        access_decision_manager:
```

```
            # Strategy can be: affirmative, unanimous or consensus
            strategy: unanimous
```

That's it! Now, when deciding whether or not a user should have access, the new voter will deny access to any user in the list of blacklisted IPs.

3.1.45 Access Control Lists (ACLs)

In complex applications, you will often face the problem that access decisions cannot only be based on the person (`Token`) who is requesting access, but also involve a domain object that access is being requested for. This is where the ACL system comes in.

Imagine you are designing a blog system where your users can comment on your posts. Now, you want a user to be able to edit his own comments, but not those of other users; besides, you yourself want to be able to edit all comments. In this scenario, `Comment` would be our domain object that you want to restrict access to. You could take several approaches to accomplish this using Symfony2, two basic approaches are (non-exhaustive):

- *Enforce security in your business methods*: Basically, that means keeping a reference inside each `Comment` to all users who have access, and then compare these users to the provided `Token`.
- *Enforce security with roles*: In this approach, you would add a role for each `Comment` object, i.e. `ROLE_COMMENT_1`, `ROLE_COMMENT_2`, etc.

Both approaches are perfectly valid. However, they couple your authorization logic to your business code which makes it less reusable elsewhere, and also increases the difficulty of unit testing. Besides, you could run into performance issues if many users would have access to a single domain object.

Fortunately, there is a better way, which we will talk about now.

Bootstrapping

Now, before we finally can get into action, we need to do some bootstrapping. First, we need to configure the connection the ACL system is supposed to use:

- *YAML*

```
    # app/config/security.yml
    security:
        acl:
            connection: default
```

- *XML*

```
    <!-- app/config/security.xml -->
    <acl>
        <connection>default</connection>
    </acl>
```

- *PHP*

```
    // app/config/security.php
    $container->loadFromExtension('security', 'acl', array(
        'connection' => 'default',
    ));
```

Note: The ACL system requires at least one Doctrine DBAL connection to be configured. However, that does not mean that you have to use Doctrine for mapping your domain objects. You can use whatever mapper you like for your objects, be it Doctrine ORM, Mongo ODM, Propel, or raw SQL, the choice is yours.

After the connection is configured, we have to import the database structure. Fortunately, we have a task for this. Simply run the following command:

```
php app/console init:acl
```

Getting Started

Coming back to our small example from the beginning, let's implement ACL for it.

Creating an ACL, and adding an ACE

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Acl\Domain\ObjectIdentity;
use Symfony\Component\Security\Acl\Domain\UserSecurityIdentity;
use Symfony\Component\Security\Acl\Permission\MaskBuilder;
// ...

// BlogController.php
public function addCommentAction(Post $post)
{
    $comment = new Comment();

    // setup $form, and bind data
    // ...

    if ($form->isValid()) {
        $entityManager = $this->get('doctrine.orm.default_entity_manager');
        $entityManager->persist($comment);
        $entityManager->flush();

        // creating the ACL
        $aclProvider = $this->get('security.acl.provider');
        $objectIdentity = ObjectIdentity::fromDomainObject($comment);
        $acl = $aclProvider->createAcl($objectIdentity);

        // retrieving the security identity of the currently logged-in user
        $securityContext = $this->get('security.context');
        $user = $securityContext->getToken()->getUser();
        $securityIdentity = UserSecurityIdentity::fromAccount($user);

        // grant owner access
        $acl->insertObjectAce($securityIdentity, MaskBuilder::MASK_OWNER);
        $aclProvider->updateAcl($acl);
    }
}
```

There are a couple of important implementation decisions in this code snippet. For now, I only want to highlight two:

First, you may have noticed that `->createAcl()` does not accept domain objects directly, but only implementations of the `ObjectIdentityInterface`. This additional step of indirection allows you to work with ACLs even when you have no actual domain object instance at hand. This will be extremely helpful if you want to check permissions for a large number of objects without actually hydrating these objects.

The other interesting part is the `->insertObjectAce()` call. In our example, we are granting the user who is currently logged in owner access to the Comment. The `MaskBuilder::MASK_OWNER` is a pre-defined integer

bitmask; don't worry the mask builder will abstract away most of the technical details, but using this technique we can store many different permissions in one database row which gives us a considerable boost in performance.

Tip: The order in which ACEs are checked is significant. As a general rule, you should place more specific entries at the beginning.

Checking Access

```
// BlogController.php
public function editCommentAction(Comment $comment)
{
    $securityContext = $this->get('security.context');

    // check for edit access
    if (false === $securityContext->isGranted('EDIT', $comment))
    {
        throw new AccessDeniedException();
    }

    // retrieve actual comment object, and do your editing here
    // ...
}
```

In this example, we check whether the user has the `EDIT` permission. Internally, Symfony2 maps the permission to several integer bitmasks, and checks whether the user has any of them.

Note: You can define up to 32 base permissions (depending on your OS PHP might vary between 30 to 32). In addition, you can also define cumulative permissions.

Cumulative Permissions

In our first example above, we only granted the user the `OWNER` base permission. While this effectively also allows the user to perform any operation such as view, edit, etc. on the domain object, there are cases where we want to grant these permissions explicitly.

The `MaskBuilder` can be used for creating bit masks easily by combining several base permissions:

```
$builder = new MaskBuilder();
$builder
    ->add('view')
    ->add('edit')
    ->add('delete')
    ->add('undelete')
;
$mask = $builder->get(); // int(15)
```

This integer bitmask can then be used to grant a user the base permissions you added above:

```
$acl->insertObjectAce(new UserSecurityIdentity('johannes'), $mask);
```

The user is now allowed to view, edit, delete, and un-delete objects.

3.1.46 Advanced ACL Concepts

The aim of this chapter is to give a more in-depth view of the ACL system, and also explain some of the design decisions behind it.

Design Concepts

Symfony2's object instance security capabilities are based on the concept of an Access Control List. Every domain object **instance** has its own ACL. The ACL instance holds a detailed list of Access Control Entries (ACEs) which are used to make access decisions. Symfony2's ACL system focuses on two main objectives:

- providing a way to efficiently retrieve a large amount of ACLs/ACEs for your domain objects, and to modify them;
- providing a way to easily make decisions of whether a person is allowed to perform an action on a domain object or not.

As indicated by the first point, one of the main capabilities of Symfony2's ACL system is a high-performance way of retrieving ACLs/ACEs. This is extremely important since each ACL might have several ACEs, and inherit from another ACL in a tree-like fashion. Therefore, we specifically do not leverage any ORM, but the default implementation interacts with your connection directly using Doctrine's DBAL.

Object Identities

The ACL system is completely decoupled from your domain objects. They don't even have to be stored in the same database, or on the same server. In order to achieve this decoupling, in the ACL system your objects are represented through object identity objects. Everytime, you want to retrieve the ACL for a domain object, the ACL system will first create an object identity from your domain object, and then pass this object identity to the ACL provider for further processing.

Security Identities

This is analog to the object identity, but represents a user, or a role in your application. Each role, or user has its own security identity.

Database Table Structure

The default implementation uses five database tables as listed below. The tables are ordered from least rows to most rows in a typical application:

- *acl_security_identities*: This table records all security identities (SID) which hold ACEs. The default implementation ships with two security identities: `RoleSecurityIdentity`, and `UserSecurityIdentity`
- *acl_classes*: This table maps class names to a unique id which can be referenced from other tables.
- *acl_object_identities*: Each row in this table represents a single domain object instance.
- *acl_object_identity_ancestors*: This table allows us to determine all the ancestors of an ACL in a very efficient way.
- *acl_entries*: This table contains all ACEs. This is typically the table with the most rows. It can contain tens of millions without significantly impacting performance.

Scope of Access Control Entries

Access control entries can have different scopes in which they apply. In Symfony2, we have basically two different scopes:

- Class-Scope: These entries apply to all objects with the same class.
- Object-Scope: This was the scope we solely used in the previous chapter, and it only applies to one specific object.

Sometimes, you will find the need to apply an ACE only to a specific field of the object. Let's say you want the ID only to be viewable by an administrator, but not by your customer service. To solve this common problem, we have added two more sub-scopes:

- Class-Field-Scope: These entries apply to all objects with the same class, but only to a specific field of the objects.
- Object-Field-Scope: These entries apply to a specific object, and only to a specific field of that object.

Pre-Authorization Decisions

For pre-authorization decisions, that is decisions before any method, or secure action is invoked, we rely on the proven AccessDecisionManager service that is also used for reaching authorization decisions based on roles. Just like roles, the ACL system adds several new attributes which may be used to check for different permissions.

Built-in Permission Map

Attribute	Intended Meaning	Integer Bitmasks
VIEW	Whether someone is allowed to view the domain object.	VIEW, EDIT, OPERATOR, MASTER, or OWNER
EDIT	Whether someone is allowed to make changes to the domain object.	EDIT, OPERATOR, MASTER, or OWNER
CREATE	Whether someone is allowed to create the domain object.	CREATE, OPERATOR, MASTER, or OWNER
DELETE	Whether someone is allowed to delete the domain object.	DELETE, OPERATOR, MASTER, or OWNER
UNDELETE	Whether someone is allowed to restore a previously deleted domain object.	UNDELETE, OPERATOR, MASTER, or OWNER
OPERATOR	Whether someone is allowed to perform all of the above actions.	OPERATOR, MASTER, or OWNER
MASTER	Whether someone is allowed to perform all of the above actions, and in addition is allowed to grant any of the above permissions to others.	MASTER, or OWNER
OWNER	Whether someone owns the domain object. An owner can perform any of the above actions *and* grant master and owner permissions.	OWNER

Permission Attributes vs. Permission Bitmasks

Attributes are used by the AccessDecisionManager, just like roles are attributes used by the AccessDecisionManager. Often, these attributes represent in fact an aggregate of integer bitmasks. Integer bitmasks on the other hand, are used

by the ACL system internally to efficiently store your users' permissions in the database, and perform access checks using extremely fast bitmask operations.

Extensibility

The above permission map is by no means static, and theoretically could be completely replaced at will. However, it should cover most problems you encounter, and for interoperability with other bundles, we encourage you to stick to the meaning we have envisaged for them.

Post Authorization Decisions

Post authorization decisions are made after a secure method has been invoked, and typically involve the domain object which is returned by such a method. After invocation providers also allow to modify, or filter the domain object before it is returned.

Due to current limitations of the PHP language, there are no post-authorization capabilities build into the core Security component. However, there is an experimental JMSSecurityExtraBundle which adds these capabilities. See its documentation for further information on how this is accomplished.

Process for Reaching Authorization Decisions

The ACL class provides two methods for determining whether a security identity has the required bitmasks, `isGranted` and `isFieldGranted`. When the ACL receives an authorization request through one of these methods, it delegates this request to an implementation of PermissionGrantingStrategy. This allows you to replace the way access decisions are reached without actually modifying the ACL class itself.

The PermissionGrantingStrategy first checks all your object-scope ACEs if none is applicable, the class-scope ACEs will be checked, if none is applicable, then the process will be repeated with the ACEs of the parent ACL. If no parent ACL exists, an exception will be thrown.

3.1.47 How to force HTTPS or HTTP for Different URLs

You can force areas of your site to use the `HTTPS` protocol in the security config. This is done through the `access_control` rules using the `requires_channel` option. For example, if you want to force all URLs starting with `/secure` to use `HTTPS` then you could use the following config:

- *YAML*

```
access_control:
    - path: ^/secure
      roles: ROLE_ADMIN
      requires_channel: https
```

- *XML*

```
<access-control>
    <rule path="^/secure" role="ROLE_ADMIN" requires_channel="https" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/secure',
          'role' => 'ROLE_ADMIN',
```

```
            'requires_channel' => 'https'
        ),
    ),
```

The login form itself needs to allow anonymous access otherwise users will be unable to authenticate. To force it to use `HTTPS` you can still use `access_control` rules by using the `IS_AUTHENTICATED_ANONYMOUSLY` role:

- *YAML*

```
access_control:
    - path: ^/login
      roles: IS_AUTHENTICATED_ANONYMOUSLY
      requires_channel: https
```

- *XML*

```
<access-control>
    <rule path="^/login"
          role="IS_AUTHENTICATED_ANONYMOUSLY"
          requires_channel="https" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/login',
          'role' => 'IS_AUTHENTICATED_ANONYMOUSLY',
          'requires_channel' => 'https'
    ),
),
```

It is also possible to specify using `HTTPS` in the routing configuration see How to force routes to always use HTTPS or HTTP for more details.

3.1.48 How to customize your Form Login

Using a *form login* for authentication is a common, and flexible, method for handling authentication in Symfony2. Pretty much every aspect of the form login can be customized. The full, default configuration is shown in the next section.

Form Login Configuration Reference

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # the user is redirected here when he/she needs to login
                login_path:                     /login

                # if true, forward the user to the login form instead of redirecting
                use_forward:                    false

                # submit the login form here
                check_path:                     /login_check
```

```
                    # by default, the login form *must* be a POST, not a GET
                    post_only:                      true

                    # login success redirecting options (read further below)
                    always_use_default_target_path: false
                    default_target_path:            /
                    target_path_parameter:          _target_path
                    use_referer:                    false

                    # login failure redirecting options (read further below)
                    failure_path:                   null
                    failure_forward:                false

                    # field names for the username and password fields
                    username_parameter:             _username
                    password_parameter:             _password

                    # csrf token options
                    csrf_parameter:                 _csrf_token
                    intention:                      authenticate
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            check_path="/login_check"
            login_path="/login"
            use_forward="false"
            always_use_default_target_path="false"
            default_target_path="/"
            target_path_parameter="_target_path"
            use_referer="false"
            failure_path="null"
            failure_forward="false"
            username_parameter="_username"
            password_parameter="_password"
            csrf_parameter="_csrf_token"
            intention="authenticate"
            post_only="true"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            'check_path'                    => '/login_check',
            'login_path'                    => '/login',
            'user_forward'                  => false,
            'always_use_default_target_path' => false,
            'default_target_path'           => '/',
            'target_path_parameter'         => _target_path,
            'use_referer'                   => false,
```

```
                'failure_path'                  => null,
                'failure_forward'               => false,
                'username_parameter'            => '_username',
                'password_parameter'            => '_password',
                'csrf_parameter'                => '_csrf_token',
                'intention'                     => 'authenticate',
                'post_only'                     => true,
            )),
        ),
    ));
```

Redirecting after Success

You can change where the login form redirects after a successful login using the various config options. By default the form will redirect to the URL the user requested (i.e. the URL which triggered the login form being shown). For example, if the user requested `http://www.example.com/admin/post/18/edit` then after he/she will eventually be sent back to `http://www.example.com/admin/post/18/edit` after successfully logging in. This is done by storing the requested URL in the session. If no URL is present in the session (perhaps the user went directly to the login page), then the user is redirected to the default page, which is `/` (i.e. the homepage) by default. You can change this behavior in several ways.

Note: As mentioned, by default the user is redirected back to the page he originally requested. Sometimes, this can cause problems, like if a background AJAX request "appears" to be the last visited URL, causing the user to be redirected there. For information on controlling this behavior, see How to change the Default Target Path Behavior.

Changing the Default Page

First, the default page can be set (i.e. the page the user is redirected to if no previous page was stored in the session). To set it to `/admin` use the following config:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                default_target_path: /admin
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            default_target_path="/admin"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
```

```
        'firewalls' => array(
            'main' => array('form_login' => array(
                // ...
                'default_target_path' => '/admin',
            )),
        ),
    ));
```

Now, when no URL is set in the session users will be sent to `/admin`.

Always Redirect to the Default Page

You can make it so that users are always redirected to the default page regardless of what URL they had requested previously by setting the `always_use_default_target_path` option to true:

- *YAML*

```
    # app/config/security.yml
    security:
        firewalls:
            main:
                form_login:
                    # ...
                    always_use_default_target_path: true
```

- *XML*

```
    <!-- app/config/security.xml -->
    <config>
        <firewall>
            <form-login
                always_use_default_target_path="true"
            />
        </firewall>
    </config>
```

- *PHP*

```
    // app/config/security.php
    $container->loadFromExtension('security', array(
        'firewalls' => array(
            'main' => array('form_login' => array(
                // ...
                'always_use_default_target_path' => true,
            )),
        ),
    ));
```

Using the Referring URL

In case no previous URL was stored in the session, you may wish to try using the `HTTP_REFERER` instead, as this will often be the same. You can do this by setting `use_referer` to true (it defaults to false):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                use_referer:        true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            use_referer="true"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            // ...
            'use_referer' => true,
        )),
    ),
));
```

New in version 2.1: As of 2.1, if the referer is equal to the `login_path` option, the user will be redirected to the `default_target_path`.

Control the Redirect URL from inside the Form

You can also override where the user is redirected to via the form itself by including a hidden field with the name `_target_path`. For example, to redirect to the URL defined by some `acount` route, use the following:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="hidden" name="_target_path" value="account" />

    <input type="submit" name="login" />
</form>
```

- *PHP*

```
<?php // src/Acme/SecurityBundle/Resources/views/Security/login.html.php ?>
<?php if ($error): ?>
    <div><?php echo $error->getMessage() ?></div>
<?php endif; ?>

<form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="<?php echo $last_username ?>" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="hidden" name="_target_path" value="account" />

    <input type="submit" name="login" />
</form>
```

Now, the user will be redirected to the value of the hidden form field. The value attribute can be a relative path, absolute URL, or a route name. You can even change the name of the hidden form field by changing the `target_path_parameter` option to another value.

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                target_path_parameter: redirect_url
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            target_path_parameter="redirect_url"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            'target_path_parameter' => redirect_url,
        )),
    ),
));
```

Redirecting on Login Failure

In addition to redirect the user after a successful login, you can also set the URL that the user should be redirected to after a failed login (e.g. an invalid username or password was submitted). By default, the user is redirected back to the login form itself. You can set this to a different URL with the following config:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                failure_path: /login_failure
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            failure_path="login_failure"
        />
    </firewall>
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            // ...
            'failure_path' => login_failure,
        )),
    ),
));
```

3.1.49 How to secure any Service or Method in your Application

In the security chapter, you can see how to *secure a controller* by requesting the `security.context` service from the Service Container and checking the current user's role:

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function helloAction($name)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }

    // ...
}
```

You can also secure *any* service in a similar way by injecting the `security.context` service into it. For a general introduction to injecting dependencies into services see the Service Container chapter of the book. For example, suppose you have a `NewsletterManager` class that sends out emails and you want to restrict its use to only users who have some `ROLE_NEWSLETTER_ADMIN` role. Before you add security, the class looks something like this:

```
namespace Acme\HelloBundle\Newsletter;

class NewsletterManager
```

```
{

    public function sendNewsletter()
    {
        // where you actually do the work
    }

    // ...
}
```

Your goal is to check the user's role when the `sendNewsletter()` method is called. The first step towards this is to inject the `security.context` service into the object. Since it won't make sense *not* to perform the security check, this is an ideal candidate for constructor injection, which guarantees that the security context object will be available inside the `NewsletterManager` class:

```
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\Security\Core\SecurityContextInterface;

class NewsletterManager
{
    protected $securityContext;

    public function __construct(SecurityContextInterface $securityContext)
    {
        $this->securityContext = $securityContext;
    }

    // ...
}
```

Then in your service configuration, you can inject the service:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

services:
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@security.context]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager</par
</parameters>

<services>
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <argument type="service" id="security.context"/>
    </service>
</services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    $container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana

    $container->setDefinition('newsletter_manager', new Definition(
        '%newsletter_manager.class%',
        array(new Reference('security.context'))
    ));
```

The injected service can then be used to perform the security check when the `sendNewsletter()` method is called:

```
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Core\SecurityContextInterface;
// ...

class NewsletterManager
{
    protected $securityContext;

    public function __construct(SecurityContextInterface $securityContext)
    {
        $this->securityContext = $securityContext;
    }

    public function sendNewsletter()
    {
        if (false === $this->securityContext->isGranted('ROLE_NEWSLETTER_ADMIN')) {
            throw new AccessDeniedException();
        }

        //--
    }

    // ...
}
```

If the current user does not have the `ROLE_NEWSLETTER_ADMIN`, they will be prompted to log in.

Securing Methods Using Annotations

You can also secure method calls in any service with annotations by using the optional JMSSecurityExtraBundle bundle. This bundle is included in the Symfony2 Standard Distribution.

To enable the annotations functionality, *tag* the service you want to secure with the `security.secure_service` tag (you can also automatically enable this functionality for all services, see the *sidebar* below):

- *YAML*

```
    # src/Acme/HelloBundle/Resources/config/services.yml
    # ...

    services:
        newsletter_manager:
            # ...
```

```
            tags:
                -  { name: security.secure_service }
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/services.xml -->
    <!-- ... -->

    <services>
        <service id="newsletter_manager" class="%newsletter_manager.class%">
            <!-- ... -->
            <tag name="security.secure_service" />
        </service>
    </services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    $definition = new Definition(
        '%newsletter_manager.class%',
        array(new Reference('security.context'))
    ));
    $definition->addTag('security.secure_service');
    $container->setDefinition('newsletter_manager', $definition);
```

You can then achieve the same results as above using an annotation:

```
namespace Acme\HelloBundle\Newsletter;

use JMS\SecurityExtraBundle\Annotation\Secure;
// ...

class NewsletterManager
{

    /**
     * @Secure(roles="ROLE_NEWSLETTER_ADMIN")
     */
    public function sendNewsletter()
    {
        //--
    }

    // ...
}
```

Note: The annotations work because a proxy class is created for your class which performs the security checks. This means that, whilst you can use annotations on public and protected methods, you cannot use them with private methods or methods marked final.

The `JMSSecurityExtraBundle` also allows you to secure the parameters and return values of methods. For more information, see the JMSSecurityExtraBundle documentation.

Activating the Annotations Functionality for all Services

When securing the method of a service (as shown above), you can either tag each service individually, or activate the functionality for *all* services at once. To do so, set the `secure_all_services` configuration option to true:

- *YAML*

```yaml
# app/config/config.yml
jms_security_extra:
    # ...
    secure_all_services: true
```

- *XML*

```xml
<!-- app/config/config.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/se

    <jms_security_extra secure_controllers="true" secure_all_services="true" />

</srv:container>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('jms_security_extra', array(
    // ...
    'secure_all_services' => true,
));
```

The disadvantage of this method is that, if activated, the initial page load may be very slow depending on how many services you have defined.

3.1.50 How to load Security Users from the Database (the Entity Provider)

The security layer is one of the smartest tools of Symfony. It handles two things: the authentication and the authorization processes. Although it may seem difficult to understand how it works internally, the security system is very flexible and allows you to integrate your application with any authentication backend, like Active Directory, an OAuth server or a database.

Introduction

This article focuses on how to authenticate users against a database table managed by a Doctrine entity class. The content of this cookbook entry is split in three parts. The first part is about designing a Doctrine `User` entity class and making it usable in the security layer of Symfony. The second part describes how to easily authenticate a user with the Doctrine `Symfony\Bridge\Doctrine\Security\User\EntityUserProvider` object bundled with the framework and some configuration. Finally, the tutorial will demonstrate how to create a custom `Symfony\Bridge\Doctrine\Security\User\EntityUserProvider` object to retrieve users from a database with custom conditions.

This tutorial assumes there is a bootstrapped and loaded `Acme\UserBundle` bundle in the application kernel.

The Data Model

For the purpose of this cookbook, the `AcmeUserBundle` bundle contains a `User` entity class with the following fields: `id`, `username`, `salt`, `password`, `email` and `isActive`. The `isActive` field tells whether or not the user account is active.

To make it shorter, the getter and setter methods for each have been removed to focus on the most important methods that come from the `Symfony\Component\Security\Core\User\UserInterface`.

New in version 2.1: In Symfony 2.1, the `equals` method was removed from `UserInterface`. If you need to override the default implementation of comparison logic, implement the new `Symfony\Component\Security\Core\User\EquatableInterface` interface.

```
// src/Acme/UserBundle/Entity/User.php

namespace Acme\UserBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;

/**
 * Acme\UserBundle\Entity\User
 *
 * @ORM\Table(name="acme_users")
 * @ORM\Entity(repositoryClass="Acme\UserBundle\Entity\UserRepository")
 */
class User implements UserInterface
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id()
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(name="username", type="string", length=25, unique=true)
     */
    private $username;

    /**
     * @ORM\Column(name="salt", type="string", length=40)
     */
    private $salt;

    /**
     * @ORM\Column(name="password", type="string", length=40)
     */
    private $password;

    /**
     * @ORM\Column(name="email", type="string", length=60, unique=true)
     */
    private $email;

    /**
     * @ORM\Column(name="is_active", type="boolean")
     */
    private $isActive;
```

```
    public function __construct()
    {
        $this->isActive = true;
        $this->salt = base_convert(sha1(uniqid(mt_rand(), true)), 16, 36);
    }

    public function getRoles()
    {
        return array('ROLE_USER');
    }

    public function eraseCredentials()
    {
    }

    public function getUsername()
    {
        return $this->username;
    }

    public function getSalt()
    {
        return $this->salt;
    }

    public function getPassword()
    {
        return $this->password;
    }
}
```

In order to use an instance of the `AcmeUserBundle:User` class in the Symfony security layer, the entity class must implement the `Symfony\Component\Security\Core\User\UserInterface`. This interface forces the class to implement the five following methods: `getRoles()`, `getPassword()`, `getSalt()`, `getUsername()`, `eraseCredentials()`. For more details on each of these, see `Symfony\Component\Security\Core\User\UserInterface`.

Below is an export of my `User` table from MySQL. For details on how to create user records and encode their password, see *Encoding the User's Password*.

```
mysql> select * from user;
+----+----------+------------------------------------------+------------------------------------
| id | username | salt                                     | password
+----+----------+------------------------------------------+------------------------------------
|  1 | hhamon   | 7308e59b97f6957fb42d66f894793079c366d7c2 | 09610f61637408828a35d7debee5b38a8350eebe
|  2 | jsmith   | ce617a6cca9126bf4036ca0c02e82deea081e564 | 8390105917f3a3d533815250ed7c64b4594d7eb
|  3 | maxime   | cd01749bb995dc658fa56ed45458d807b523e4cf | 9764731e5f7fb944de5fd8efad4949b995b72a3
|  4 | donald   | 6683c2bfd90c0426088402930cadd0f84901f2f4 | 5c3bcec385f59edcc04490d1db95fdb8673bf61
+----+----------+------------------------------------------+------------------------------------
4 rows in set (0.00 sec)
```

The database now contains four users with different usernames, emails and statuses. The next part will focus on how to authenticate one of these users thanks to the Doctrine entity user provider and a couple of lines of configuration.

Authenticating Someone against a Database

Authenticating a Doctrine user against the database with the Symfony security layer is a piece of cake. Everything resides in the configuration of the SecurityBundle stored in the `app/config/security.yml` file.

Below is an example of configuration where the user will enter his/her username and password via HTTP basic authentication. That information will then be checked against our User entity records in the database:

- *YAML*

```yaml
# app/config/security.yml
security:
    encoders:
        Acme\UserBundle\Entity\User:
            algorithm: sha1
            encode_as_base64: false
            iterations: 1

    providers:
        administrators:
            entity: { class: AcmeUserBundle:User, property: username }

    firewalls:
        admin_area:
            pattern:    ^/admin
            http_basic: ~

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
```

The `encoders` section associates the `sha1` password encoder to the entity class. This means that Symfony will expect the password that's encoded in the database to be encoded using this algorithm. For details on how to create a new User object with a properly encoded password, see the *Encoding the User's Password* section of the security chapter.

The `providers` section defines an `administrators` user provider. A user provider is a "source" of where users are loaded during authentication. In this case, the `entity` keyword means that Symfony will use the Doctrine entity user provider to load User entity objects from the database by using the `username` unique field. In other words, this tells Symfony how to fetch the user from the database before checking the password validity.

This code and configuration works but it's not enough to secure the application for **active** users. As of now, we still can authenticate with `maxime`. The next section explains how to forbid non active users.

Forbid non Active Users

The easiest way to exclude non active users is to implement the `Symfony\Component\Security\Core\User\AdvancedUser` interface that takes care of checking the user's account status. The `Symfony\Component\Security\Core\User\AdvancedUserInterface` extends the `Symfony\Component\Security\Core\User\UserInterface` interface, so you just need to switch to the new interface in the `AcmeUserBundle:User` entity class to benefit from simple and advanced authentication behaviors.

The `Symfony\Component\Security\Core\User\AdvancedUserInterface` interface adds four extra methods to validate the account status:

- `isAccountNonExpired()` checks whether the user's account has expired,
- `isAccountNonLocked()` checks whether the user is locked,

- `isCredentialsNonExpired()` checks whether the user's credentials (password) has expired,
- `isEnabled()` checks whether the user is enabled.

For this example, the first three methods will return `true` whereas the `isEnabled()` method will return the boolean value in the `isActive` field.

```
// src/Acme/UserBundle/Entity/User.php

namespace Acme\Bundle\UserBundle\Entity;

// ...
use Symfony\Component\Security\Core\User\AdvancedUserInterface;

// ...
class User implements AdvancedUserInterface
{
    // ...
    public function isAccountNonExpired()
    {
        return true;
    }

    public function isAccountNonLocked()
    {
        return true;
    }

    public function isCredentialsNonExpired()
    {
        return true;
    }

    public function isEnabled()
    {
        return $this->isActive;
    }
}
```

If we try to authenticate a `maxime`, the access is now forbidden as this user does not have an enabled account. The next session will focus on how to write a custom entity provider to authenticate a user with his username or his email address.

Authenticating Someone with a Custom Entity Provider

The next step is to allow a user to authenticate with his username or his email address as they are both unique in the database. Unfortunately, the native entity provider is only able to handle a single property to fetch the user from the database.

To accomplish this, create a custom entity provider that looks for a user whose username *or* email field matches the submitted login username. The good news is that a Doctrine repository object can act as an entity user provider if it implements the `Symfony\Component\Security\Core\User\UserProviderInterface`. This interface comes with three methods to implement: `loadUserByUsername($username)`, `refreshUser(UserInterface $user)`, and `supportsClass($class)`. For more details, see `Symfony\Component\Security\Core\User\UserProviderInterface`.

The code below shows the implementation of the `Symfony\Component\Security\Core\User\UserProviderInterface` in the `UserRepository` class:

```
// src/Acme/UserBundle/Entity/UserRepository.php

namespace Acme\UserBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
use Symfony\Component\Security\Core\Exception\UnsupportedUserException;
use Doctrine\ORM\EntityRepository;
use Doctrine\ORM\NoResultException;

class UserRepository extends EntityRepository implements UserProviderInterface
{
    public function loadUserByUsername($username)
    {
        $q = $this
            ->createQueryBuilder('u')
            ->where('u.username = :username OR u.email = :email')
            ->setParameter('username', $username)
            ->setParameter('email', $username)
            ->getQuery()
        ;

        try {
            // The Query::getSingleResult() method throws an exception
            // if there is no record matching the criteria.
            $user = $q->getSingleResult();
        } catch (NoResultException $e) {
            throw new UsernameNotFoundException(sprintf('Unable to find an active admin AcmeUserBundl
        }

        return $user;
    }

    public function refreshUser(UserInterface $user)
    {
        $class = get_class($user);
        if (!$this->supportsClass($class)) {
            throw new UnsupportedUserException(sprintf('Instances of "%s" are not supported.', $class
        }

        return $this->loadUserByUsername($user->getUsername());
    }

    public function supportsClass($class)
    {
        return $this->getEntityName() === $class || is_subclass_of($class, $this->getEntityName());
    }
}
```

To finish the implementation, the configuration of the security layer must be changed to tell Symfony to use the new custom entity provider instead of the generic Doctrine entity provider. It's trival to achieve by removing the `property` field in the `security.providers.administrators.entity` section of the `security.yml` file.

- *YAML*

```
    # app/config/security.yml
    security:
        # ...
        providers:
            administrators:
                entity: { class: AcmeUserBundle:User }
        # ...
```

By doing this, the security layer will use an instance of `UserRepository` and call its `loadUserByUsername()` method to fetch a user from the database whether he filled in his username or email address.

Managing Roles in the Database

The end of this tutorial focuses on how to store and retrieve a list of roles from the database. As mentioned previously, when your user is loaded, its `getRoles()` method returns the array of security roles that should be assigned to the user. You can load this data from anywhere - a hardcoded list used for all users (e.g. `array('ROLE_USER')`), a Doctrine array property called `roles`, or via a Doctrine relationship, as we'll learn about in this section.

Caution: In a typical setup, you should always return at least 1 role from the `getRoles()` method. By convention, a role called `ROLE_USER` is usually returned. If you fail to return any roles, it may appear as if your user isn't authenticated at all.

In this example, the `AcmeUserBundle:User` entity class defines a many-to-many relationship with a `AcmeUserBundle:Group` entity class. A user can be related several groups and a group can be composed of one or more users. As a group is also a role, the previous `getRoles()` method now returns the list of related groups:

```
// src/Acme/UserBundle/Entity/User.php

namespace Acme\Bundle\UserBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

// ...
class User implements AdvancedUserInterface
{
    /**
     * @ORM\ManyToMany(targetEntity="Group", inversedBy="users")
     *
     */
    private $groups;

    public function __construct()
    {
        $this->groups = new ArrayCollection();
    }

    // ...

    public function getRoles()
    {
        return $this->groups->toArray();
    }
}
```

The `AcmeUserBundle:Group` entity class defines three table fields (`id`, `name` and `role`). The unique `role` field contains the role name used by the Symfony security layer to secure parts of the applica-

tion. The most important thing to notice is that the `AcmeUserBundle:Group` entity class implements the `Symfony\Component\Security\Core\Role\RoleInterface` that forces it to have a `getRole()` method:

```
namespace Acme\Bundle\UserBundle\Entity;

use Symfony\Component\Security\Core\Role\RoleInterface;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table(name="acme_groups")
 * @ORM\Entity()
 */
class Group implements RoleInterface
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id()
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /** @ORM\Column(name="name", type="string", length=30) */
    private $name;

    /** @ORM\Column(name="role", type="string", length=20, unique=true) */
    private $role;

    /** @ORM\ManyToMany(targetEntity="User", mappedBy="groups") */
    private $users;

    public function __construct()
    {
        $this->users = new ArrayCollection();
    }

    // ... getters and setters for each property

    /** @see RoleInterface */
    public function getRole()
    {
        return $this->role;
    }
}
```

To improve performances and avoid lazy loading of groups when retrieving a user from the custom entity provider, the best solution is to join the groups relationship in the `UserRepository::loadUserByUsername()` method. This will fetch the user and his associated roles / groups with a single query:

```
// src/Acme/UserBundle/Entity/UserRepository.php

namespace Acme\Bundle\UserBundle\Entity;

// ...

class UserRepository extends EntityRepository implements UserProviderInterface
{
    public function loadUserByUsername($username)
```

```
    {
        $q = $this
            ->createQueryBuilder('u')
            ->select('u, g')
            ->leftJoin('u.groups', 'g')
            ->where('u.username = :username OR u.email = :email')
            ->setParameter('username', $username)
            ->setParameter('email', $username)
            ->getQuery()
        ;

        // ...
    }

    // ...
}
```

The `QueryBuilder::leftJoin()` method joins and fetches related groups from the `AcmeUserBundle:User` model class when a user is retrieved with his email address or username.

3.1.51 How to create a custom User Provider

Part of Symfony's standard authentication process depends on "user providers". When a user submits a username and password, the authentication layer asks the configured user provider to return a user object for a given username. Symfony then checks whether the password of this user is correct and generates a security token so the user stays authenticated during the current session. Out of the box, Symfony has an "in_memory" and an "entity" user provider. In this entry we'll see how you can create your own user provider, which could be useful if your users are accessed via a custom database, a file, or - as we show in this example - a web service.

Create a User Class

First, regardless of *where* your user data is coming from, you'll need to create a `User` class that represents that data. The `User` can look however you want and contain any data. The only requirement is that the class implements `Symfony\Component\Security\Core\User\UserInterface`. The methods in this interface should therefore be defined in the custom user class: `getRoles()`, `getPassword()`, `getSalt()`, `getUsername()`, `eraseCredentials()`, `equals()`.

Let's see this in action:

```
// src/Acme/WebserviceUserBundle/Security/User.php
namespace Acme\WebserviceUserBundle\Security\User;

use Symfony\Component\Security\Core\User\UserInterface;

class WebserviceUser implements UserInterface
{
    private $username;
    private $password;
    private $salt;
    private $roles;

    public function __construct($username, $password, $salt, array $roles)
    {
        $this->username = $username;
        $this->password = $password;
        $this->salt = $salt;
```

```
        $this->roles = $roles;
    }

    public function getRoles()
    {
        return $this->roles;
    }

    public function getPassword()
    {
        return $this->password;
    }

    public function getSalt()
    {
        return $this->salt;
    }

    public function getUsername()
    {
        return $this->username;
    }

    public function eraseCredentials()
    {
    }

    public function equals(UserInterface $user)
    {
        if (!$user instanceof WebserviceUser) {
            return false;
        }

        if ($this->password !== $user->getPassword()) {
            return false;
        }

        if ($this->getSalt() !== $user->getSalt()) {
            return false;
        }

        if ($this->username !== $user->getUsername()) {
            return false;
        }

        return true;
    }
}
```

If you have more information about your users - like a "first name" - then you can add a `firstName` field to hold that data.

For more details on each of the methods, see `Symfony\Component\Security\Core\User\UserInterface`.

Create a User Provider

Now that we have a `User` class, we'll create a user provider, which will grab user information from some web service, create a `WebserviceUser` object, and populate it with data.

The user provider is just a plain PHP class that has to implement the `Symfony\Component\Security\Core\User\UserProviderInterface`, which requires three methods to be defined: `loadUserByUsername($username)`, `refreshUser(UserInterface $user)`, and `supportsClass($class)`. For more details, see `Symfony\Component\Security\Core\User\UserProviderInterface`.

Here's an example of how this might look:

```
// src/Acme/WebserviceUserBundle/Security/User/WebserviceUserProvider.php
namespace Acme\WebserviceUserBundle\Security\User;

use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
use Symfony\Component\Security\Core\Exception\UnsupportedUserException;

class WebserviceUserProvider implements UserProviderInterface
{
    public function loadUserByUsername($username)
    {
        // make a call to your webservice here
        // $userData = ...
        // pretend it returns an array on success, false if there is no user

        if ($userData) {
            // $password = '...';
            // ...

            return new WebserviceUser($username, $password, $salt, $roles)
        } else {
            throw new UsernameNotFoundException(sprintf('Username "%s" does not exist.', $username));
        }
    }

    public function refreshUser(UserInterface $user)
    {
        if (!$user instanceof WebserviceUser) {
            throw new UnsupportedUserException(sprintf('Instances of "%s" are not supported.', get_cl
        }

        return $this->loadUserByUsername($user->getUsername());
    }

    public function supportsClass($class)
    {
        return $class === 'Acme\WebserviceUserBundle\Security\User\WebserviceUser';
    }
}
```

Create a Service for the User Provider

Now we make the user provider available as service.

- *YAML*

```
# src/Acme/MailerBundle/Resources/config/services.yml
parameters:
    webservice_user_provider.class: Acme\WebserviceUserBundle\Security\User\WebserviceUserProvid

services:
    webservice_user_provider:
        class: %webservice_user_provider.class%
```

- *XML*

```
<!-- src/Acme/WebserviceUserBundle/Resources/config/services.xml -->
<parameters>
    <parameter key="webservice_user_provider.class">Acme\WebserviceUserBundle\Security\User\Webs
</parameters>

<services>
    <service id="webservice_user_provider" class="%webservice_user_provider.class%"></service>
</services>
```

- *PHP*

```
// src/Acme/WebserviceUserBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('webservice_user_provider.class', 'Acme\WebserviceUserBundle\Security\U

$container->setDefinition('webservice_user_provider', new Definition('%webservice_user_provider.
```

Tip: The real implementation of the user provider will probably have some dependencies or configuration options or other services. Add these as arguments in the service definition.

Note: Make sure the services file is being imported. See *Importing Configuration with imports* for details.

Modify `security.yml`

In `/app/config/security.yml` everything comes together. Add the user provider to the list of providers in the "security" section. Choose a name for the user provider (e.g. "webservice") and mention the id of the service you just defined.

```
security:
    providers:
        webservice:
            id: webservice_user_provider
```

Symfony also needs to know how to encode passwords that are supplied by website users, e.g. by filling in a login form. You can do this by adding a line to the "encoders" section in `/app/config/security.yml`.

```
security:
    encoders:
        Acme\WebserviceUserBundle\Security\User\WebserviceUser: sha512
```

The value here should correspond with however the passwords were originally encoded when creating your users (however those users were created). When a user submits her password, the password is appended to the salt value and then encoded using this algorithm before being compared to the hashed password returned by your `getPassword()` method. Additionally, depending on your options, the password may be encoded multiple times and encoded to base64.

Specifics on how passwords are encoded

Symfony uses a specific method to combine the salt and encode the password before comparing it to your encoded password. If `getSalt()` returns nothing, then the submitted password is simply encoded using the algorithm you specify in `security.yml`. If a salt *is* specified, then the following value is created and *then* hashed via the algorithm:

```
$password.'{'.$salt.'}';
```

If your external users have their passwords salted via a different method, then you'll need to do a bit more work so that Symfony properly encodes the password. That is beyond the scope of this entry, but would include sub-classing `MessageDigestPasswordEncoder` and overriding the `mergePasswordAndSalt` method. Additionally, the hash, by default, is encoded multiple times and encoded to base64. For specific details, see MessageDigestPasswordEncoder. To prevent this, configure it in `security.yml`:

```
security:
    encoders:
        Acme\WebserviceUserBundle\Security\User\WebserviceUser:
            algorithm: sha512
            encode_as_base64: false
            iterations: 1
```

3.1.52 How to create a custom Authentication Provider

If you have read the chapter on Security, you understand the distinction Symfony2 makes between authentication and authorization in the implementation of security. This chapter discusses the core classes involved in the authentication process, and how to implement a custom authentication provider. Because authentication and authorization are separate concepts, this extension will be user-provider agnostic, and will function with your application's user providers, may they be based in memory, a database, or wherever else you choose to store them.

Meet WSSE

The following chapter demonstrates how to create a custom authentication provider for WSSE authentication. The security protocol for WSSE provides several security benefits:

1. Username / Password encryption
2. Safe guarding against replay attacks
3. No web server configuration required

WSSE is very useful for the securing of web services, may they be SOAP or REST.

There is plenty of great documentation on WSSE, but this article will focus not on the security protocol, but rather the manner in which a custom protocol can be added to your Symfony2 application. The basis of WSSE is that a request header is checked for encrypted credentials, verified using a timestamp and nonce, and authenticated for the requested user using a password digest.

Note: WSSE also supports application key validation, which is useful for web services, but is outside the scope of this chapter.

The Token

The role of the token in the Symfony2 security context is an important one. A token represents the user authentication data present in the request. Once a request is authenticated, the token retains the user's data, and delivers this data

across the security context. First, we will create our token class. This will allow the passing of all relevant information to our authentication provider.

```
// src/Acme/DemoBundle/Security/Authentication/Token/WsseUserToken.php
namespace Acme\DemoBundle\Security\Authentication\Token;

use Symfony\Component\Security\Core\Authentication\Token\AbstractToken;

class WsseUserToken extends AbstractToken
{
    public $created;
    public $digest;
    public $nonce;

    public function getCredentials()
    {
        return '';
    }
}
```

Note: The `WsseUserToken` class extends the security component's `Symfony\Component\Security\Core\Authentication\Token\AbstractToken` class, which provides basic token functionality. Implement the `Symfony\Component\Security\Core\Authentication\Token\TokenIn` on any class to use as a token.

The Listener

Next, you need a listener to listen on the security context. The listener is responsible for fielding requests to the firewall and calling the authentication provider. A listener must be an instance of `Symfony\Component\Security\Http\Firewall\ListenerInterface`. A security listener should handle the `Symfony\Component\HttpKernel\Event\GetResponseEvent` event, and set an authenticated token in the security context if successful.

```
// src/Acme/DemoBundle/Security/Firewall/WsseListener.php
namespace Acme\DemoBundle\Security\Firewall;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;
use Symfony\Component\Security\Http\Firewall\ListenerInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\SecurityContextInterface;
use Symfony\Component\Security\Core\Authentication\AuthenticationManagerInterface;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Acme\DemoBundle\Security\Authentication\Token\WsseUserToken;

class WsseListener implements ListenerInterface
{
    protected $securityContext;
    protected $authenticationManager;

    public function __construct(SecurityContextInterface $securityContext, AuthenticationManagerInter
    {
        $this->securityContext = $securityContext;
        $this->authenticationManager = $authenticationManager;
    }
```

```
    public function handle(GetResponseEvent $event)
    {
        $request = $event->getRequest();

        if ($request->headers->has('x-wsse')) {

            $wsseRegex = '/UsernameToken Username="([^"]+)", PasswordDigest="([^"]+)", Nonce="([^"]+)

            if (preg_match($wsseRegex, $request->headers->get('x-wsse'), $matches)) {
                $token = new WsseUserToken();
                $token->setUser($matches[1]);

                $token->digest   = $matches[2];
                $token->nonce    = $matches[3];
                $token->created  = $matches[4];

                try {
                    $returnValue = $this->authenticationManager->authenticate($token);

                    if ($returnValue instanceof TokenInterface) {
                        return $this->securityContext->setToken($returnValue);
                    } else if ($returnValue instanceof Response) {
                        return $event->setResponse($returnValue);
                    }
                } catch (AuthenticationException $e) {
                    // you might log something here
                }
            }
        }

        $response = new Response();
        $response->setStatusCode(403);
        $event->setResponse($response);
    }
}
```

This listener checks the request for the expected *X-WSSE* header, matches the value returned for the expected WSSE information, creates a token using that information, and passes the token on to the authentication manager. If the proper information is not provided, or the authentication manager throws an `Symfony\Component\Security\Core\Exception\AuthenticationException`, a 403 Response is returned.

Note: A class not used above, the `Symfony\Component\Security\Http\Firewall\AbstractAuthenticationListe` class, is a very useful base class which provides commonly needed functionality for security extensions. This includes maintaining the token in the session, providing success / failure handlers, login form urls, and more. As WSSE does not require maintaining authentication sessions or login forms, it won't be used for this example.

The Authentication Provider

The authentication provider will do the verification of the `WsseUserToken`. Namely, the provider will verify the `Created` header value is valid within five minutes, the `Nonce` header value is unique within five minutes, and the `PasswordDigest` header value matches with the user's password.

```
// src/Acme/DemoBundle/Security/Authentication/Provider/WsseProvider.php
namespace Acme\DemoBundle\Security\Authentication\Provider;
```

```
use Symfony\Component\Security\Core\Authentication\Provider\AuthenticationProviderInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\Exception\NonceExpiredException;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Acme\DemoBundle\Security\Authentication\Token\WsseUserToken;

class WsseProvider implements AuthenticationProviderInterface
{
    private $userProvider;
    private $cacheDir;

    public function __construct(UserProviderInterface $userProvider, $cacheDir)
    {
        $this->userProvider = $userProvider;
        $this->cacheDir     = $cacheDir;
    }

    public function authenticate(TokenInterface $token)
    {
        $user = $this->userProvider->loadUserByUsername($token->getUsername());

        if ($user && $this->validateDigest($token->digest, $token->nonce, $token->created, $user->get
            $authenticatedToken = new WsseUserToken($user->getRoles());
            $authenticatedToken->setUser($user);

            return $authenticatedToken;
        }

        throw new AuthenticationException('The WSSE authentication failed.');
    }

    protected function validateDigest($digest, $nonce, $created, $secret)
    {
        // Expire timestamp after 5 minutes
        if (time() - strtotime($created) > 300) {
            return false;
        }

        // Validate nonce is unique within 5 minutes
        if (file_exists($this->cacheDir.'/'.$nonce) && file_get_contents($this->cacheDir.'/'.$nonce)
            throw new NonceExpiredException('Previously used nonce detected');
        }
        file_put_contents($this->cacheDir.'/'.$nonce, time());

        // Validate Secret
        $expected = base64_encode(sha1(base64_decode($nonce).$created.$secret, true));

        return $digest === $expected;
    }

    public function supports(TokenInterface $token)
    {
        return $token instanceof WsseUserToken;
    }
}
```

Note: The `Symfony\Component\Security\Core\Authentication\Provider\AuthenticationProviderInter`

requires an `authenticate` method on the user token, and a `supports` method, which tells the authentication manager whether or not to use this provider for the given token. In the case of multiple providers, the authentication manager will then move to the next provider in the list.

The Factory

You have created a custom token, custom listener, and custom provider. Now you need to tie them all together. How do you make your provider available to your security configuration? The answer is by using a `factory`. A factory is where you hook into the security component, telling it the name of your provider and any configuration options available for it. First, you must create a class which implements `Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Factory\SecurityFactoryInterface`

```
// src/Acme/DemoBundle/DependencyInjection/Security/Factory/WsseFactory.php
namespace Acme\DemoBundle\DependencyInjection\Security\Factory;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Reference;
use Symfony\Component\DependencyInjection\DefinitionDecorator;
use Symfony\Component\Config\Definition\Builder\NodeDefinition;
use Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Factory\SecurityFactoryInterface;

class WsseFactory implements SecurityFactoryInterface
{
    public function create(ContainerBuilder $container, $id, $config, $userProvider, $defaultEntryPo
    {
        $providerId = 'security.authentication.provider.wsse.'.$id;
        $container
            ->setDefinition($providerId, new DefinitionDecorator('wsse.security.authentication.provid
            ->replaceArgument(0, new Reference($userProvider))
        ;

        $listenerId = 'security.authentication.listener.wsse.'.$id;
        $listener = $container->setDefinition($listenerId, new DefinitionDecorator('wsse.security.aut

        return array($providerId, $listenerId, $defaultEntryPoint);
    }

    public function getPosition()
    {
        return 'pre_auth';
    }

    public function getKey()
    {
        return 'wsse';
    }

    public function addConfiguration(NodeDefinition $node)
    {}
}
```

The `Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Factory\SecurityFactoryInterf` requires the following methods:

- `create` method, which adds the listener and authentication provider to the DI container for the appropriate security context;

- `getPosition` method, which must be of type `pre_auth`, `form`, `http`, and `remember_me` and defines the position at which the provider is called;
- `getKey` method which defines the configuration key used to reference the provider;
- `addConfiguration` method, which is used to define the configuration options underneath the configuration key in your security configuration. Setting configuration options are explained later in this chapter.

Note: A class not used in this example, `Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Facto` is a very useful base class which provides commonly needed functionality for security factories. It may be useful when defining an authentication provider of a different type.

Now that you have created a factory class, the `wsse` key can be used as a firewall in your security configuration.

Note: You may be wondering "why do we need a special factory class to add listeners and providers to the dependency injection container?". This is a very good question. The reason is you can use your firewall multiple times, to secure multiple parts of your application. Because of this, each time your firewall is used, a new service is created in the DI container. The factory is what creates these new services.

Configuration

It's time to see your authentication provider in action. You will need to do a few things in order to make this work. The first thing is to add the services above to the DI container. Your factory class above makes reference to service ids that do not exist yet: `wsse.security.authentication.provider` and `wsse.security.authentication.listener`. It's time to define those services.

- *YAML*

```
# src/Acme/DemoBundle/Resources/config/services.yml
services:
  wsse.security.authentication.provider:
    class:  Acme\DemoBundle\Security\Authentication\Provider\WsseProvider
    arguments: ['', %kernel.cache_dir%/security/nonces]

  wsse.security.authentication.listener:
    class:  Acme\DemoBundle\Security\Firewall\WsseListener
    arguments: [@security.context, @security.authentication.manager]
```

- *XML*

```
<!-- src/Acme/DemoBundle/Resources/config/services.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

   <services>
       <service id="wsse.security.authentication.provider"
         class="Acme\DemoBundle\Security\Authentication\Provider\WsseProvider" public="false">
           <argument /> <!-- User Provider -->
           <argument>%kernel.cache_dir%/security/nonces</argument>
       </service>

       <service id="wsse.security.authentication.listener"
         class="Acme\DemoBundle\Security\Firewall\WsseListener" public="false">
           <argument type="service" id="security.context"/>
           <argument type="service" id="security.authentication.manager" />
       </service>
```

```
        </services>
    </container>
```

- *PHP*

```
    // src/Acme/DemoBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    $container->setDefinition('wsse.security.authentication.provider',
      new Definition(
        'Acme\DemoBundle\Security\Authentication\Provider\WsseProvider',
        array('', '%kernel.cache_dir%/security/nonces')
    ));

    $container->setDefinition('wsse.security.authentication.listener',
      new Definition(
        'Acme\DemoBundle\Security\Firewall\WsseListener', array(
          new Reference('security.context'),
          new Reference('security.authentication.manager'))
    ));
```

Now that your services are defined, tell your security context about your factory. Factories must be included in an individual configuration file, at the time of this writing. So, start first by creating the file with the factory service, tagged as `security.listener.factory`:

- *YAML*

```
    # src/Acme/DemoBundle/Resources/config/security_factories.yml
    services:
        security.authentication.factory.wsse:
            class:  Acme\DemoBundle\DependencyInjection\Security\Factory\WsseFactory
            tags:
                - { name: security.listener.factory }
```

- *XML*

```
    <!-- src/Acme/DemoBundle/Resources/config/security_factories.xml -->
    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

        <services>
            <service id="security.authentication.factory.wsse"
              class="Acme\DemoBundle\DependencyInjection\Security\Factory\WsseFactory" public="false
                <tag name="security.listener.factory" />
            </service>
        </services>
    </container>
```

New in version 2.1: Before 2.1, the factory below was added via `security.yml` instead.

As a final step, add the factory to the security extension in your bundle class.

```
// src/Acme/DemoBundle/AcmeDemoBundle.php
namespace Acme\DemoBundle;

use Acme\DemoBundle\DependencyInjection\Security\Factory\WsseFactory;
use Symfony\Component\HttpKernel\Bundle\Bundle;
use Symfony\Component\DependencyInjection\ContainerBuilder;
```

```
class AcmeDemoBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);

        $extension = $container->getExtension('security');
        $extension->addSecurityListenerFactory(new WsseFactory());
    }
}
```

You are finished! You can now define parts of your app as under WSSE protection.

```
security:
    firewalls:
        wsse_secured:
            pattern:   /api/.*
            wsse:      true
```

Congratulations! You have written your very own custom security authentication provider!

A Little Extra

How about making your WSSE authentication provider a bit more exciting? The possibilities are endless. Why don't you start by adding some spackle to that shine?

Configuration

You can add custom options under the `wsse` key in your security configuration. For instance, the time allowed before expiring the Created header item, by default, is 5 minutes. Make this configurable, so different firewalls can have different timeout lengths.

You will first need to edit `WsseFactory` and define the new option in the `addConfiguration` method.

```
class WsseFactory implements SecurityFactoryInterface
{
    # ...

    public function addConfiguration(NodeDefinition $node)
    {
      $node
        ->children()
          ->scalarNode('lifetime')->defaultValue(300)
        ->end()
      ;
    }
}
```

Now, in the `create` method of the factory, the `$config` argument will contain a 'lifetime' key, set to 5 minutes (300 seconds) unless otherwise set in the configuration. Pass this argument to your authentication provider in order to put it to use.

```
class WsseFactory implements SecurityFactoryInterface
{
    public function create(ContainerBuilder $container, $id, $config, $userProvider, $defaultEntryPo
    {
```

```
        $providerId = 'security.authentication.provider.wsse.'.$id;
        $container
            ->setDefinition($providerId,
              new DefinitionDecorator('wsse.security.authentication.provider'))
            ->replaceArgument(0, new Reference($userProvider))
            ->replaceArgument(2, $config['lifetime'])
        ;
        // ...
    }
    // ...
}
```

Note: You'll also need to add a third argument to the `wsse.security.authentication.provider` service configuration, which can be blank, but will be filled in with the lifetime in the factory. The `WsseProvider` class will also now need to accept a third constructor argument - the lifetime - which it should use instead of the hard-coded 300 seconds. These two steps are not shown here.

The lifetime of each wsse request is now configurable, and can be set to any desirable value per firewall.

```
security:
    firewalls:
        wsse_secured:
            pattern:   /api/.*
            wsse:      { lifetime: 30 }
```

The rest is up to you! Any relevant configuration items can be defined in the factory and consumed or passed to the other classes in the container.

3.1.53 How to change the Default Target Path Behavior

By default, the security component retains the information of the last request URI in a session variable named `_security.target_path`. Upon a successful login, the user is redirected to this path, as to help her continue from the last known page she visited.

On some occasions, this is unexpected. For example when the last request URI was an HTTP POST against a route which is configured to allow only a POST method, the user is redirected to this route only to get a 404 error.

To get around this behavior, you would simply need to extend the `ExceptionListener` class and override the default method named `setTargetPath()`.

First, override the `security.exception_listener.class` parameter in your configuration file. This can be done from your main configuration file (in *app/config*) or from a configuration file being imported from a bundle:

- *YAML*

```
    # src/Acme/HelloBundle/Resources/config/services.yml
    parameters:
        # ...
        security.exception_listener.class: Acme\HelloBundle\Security\Firewall\ExceptionListener
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/services.xml -->
    <parameters>
        <!-- ... -->
        <parameter key="security.exception_listener.class">Acme\HelloBundle\Security\Firewall\Except
    </parameters>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
// ...
$container->setParameter('security.exception_listener.class', 'Acme\HelloBundle\Security\Firewal
```

Next, create your own `ExceptionListener`:

```
// src/Acme/HelloBundle/Security/Firewall/ExceptionListener.php
namespace Acme\HelloBundle\Security\Firewall;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Http\Firewall\ExceptionListener as BaseExceptionListener;

class ExceptionListener extends BaseExceptionListener
{
    protected function setTargetPath(Request $request)
    {
        // Do not save target path for XHR and non-GET requests
        // You can add any more logic here you want
        if ($request->isXmlHttpRequest() || 'GET' !== $request->getMethod()) {
            return;
        }

        $request->getSession()->set('_security.target_path', $request->getUri());
    }
}
```

Add as much or few logic here as required for your scenario!

3.1.54 How to use Varnish to speed up my Website

Because Symfony2's cache uses the standard HTTP cache headers, the *Symfony2 Reverse Proxy* can easily be replaced with any other reverse proxy. Varnish is a powerful, open-source, HTTP accelerator capable of serving cached content quickly and including support for *Edge Side Includes*.

Configuration

As seen previously, Symfony2 is smart enough to detect whether it talks to a reverse proxy that understands ESI or not. It works out of the box when you use the Symfony2 reverse proxy, but you need a special configuration to make it work with Varnish. Thankfully, Symfony2 relies on yet another standard written by Akamaï (Edge Architecture), so the configuration tips in this chapter can be useful even if you don't use Symfony2.

Note: Varnish only supports the `src` attribute for ESI tags (`onerror` and `alt` attributes are ignored).

First, configure Varnish so that it advertises its ESI support by adding a `Surrogate-Capability` header to requests forwarded to the backend application:

```
sub vcl_recv {
    set req.http.Surrogate-Capability = "abc=ESI/1.0";
}
```

Then, optimize Varnish so that it only parses the Response contents when there is at least one ESI tag by checking the `Surrogate-Control` header that Symfony2 adds automatically:

```
sub vcl_fetch {
    if (beresp.http.Surrogate-Control ~ "ESI/1.0") {
        unset beresp.http.Surrogate-Control;

        // for Varnish >= 3.0
        set beresp.do_esi = true;
        // for Varnish < 3.0
        // esi;
    }
}
```

Caution: Compression with ESI was not supported in Varnish until version 3.0 (read GZIP and Varnish). If you're not using Varnish 3.0, put a web server in front of Varnish to perform the compression.

Cache Invalidation

You should never need to invalidate cached data because invalidation is already taken into account natively in the HTTP cache models (see *Cache Invalidation*).

Still, Varnish can be configured to accept a special HTTP `PURGE` method that will invalidate the cache for a given resource:

```
sub vcl_hit {
    if (req.request == "PURGE") {
        set obj.ttl = 0s;
        error 200 "Purged";
    }
}

sub vcl_miss {
    if (req.request == "PURGE") {
        error 404 "Not purged";
    }
}
```

Caution: You must protect the `PURGE` HTTP method somehow to avoid random people purging your cached data.

3.1.55 Injecting variables into all templates (i.e. Global Variables)

Sometimes you want a variable to be accessible to all the templates you use. This is possible inside your `app/config/config.yml` file:

```
# app/config/config.yml
twig:
    # ...
    globals:
        ga_tracking: UA-xxxxx-x
```

Now, the variable `ga_tracking` is available in all Twig templates:

```
<p>Our google tracking code is: {{ ga_tracking }} </p>
```

It's that easy! You can also take advantage of the built-in *Service Parameters* system, which lets you isolate or reuse the value:

```
; app/config/parameters.yml
[parameters]
    ga_tracking: UA-xxxxx-x
```

```
# app/config/config.yml
twig:
    globals:
        ga_tracking: %ga_tracking%
```

The same variable is available exactly as before.

More Complex Global Variables

If the global variable you want to set is more complicated - say an object - then you won't be able to use the above method. Instead, you'll need to create a *Twig Extension* and return the global variable as one of the entries in the `getGlobals` method.

3.1.56 How to use PHP instead of Twig for Templates

Even if Symfony2 defaults to Twig for its template engine, you can still use plain PHP code if you want. Both templating engines are supported equally in Symfony2. Symfony2 adds some nice features on top of PHP to make writing templates with PHP more powerful.

Rendering PHP Templates

If you want to use the PHP templating engine, first, make sure to enable it in your application configuration file:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    templating:    { engines: ['twig', 'php'] }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config ... >
    <!-- ... -->
    <framework:templating ... >
        <framework:engine id="twig" />
        <framework:engine id="php" />
    </framework:templating>
</framework:config>
```

- *PHP*

```
$container->loadFromExtension('framework', array(
    // ...
    'templating'      => array(
        'engines' => array('twig', 'php'),
    ),
));
```

You can now render a PHP template instead of a Twig one simply by using the `.php` extension in the template name instead of `.twig`. The controller below renders the `index.html.php` template:

```
// src/Acme/HelloBundle/Controller/HelloController.php

public function indexAction($name)
{
    return $this->render('AcmeHelloBundle:Hello:index.html.php', array('name' => $name));
}
```

Decorating Templates

More often than not, templates in a project share common elements, like the well-known header and footer. In Symfony2, we like to think about this problem differently: a template can be decorated by another one.

The `index.html.php` template is decorated by `layout.html.php`, thanks to the `extend()` call:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php $view->extend('AcmeHelloBundle::layout.html.php') ?>

Hello <?php echo $name ?>!
```

The `AcmeHelloBundle::layout.html.php` notation sounds familiar, doesn't it? It is the same notation used to reference a template. The `::` part simply means that the controller element is empty, so the corresponding file is directly stored under `views/`.

Now, let's have a look at the `layout.html.php` file:

```
<!-- src/Acme/HelloBundle/Resources/views/layout.html.php -->
<?php $view->extend('::base.html.php') ?>

<h1>Hello Application</h1>

<?php $view['slots']->output('_content') ?>
```

The layout is itself decorated by another one (`::base.html.php`). Symfony2 supports multiple decoration levels: a layout can itself be decorated by another one. When the bundle part of the template name is empty, views are looked for in the `app/Resources/views/` directory. This directory store global views for your entire project:

```
<!-- app/Resources/views/base.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->output('title', 'Hello Application') ?></title>
    </head>
    <body>
        <?php $view['slots']->output('_content') ?>
    </body>
</html>
```

For both layouts, the `$view['slots']->output('_content')` expression is replaced by the content of the child template, `index.html.php` and `layout.html.php` respectively (more on slots in the next section).

As you can see, Symfony2 provides methods on a mysterious `$view` object. In a template, the `$view` variable is always available and refers to a special object that provides a bunch of methods that makes the template engine tick.

Working with Slots

A slot is a snippet of code, defined in a template, and reusable in any layout decorating the template. In the `index.html.php` template, define a `title` slot:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php $view->extend('AcmeHelloBundle::layout.html.php') ?>

<?php $view['slots']->set('title', 'Hello World Application') ?>

Hello <?php echo $name ?>!
```

The base layout already has the code to output the title in the header:

```
<!-- app/Resources/views/base.html.php -->
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title><?php $view['slots']->output('title', 'Hello Application') ?></title>
</head>
```

The `output()` method inserts the content of a slot and optionally takes a default value if the slot is not defined. And `_content` is just a special slot that contains the rendered child template.

For large slots, there is also an extended syntax:

```
<?php $view['slots']->start('title') ?>
    Some large amount of HTML
<?php $view['slots']->stop() ?>
```

Including other Templates

The best way to share a snippet of template code is to define a template that can then be included into other templates.

Create a `hello.html.php` template:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/hello.html.php -->
Hello <?php echo $name ?>!
```

And change the `index.html.php` template to include it:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php $view->extend('AcmeHelloBundle::layout.html.php') ?>

<?php echo $view->render('AcmeHelloBundle:Hello:hello.html.php', array('name' => $name)) ?>
```

The `render()` method evaluates and returns the content of another template (this is the exact same method as the one used in the controller).

Embedding other Controllers

And what if you want to embed the result of another controller in a template? That's very useful when working with Ajax, or when the embedded template needs some variable not available in the main template.

If you create a `fancy` action, and want to include it into the `index.html.php` template, simply use the following code:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php echo $view['actions']->render('AcmeHelloBundle:Hello:fancy', array('name' => $name, 'color' =>
```

Here, the `AcmeHelloBundle:Hello:fancy` string refers to the `fancy` action of the `Hello` controller:

```
// src/Acme/HelloBundle/Controller/HelloController.php

class HelloController extends Controller
{
    public function fancyAction($name, $color)
    {
        // create some object, based on the $color variable
        $object = ...;

        return $this->render('AcmeHelloBundle:Hello:fancy.html.php', array('name' => $name, 'object'

    }

    // ...
}
```

But where is the `$view['actions']` array element defined? Like `$view['slots']`, it's called a template helper, and the next section tells you more about those.

Using Template Helpers

The Symfony2 templating system can be easily extended via helpers. Helpers are PHP objects that provide features useful in a template context. `actions` and `slots` are two of the built-in Symfony2 helpers.

Creating Links between Pages

Speaking of web applications, creating links between pages is a must. Instead of hardcoding URLs in templates, the `router` helper knows how to generate URLs based on the routing configuration. That way, all your URLs can be easily updated by changing the configuration:

```
<a href="<?php echo $view['router']->generate('hello', array('name' => 'Thomas')) ?>">
    Greet Thomas!
</a>
```

The `generate()` method takes the route name and an array of parameters as arguments. The route name is the main key under which routes are referenced and the parameters are the values of the placeholders defined in the route pattern:

```
# src/Acme/HelloBundle/Resources/config/routing.yml
hello: # The route name
    pattern:  /hello/{name}
    defaults: { _controller: AcmeHelloBundle:Hello:index }
```

Using Assets: images, JavaScripts, and stylesheets

What would the Internet be without images, JavaScripts, and stylesheets? Symfony2 provides the `assets` tag to deal with them easily:

```
<link href="<?php echo $view['assets']->getUrl('css/blog.css') ?>" rel="stylesheet" type="text/css" /

<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" />
```

The `assets` helper's main purpose is to make your application more portable. Thanks to this helper, you can move the application root directory anywhere under your web root directory without changing anything in your template's code.

Output Escaping

When using PHP templates, escape variables whenever they are displayed to the user:

```
<?php echo $view->escape($var) ?>
```

By default, the `escape()` method assumes that the variable is outputted within an HTML context. The second argument lets you change the context. For instance, to output something in a JavaScript script, use the `js` context:

```
<?php echo $view->escape($var, 'js') ?>
```

3.1.57 How to use Monolog to write Logs

Monolog is a logging library for PHP 5.3 used by Symfony2. It is inspired by the Python LogBook library.

Usage

In Monolog each logger defines a logging channel. Each channel has a stack of handlers to write the logs (the handlers can be shared).

Tip: When injecting the logger in a service you can *use a custom channel* to see easily which part of the application logged the message.

The basic handler is the `StreamHandler` which writes logs in a stream (by default in the `app/logs/prod.log` in the prod environment and `app/logs/dev.log` in the dev environment).

Monolog comes also with a powerful built-in handler for the logging in prod environment: `FingersCrossedHandler`. It allows you to store the messages in a buffer and to log them only if a message reaches the action level (ERROR in the configuration provided in the standard edition) by forwarding the messages to another handler.

To log a message simply get the logger service from the container in your controller:

```
$logger = $this->get('logger');
$logger->info('We just got the logger');
$logger->err('An error occurred');
```

Tip: Using only the methods of the `Symfony\Component\HttpKernel\Log\LoggerInterface` interface allows to change the logger implementation without changing your code.

Using several handlers

The logger uses a stack of handlers which are called successively. This allows you to log the messages in several ways easily.

- *YAML*

```
monolog:
    handlers:
        syslog:
            type: stream
            path: /var/log/symfony.log
            level: error
        main:
            type: fingers_crossed
            action_level: warning
            handler: file
        file:
            type: stream
            level: debug
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <monolog:config>
        <monolog:handler
            name="syslog"
            type="stream"
            path="/var/log/symfony.log"
            level="error"
        />
        <monolog:handler
            name="main"
            type="fingers_crossed"
            action-level="warning"
            handler="file"
        />
        <monolog:handler
            name="file"
            type="stream"
            level="debug"
        />
    </monolog:config>
</container>
```

The above configuration defines a stack of handlers which will be called in the order where they are defined.

Tip: The handler named "file" will not be included in the stack itself as it is used as a nested handler of the `fingers_crossed` handler.

Note: If you want to change the config of MonologBundle in another config file you need to redefine the whole stack. It cannot be merged because the order matters and a merge does not allow to control the order.

Changing the formatter

The handler uses a `Formatter` to format the record before logging it. All Monolog handlers use an instance of `Monolog\Formatter\LineFormatter` by default but you can replace it easily. Your formatter must implement

`Monolog\Formatter\FormatterInterface`.

- *YAML*

```
services:
    my_formatter:
        class: Monolog\Formatter\JsonFormatter
monolog:
    handlers:
        file:
            type: stream
            level: debug
            formatter: my_formatter
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <services>
        <service id="my_formatter" class="Monolog\Formatter\JsonFormatter" />
    </services>
    <monolog:config>
        <monolog:handler
            name="file"
            type="stream"
            level="debug"
            formatter="my_formatter"
        />
    </monolog:config>
</container>
```

Adding some extra data in the log messages

Monolog allows to process the record before logging it to add some extra data. A processor can be applied for the whole handler stack or only for a specific handler.

A processor is simply a callable receiving the record as it's first argument.

Processors are configured using the `monolog.processor` DIC tag. See the *reference about it*.

Adding a Session/Request Token

Sometimes it is hard to tell which entries in the log belong to which session and/or request. The following example will add a unique token for each request using a processor.

```
namespace Acme\MyBundle;

use Symfony\Component\HttpFoundation\Session;

class SessionRequestProcessor
{
    private $session;
    private $token;
```

```
    public function __construct(Session $session)
    {
        $this->session = $session;
    }

    public function processRecord(array $record)
    {
        if (null === $this->token) {
            try {
                $this->token = substr($this->session->getId(), 0, 8);
            } catch (\RuntimeException $e) {
                $this->token = '????????';
            }
            $this->token .= '-' . substr(uniqid(), -8);
        }
        $record['extra']['token'] = $this->token;

        return $record;
    }
}
```

- *YAML*

```
    services:
        monolog.formatter.session_request:
            class: Monolog\Formatter\LineFormatter
            arguments:
                - "[%%datetime%%] [%%extra.token%%] %%channel%%.%%level_name%%: %%message%%\n"

        monolog.processor.session_request:
            class: Acme\MyBundle\SessionRequestProcessor
            arguments:  [ @session ]
            tags:
                - { name: monolog.processor, method: processRecord }

    monolog:
        handlers:
            main:
                type: stream
                path: %kernel.logs_dir%/%kernel.environment%.log
                level: debug
                formatter: monolog.formatter.session_request
```

Note: If you use several handlers, you can also register the processor at the handler level instead of globally.

3.1.58 How to Configure Monolog to Email Errors

Monolog can be configured to send an email when an error occurs with an application. The configuration for this requires a few nested handlers in order to avoid receiving too many emails. This configuration looks complicated at first but each handler is fairly straight forward when it is broken down.

- *YAML*

```
    # app/config/config.yml
    monolog:
        handlers:
```

```
            mail:
                type:         fingers_crossed
                action_level: critical
                handler:      buffered
            buffered:
                type:    buffer
                handler: swift
            swift:
                type:       swift_mailer
                from_email: error@example.com
                to_email:   error@example.com
                subject:    An Error Occurred!
                level:      debug
```

- *XML*

```
    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:monolog="http://symfony.com/schema/dic/monolog"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                            http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

        <monolog:config>
            <monolog:handler
                name="mail"
                type="fingers_crossed"
                action-level="critical"
                handler="buffered"
            />
            <monolog:handler
                name="buffered"
                type="buffer"
                handler="swift"
            />
            <monolog:handler
                name="swift"
                from-email="error@example.com"
                to-email="error@example.com"
                subject="An Error Occurred!"
                level="debug"
            />
        </monolog:config>
    </container>
```

The `mail` handler is a `fingers_crossed` handler which means that it is only triggered when the action level, in this case `critical` is reached. It then logs everything including messages below the action level. The `critical` level is only triggered for 5xx HTTP code errors. The `handler` setting means that the output is then passed onto the `buffered` handler.

Tip: If you want both 400 level and 500 level errors to trigger an email, set the `action_level` to `error` instead of `critical`.

The `buffered` handler simply keeps all the messages for a request and then passes them onto the nested handler in one go. If you do not use this handler then each message will be emailed separately. This is then passed to the `swift` handler. This is the handler that actually deals with emailing you the error. The settings for this are straightforward, the to and from addresses and the subject.

You can combine these handlers with other handlers so that the errors still get logged on the server as well as the emails being sent:

- *YAML*

```yaml
# app/config/config.yml
monolog:
    handlers:
        main:
            type:         fingers_crossed
            action_level: critical
            handler:      grouped
        grouped:
            type:    group
            members: [streamed, buffered]
        streamed:
            type:  stream
            path:  %kernel.logs_dir%/%kernel.environment%.log
            level: debug
        buffered:
            type:    buffer
            handler: swift
        swift:
            type:       swift_mailer
            from_email: error@example.com
            to_email:   error@example.com
            subject:    An Error Occurred!
            level:      debug
```

- *XML*

```xml
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <monolog:config>
        <monolog:handler
            name="main"
            type="fingers_crossed"
            action_level="critical"
            handler="grouped"
        />
        <monolog:handler
            name="grouped"
            type="group"
        >
            <member type="stream"/>
            <member type="buffered"/>
        </monolog:handler>
        <monolog:handler
            name="stream"
            path="%kernel.logs_dir%/%kernel.environment%.log"
            level="debug"
        />
        <monolog:handler
            name="buffered"
            type="buffer"
```

```
                handler="swift"
            />
            <monolog:handler
                name="swift"
                from-email="error@example.com"
                to-email="error@example.com"
                subject="An Error Occurred!"
                level="debug"
            />
        </monolog:config>
    </container>
```

This uses the `group` handler to send the messages to the two group members, the `buffered` and the `stream` handlers. The messages will now be both written to the log file and emailed.

3.1.59 How to optimize your development Environment for debugging

When you work on a Symfony project on your local machine, you should use the `dev` environment (`app_dev.php` front controller). This environment configuration is optimized for two main purposes:

- Give the developer accurate feedback whenever something goes wrong (web debug toolbar, nice exception pages, profiler, ...);
- Be as similar as possible as the production environment to avoid problems when deploying the project.

Disabling the Bootstrap File and Class Caching

And to make the production environment as fast as possible, Symfony creates big PHP files in your cache containing the aggregation of PHP classes your project needs for every request. However, this behavior can confuse your IDE or your debugger. This recipe shows you how you can tweak this caching mechanism to make it friendlier when you need to debug code that involves Symfony classes.

The `app_dev.php` front controller reads as follows by default:

```
// ...

require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('dev', true);
$kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

To make you debugger happier, disable all PHP class caches by removing the call to `loadClassCache()` and by replacing the require statements like below:

```
// ...

// require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../vendor/symfony/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';
require_once __DIR__.'/../app/autoload.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;
```

```
$kernel = new AppKernel('dev', true);
// $kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

Tip: If you disable the PHP caches, don't forget to revert after your debugging session.

Some IDEs do not like the fact that some classes are stored in different locations. To avoid problems, you can either tell your IDE to ignore the PHP cache files, or you can change the extension used by Symfony for these files:

```
$kernel->loadClassCache('classes', '.php.cache');
```

3.1.60 How to extend a Class without using Inheritance

To allow multiple classes to add methods to another one, you can define the magic `__call()` method in the class you want to be extended like this:

```
class Foo
{
    // ...

    public function __call($method, $arguments)
    {
        // create an event named 'foo.method_is_not_found'
        $event = new HandleUndefinedMethodEvent($this, $method, $arguments);
        $this->dispatcher->dispatch($this, 'foo.method_is_not_found', $event);

        // no listener was able to process the event? The method does not exist
        if (!$event->isProcessed()) {
            throw new \Exception(sprintf('Call to undefined method %s::%s.', get_class($this), $metho
        }

        // return the listener returned value
        return $event->getReturnValue();
    }
}
```

This uses a special `HandleUndefinedMethodEvent` that should also be created. This is a generic class that could be reused each time you need to use this pattern of class extension:

```
use Symfony\Component\EventDispatcher\Event;

class HandleUndefinedMethodEvent extends Event
{
    protected $subject;
    protected $method;
    protected $arguments;
    protected $returnValue;
    protected $isProcessed = false;

    public function __construct($subject, $method, $arguments)
    {
        $this->subject = $subject;
        $this->method = $method;
        $this->arguments = $arguments;
    }
```

```
    public function getSubject()
    {
        return $this->subject;
    }

    public function getMethod()
    {
        return $this->method;
    }

    public function getArguments()
    {
        return $this->arguments;
    }

    /**
     * Sets the value to return and stops other listeners from being notified
     */
    public function setReturnValue($val)
    {
        $this->returnValue = $val;
        $this->isProcessed = true;
        $this->stopPropagation();
    }

    public function getReturnValue($val)
    {
        return $this->returnValue;
    }

    public function isProcessed()
    {
        return $this->isProcessed;
    }
}
```

Next, create a class that will listen to the `foo.method_is_not_found` event and *add* the method `bar()`:

```
class Bar
{
    public function onFooMethodIsNotFound(HandleUndefinedMethodEvent $event)
    {
        // we only want to respond to the calls to the 'bar' method
        if ('bar' != $event->getMethod()) {
            // allow another listener to take care of this unknown method
            return;
        }

        // the subject object (the foo instance)
        $foo = $event->getSubject();

        // the bar method arguments
        $arguments = $event->getArguments();

        // do something
        // ...

        // set the return value
        $event->setReturnValue($someValue);
```

```
    }
}
```

Finally, add the new `bar` method to the `Foo` class by register an instance of `Bar` with the `foo.method_is_not_found` event:

```
$bar = new Bar();
$dispatcher->addListener('foo.method_is_not_found', $bar);
```

3.1.61 How to customize a Method Behavior without using Inheritance

Doing something before or after a Method Call

If you want to do something just before, or just after a method is called, you can dispatch an event respectively at the beginning or at the end of the method:

```
class Foo
{
    // ...

    public function send($foo, $bar)
    {
        // do something before the method
        $event = new FilterBeforeSendEvent($foo, $bar);
        $this->dispatcher->dispatch('foo.pre_send', $event);

        // get $foo and $bar from the event, they may have been modified
        $foo = $event->getFoo();
        $bar = $event->getBar();
        // the real method implementation is here
        // $ret = ...;

        // do something after the method
        $event = new FilterSendReturnValue($ret);
        $this->dispatcher->dispatch('foo.post_send', $event);

        return $event->getReturnValue();
    }
}
```

In this example, two events are thrown: `foo.pre_send`, before the method is executed, and `foo.post_send` after the method is executed. Each uses a custom Event class to communicate information to the listeners of the two events. These event classes would need to be created by you and should allow, in this example, the variables `$foo`, `$bar` and `$ret` to be retrieved and set by the listeners.

For example, assuming the `FilterSendReturnValue` has a `setReturnValue` method, one listener might look like this:

```
public function onFooPostSend(FilterSendReturnValue $event)
{
    $ret = $event->getReturnValue();
    // modify the original ``$ret`` value

    $event->setReturnValue($ret);
}
```

3.1.62 How to register a new Request Format and Mime Type

Every `Request` has a "format" (e.g. `html`, `json`), which is used to determine what type of content to return in the `Response`. In fact, the request format, accessible via **:method:'Symfony\\Component\\HttpFoundation\\Request::getRequestFormat'**, is used to set the MIME type of the `Content-Type` header on the `Response` object. Internally, Symfony contains a map of the most common formats (e.g. `html`, `json`) and their associated MIME types (e.g. `text/html`, `application/json`). Of course, additional format-MIME type entries can easily be added. This document will show how you can add the `jsonp` format and corresponding MIME type.

Create an `kernel.request` Listener

The key to defining a new MIME type is to create a class that will "listen" to the `kernel.request` event dispatched by the Symfony kernel. The `kernel.request` event is dispatched early in Symfony's request handling process and allows you to modify the request object.

Create the following class, replacing the path with a path to a bundle in your project:

```
// src/Acme/DemoBundle/RequestListener.php
namespace Acme\DemoBundle;

use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class RequestListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $event->getRequest()->setFormat('jsonp', 'application/javascript');
    }
}
```

Registering your Listener

As for any other listener, you need to add it in one of your configuration file and register it as a listener by adding the `kernel.event_listener` tag:

- *XML*

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
    <services>
    <service id="acme.demobundle.listener.request" class="Acme\DemoBundle\RequestListener">
        <tag name="kernel.event_listener" event="kernel.request" method="onKernelRequest" />
    </service>
    </services>
</container>
```

- *YAML*

```
# app/config/config.yml
services:
    acme.demobundle.listener.request:
```

```
            class: Acme\DemoBundle\RequestListener
            tags:
                - { name: kernel.event_listener, event: kernel.request, method: onKernelRequest }
```

- *PHP*

```
    # app/config/config.php
    $definition = new Definition('Acme\DemoBundle\RequestListener');
    $definition->addTag('kernel.event_listener', array('event' => 'kernel.request', 'method' => 'onK
    $container->setDefinition('acme.demobundle.listener.request', $definition);
```

At this point, the `acme.demobundle.listener.request` service has been configured and will be notified when the Symfony kernel dispatches the `kernel.request` event.

Tip: You can also register the listener in a configuration extension class (see *Importing Configuration via Container Extensions* for more information).

3.1.63 How to create a custom Data Collector

The Symfony2 `Profiler` delegates data collecting to data collectors. Symfony2 comes bundled with a few of them, but you can easily create your own.

Creating a Custom Data Collector

Creating a custom data collector is as simple as implementing the `Symfony\Component\HttpKernel\DataCollector\DataC`

```
interface DataCollectorInterface
{
    /**
     * Collects data for the given Request and Response.
     *
     * @param Request    $request   A Request instance
     * @param Response   $response  A Response instance
     * @param \Exception $exception An Exception instance
     */
    function collect(Request $request, Response $response, \Exception $exception = null);

    /**
     * Returns the name of the collector.
     *
     * @return string The collector name
     */
    function getName();
}
```

The `getName()` method must return a unique name. This is used to access the information later on (see How to use the Profiler in a Functional Test for instance).

The `collect()` method is responsible for storing the data it wants to give access to in local properties.

Caution: As the profiler serializes data collector instances, you should not store objects that cannot be serialized (like PDO objects), or you need to provide your own `serialize()` method.

Most of the time, it is convenient to extend `Symfony\Component\HttpKernel\DataCollector\DataCollector` and populate the `$this->data` property (it takes care of serializing the `$this->data` property):

```
class MemoryDataCollector extends DataCollector
{
    public function collect(Request $request, Response $response, \Exception $exception = null)
    {
        $this->data = array(
            'memory' => memory_get_peak_usage(true),
        );
    }

    public function getMemory()
    {
        return $this->data['memory'];
    }

    public function getName()
    {
        return 'memory';
    }
}
```

Enabling Custom Data Collectors

To enable a data collector, add it as a regular service in one of your configuration, and tag it with `data_collector`:

- *YAML*

```
    services:
        data_collector.your_collector_name:
            class: Fully\Qualified\Collector\Class\Name
            tags:
                - { name: data_collector }
```

- *XML*

```
    <service id="data_collector.your_collector_name" class="Fully\Qualified\Collector\Class\Name">
        <tag name="data_collector" />
    </service>
```

- *PHP*

```
    $container
        ->register('data_collector.your_collector_name', 'Fully\Qualified\Collector\Class\Name')
        ->addTag('data_collector')
    ;
```

Adding Web Profiler Templates

When you want to display the data collected by your Data Collector in the web debug toolbar or the web profiler, create a Twig template following this skeleton:

```
{% extends 'WebProfilerBundle:Profiler:layout.html.twig' %}

{% block toolbar %}
    {# the web debug toolbar content #}
{% endblock %}

{% block head %}
```

```
    {# if the web profiler panel needs some specific JS or CSS files #}
{% endblock %}

{% block menu %}
    {# the menu content #}
{% endblock %}

{% block panel %}
    {# the panel content #}
{% endblock %}
```

Each block is optional. The `toolbar` block is used for the web debug toolbar and `menu` and `panel` are used to add a panel to the web profiler.

All blocks have access to the `collector` object.

Tip: Built-in templates use a base64 encoded image for the toolbar (`<img src="src="..."`). You can easily calculate the base64 value for an image with this little script: `echo base64_encode(file_get_contents($_SERVER['argv'][1]));`.

To enable the template, add a `template` attribute to the `data_collector` tag in your configuration. For example, assuming your template is in some `AcmeDebugBundle`:

- *YAML*

```
services:
    data_collector.your_collector_name:
        class: Acme\DebugBundle\Collector\Class\Name
        tags:
            - { name: data_collector, template: "AcmeDebug:Collector:templatename", id: "your_cc
```

- *XML*

```
<service id="data_collector.your_collector_name" class="Acme\DebugBundle\Collector\Class\Name">
    <tag name="data_collector" template="AcmeDebug:Collector:templatename" id="your_collector_na
</service>
```

- *PHP*

```
$container
    ->register('data_collector.your_collector_name', 'Acme\DebugBundle\Collector\Class\Name')
    ->addTag('data_collector', array('template' => 'AcmeDebugBundle:Collector:templatename', 'id
;
```

3.1.64 How to Create a SOAP Web Service in a Symfony2 Controller

Setting up a controller to act as a SOAP server is simple with a couple tools. You must, of course, have the PHP SOAP extension installed. As the PHP SOAP extension can not currently generate a WSDL, you must either create one from scratch or use a 3rd party generator.

Note: There are several SOAP server implementations available for use with PHP. Zend SOAP and NuSOAP are two examples. Although we use the PHP SOAP extension in our examples, the general idea should still be applicable to other implementations.

SOAP works by exposing the methods of a PHP object to an external entity (i.e. the person using the SOAP service). To start, create a class - `HelloService` - which represents the functionality that you'll expose in your SOAP service. In this case, the SOAP service will allow the client to call a method called `hello`, which happens to send an email:

```
namespace Acme\SoapBundle;

class HelloService
{
    private $mailer;

    public function __construct(\Swift_Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function hello($name)
    {

        $message = \Swift_Message::newInstance()
                                 ->setTo('me@example.com')
                                 ->setSubject('Hello Service')
                                 ->setBody($name . ' says hi!');

        $this->mailer->send($message);


        return 'Hello, ' . $name;
    }
}
```

Next, you can train Symfony to be able to create an instance of this class. Since the class sends an e-mail, it's been designed to accept a `Swift_Mailer` instance. Using the Service Container, we can configure Symfony to construct a `HelloService` object properly:

- *YAML*

```
# app/config/config.yml
services:
    hello_service:
        class: Acme\DemoBundle\Services\HelloService
        arguments: [mailer]
```

- *XML*

```
<!-- app/config/config.xml -->
<services>
 <service id="hello_service" class="Acme\DemoBundle\Services\HelloService">
  <argument>mailer</argument>
 </service>
</services>
```

Below is an example of a controller that is capable of handling a SOAP request. If `indexAction()` is accessible via the route `/soap`, then the WSDL document can be retrieved via `/soap?wsdl`.

```
namespace Acme\SoapBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HelloServiceController extends Controller
{
    public function indexAction()
    {
        $server = new \SoapServer('/path/to/hello.wsdl');
```

```
        $server->setObject($this->get('hello_service'));

        $response = new Response();
        $response->headers->set('Content-Type', 'text/xml; charset=ISO-8859-1');

        ob_start();
        $server->handle();
        $response->setContent(ob_get_clean());

        return $response;
    }
}
```

Take note of the calls to `ob_start()` and `ob_get_clean()`. These methods control output buffering which allows you to "trap" the echoed output of `$server->handle()`. This is necessary because Symfony expects your controller to return a `Response` object with the output as its "content". You must also remember to set the "Content-Type" header to "text/xml", as this is what the client will expect. So, you use `ob_start()` to start buffering the STDOUT and use `ob_get_clean()` to dump the echoed output into the content of the Response and clear the output buffer. Finally, you're ready to return the `Response`.

Below is an example calling the service using NuSOAP client. This example assumes that the `indexAction` in the controller above is accessible via the route `/soap`:

```
$client = new \soapclient('http://example.com/app.php/soap?wsdl', true);

$result = $client->call('hello', array('name' => 'Scott'));
```

An example WSDL is below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <definitions xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:tns="urn:arnleadservicewsdl"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     targetNamespace="urn:helloservicewsdl">
  <types>
   <xsd:schema targetNamespace="urn:hellowsdl">
    <xsd:import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
    <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/" />
   </xsd:schema>
  </types>
  <message name="helloRequest">
   <part name="name" type="xsd:string" />
  </message>
  <message name="helloResponse">
   <part name="return" type="xsd:string" />
  </message>
  <portType name="hellowsdlPortType">
   <operation name="hello">
    <documentation>Hello World</documentation>
    <input message="tns:helloRequest"/>
    <output message="tns:helloResponse"/>
   </operation>
  </portType>
  <binding name="hellowsdlBinding" type="tns:hellowsdlPortType">
```

```
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="hello">
   <soap:operation soapAction="urn:arnleadservicewsdl#hello" style="rpc"/>
   <input>
    <soap:body use="encoded" namespace="urn:hellowsdl"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   </input>
   <output>
    <soap:body use="encoded" namespace="urn:hellowsdl"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   </output>
  </operation>
 </binding>
 <service name="hellowsdl">
  <port name="hellowsdlPort" binding="tns:hellowsdlBinding">
   <soap:address location="http://example.com/app.php/soap" />
  </port>
 </service>
</definitions>
```

3.1.65 How Symfony2 differs from symfony1

The Symfony2 framework embodies a significant evolution when compared with the first version of the framework. Fortunately, with the MVC architecture at its core, the skills used to master a symfony1 project continue to be very relevant when developing in Symfony2. Sure, `app.yml` is gone, but routing, controllers and templates all remain.

In this chapter, we'll walk through the differences between symfony1 and Symfony2. As you'll see, many tasks are tackled in a slightly different way. You'll come to appreciate these minor differences as they promote stable, predictable, testable and decoupled code in your Symfony2 applications.

So, sit back and relax as we take you from "then" to "now".

Directory Structure

When looking at a Symfony2 project - for example, the Symfony2 Standard - you'll notice a very different directory structure than in symfony1. The differences, however, are somewhat superficial.

The `app/` Directory

In symfony1, your project has one or more applications, and each lives inside the `apps/` directory (e.g. `apps/frontend`). By default in Symfony2, you have just one application represented by the `app/` directory. Like in symfony1, the `app/` directory contains configuration specific to that application. It also contains application-specific cache, log and template directories as well as a `Kernel` class (`AppKernel`), which is the base object that represents the application.

Unlike symfony1, almost no PHP code lives in the `app/` directory. This directory is not meant to house modules or library files as it did in symfony1. Instead, it's simply the home of configuration and other resources (templates, translation files).

The `src/` Directory

Put simply, your actual code goes here. In Symfony2, all actual application-code lives inside a bundle (roughly equivalent to a symfony1 plugin) and, by default, each bundle lives inside the `src` directory. In that way, the `src`

directory is a bit like the `plugins` directory in symfony1, but much more flexible. Additionally, while *your* bundles will live in the `src/` directory, third-party bundles may live in the `vendor/bundles/` directory.

To get a better picture of the `src/` directory, let's first think of a symfony1 application. First, part of your code likely lives inside one or more applications. Most commonly these include modules, but could also include any other PHP classes you put in your application. You may have also created a `schema.yml` file in the `config` directory of your project and built several model files. Finally, to help with some common functionality, you're using several third-party plugins that live in the `plugins/` directory. In other words, the code that drives your application lives in many different places.

In Symfony2, life is much simpler because *all* Symfony2 code must live in a bundle. In our pretend symfony1 project, all the code *could* be moved into one or more plugins (which is a very good practice, in fact). Assuming that all modules, PHP classes, schema, routing configuration, etc were moved into a plugin, the symfony1 `plugins/` directory would be very similar to the Symfony2 `src/` directory.

Put simply again, the `src/` directory is where your code, assets, templates and most anything else specific to your project will live.

The `vendor/` Directory

The `vendor/` directory is basically equivalent to the `lib/vendor/` directory in symfony1, which was the conventional directory for all vendor libraries and bundles. By default, you'll find the Symfony2 library files in this directory, along with several other dependent libraries such as Doctrine2, Twig and Swiftmailer. 3rd party Symfony2 bundles usually live in the `vendor/bundles/`.

The `web/` Directory

Not much has changed in the `web/` directory. The most noticeable difference is the absence of the `css/`, `js/` and `images/` directories. This is intentional. Like with your PHP code, all assets should also live inside a bundle. With the help of a console command, the `Resources/public/` directory of each bundle is copied or symbolically-linked to the `web/bundles/` directory. This allows you to keep assets organized inside your bundle, but still make them available to the public. To make sure that all bundles are available, run the following command:

```
php app/console assets:install web
```

Note: This command is the Symfony2 equivalent to the symfony1 `plugin:publish-assets` command.

Autoloading

One of the advantages of modern frameworks is never needing to worry about requiring files. By making use of an autoloader, you can refer to any class in your project and trust that it's available. Autoloading has changed in Symfony2 to be more universal, faster, and independent of needing to clear your cache.

In symfony1, autoloading was done by searching the entire project for the presence of PHP class files and caching this information in a giant array. That array told symfony1 exactly which file contained each class. In the production environment, this caused you to need to clear the cache when classes were added or moved.

In Symfony2, a new class - `UniversalClassLoader` - handles this process. The idea behind the autoloader is simple: the name of your class (including the namespace) must match up with the path to the file containing that class. Take the `FrameworkExtraBundle` from the Symfony2 Standard Edition as an example:

```
namespace Sensio\Bundle\FrameworkExtraBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
```

```
// ...

class SensioFrameworkExtraBundle extends Bundle
{
    // ...
```

The file itself lives at `vendor/bundle/Sensio/Bundle/FrameworkExtraBundle/SensioFrameworkExtraBundle.p`
As you can see, the location of the file follows the namespace of the class. Specifically, the namespace, `Sensio\Bundle\FrameworkExtraBundle`, spells out the directory that the file should live in (`vendor/bundle/Sensio/Bundle/FrameworkExtraBundle`). This is because, in the `app/autoload.php` file, you'll configure Symfony to look for the `Sensio` namespace in the `vendor/bundle` directory:

```
// app/autoload.php

// ...
$loader->registerNamespaces(array(
    // ...
    'Sensio'           => __DIR__.'/../vendor/bundles',
));
```

If the file did *not* live at this exact location, you'd receive a `Class "Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle" does not exist.` error. In Symfony2, a "class does not exist" means that the suspect class namespace and physical location do not match. Basically, Symfony2 is looking in one exact location for that class, but that location doesn't exist (or contains a different class). In order for a class to be autoloaded, you **never need to clear your cache** in Symfony2.

As mentioned before, for the autoloader to work, it needs to know that the `Sensio` namespace lives in the `vendor/bundles` directory and that, for example, the `Doctrine` namespace lives in the `vendor/doctrine/lib/` directory. This mapping is entirely controlled by you via the `app/autoload.php` file.

If you look at the `HelloController` from the Symfony2 Standard Edition you can see that it lives in the `Acme\DemoBundle\Controller` namespace. Yet, the `Acme` namespace is not defined in the `app/autoload.php`. By default you do not need to explicitly configure the location of bundles that live in the `src/` directory. The `UniversalClassLoader` is configured to fallback to the `src/` directory using its `registerNamespaceFallbacks` method:

```
// app/autoload.php

// ...
$loader->registerNamespaceFallbacks(array(
    __DIR__.'/../src',
));
```

Using the Console

In symfony1, the console is in the root directory of your project and is called `symfony`:

```
php symfony
```

In Symfony2, the console is now in the app sub-directory and is called `console`:

```
php app/console
```

Applications

In a symfony1 project, it is common to have several applications: one for the frontend and one for the backend for instance.

In a Symfony2 project, you only need to create one application (a blog application, an intranet application, ...). Most of the time, if you want to create a second application, you might instead create another project and share some bundles between them.

And if you need to separate the frontend and the backend features of some bundles, you can create sub-namespaces for controllers, sub-directories for templates, different semantic configurations, separate routing configurations, and so on.

Of course, there's nothing wrong with having multiple applications in your project, that's entirely up to you. A second application would mean a new directory, e.g. `my_app/`, with the same basic setup as the `app/` directory.

Tip: Read the definition of a Project, an Application, and a Bundle in the glossary.

Bundles and Plugins

In a symfony1 project, a plugin could contain configuration, modules, PHP libraries, assets and anything else related to your project. In Symfony2, the idea of a plugin is replaced by the "bundle". A bundle is even more powerful than a plugin because the core Symfony2 framework is brought in via a series of bundles. In Symfony2, bundles are first-class citizens that are so flexible that even core code itself is a bundle.

In symfony1, a plugin must be enabled inside the `ProjectConfiguration` class:

```
// config/ProjectConfiguration.class.php
public function setup()
{
    $this->enableAllPluginsExcept(array(/* some plugins here */));
}
```

In Symfony2, the bundles are activated inside the application kernel:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        // ...
        new Acme\DemoBundle\AcmeDemoBundle(),
    );

    return $bundles;
}
```

Routing (`routing.yml`) and Configuration (`config.yml`)

In symfony1, the `routing.yml` and `app.yml` configuration files were automatically loaded inside any plugin. In Symfony2, routing and application configuration inside a bundle must be included manually. For example, to include a routing resource from a bundle called `AcmeDemoBundle`, you can do the following:

```
# app/config/routing.yml
_hello:
    resource: "@AcmeDemoBundle/Resources/config/routing.yml"
```

This will load the routes found in the `Resources/config/routing.yml` file of the `AcmeDemoBundle`. The special `@AcmeDemoBundle` is a shortcut syntax that, internally, resolves to the full path to that bundle.

You can use this same strategy to bring in configuration from a bundle:

```
# app/config/config.yml
imports:
    - { resource: "@AcmeDemoBundle/Resources/config/config.yml" }
```

In Symfony2, configuration is a bit like `app.yml` in symfony1, except much more systematic. With `app.yml`, you could simply create any keys you wanted. By default, these entries were meaningless and depended entirely on how you used them in your application:

```
# some app.yml file from symfony1
all:
  email:
    from_address:  foo.bar@example.com
```

In Symfony2, you can also create arbitrary entries under the `parameters` key of your configuration:

```
parameters:
    email.from_address: foo.bar@example.com
```

You can now access this from a controller, for example:

```
public function helloAction($name)
{
    $fromAddress = $this->container->getParameter('email.from_address');
}
```

In reality, the Symfony2 configuration is much more powerful and is used primarily to configure objects that you can use. For more information, see the chapter titled "Service Container".

- **Workflow**
  - How to Create and store a Symfony2 Project in git
  - How to Create and store a Symfony2 Project in Subversion
- **Controllers**
  - How to customize Error Pages
  - How to define Controllers as Services
- **Routing**
  - How to force routes to always use HTTPS or HTTP
  - How to allow a "/" character in a route parameter
- **Handling JavaScript and CSS Assets**
  - How to Use Assetic for Asset Management
  - How to Minify JavaScripts and Stylesheets with YUI Compressor
  - How to Use Assetic For Image Optimization with Twig Functions
  - How to Apply an Assetic Filter to a Specific File Extension

- **Database Interaction (Doctrine)**
  - How to handle File Uploads with Doctrine
  - Doctrine Extensions: Timestampable: Sluggable, Translatable, etc.
  - Registering Event Listeners and Subscribers
  - How to use Doctrine's DBAL Layer
  - How to generate Entities from an Existing Database
  - How to work with Multiple Entity Managers
  - Registering Custom DQL Functions
- **Forms and Validation**
  - How to customize Form Rendering
  - Using Data Transformers
  - How to Dynamically Generate Forms Using Form Events
  - How to Embed a Collection of Forms
  - How to Create a Custom Form Field Type
  - How to create a Custom Validation Constraint
  - (doctrine) How to handle File Uploads with Doctrine
- **Configuration and the Service Container**
  - How to Master and Create new Environments
  - How to Set External Parameters in the Service Container
  - How to Use a Factory to Create Services
  - How to Manage Common Dependencies with Parent Services
  - How to work with Scopes
  - How to make your Services use Tags
  - How to use PdoSessionStorage to store Sessions in the Database
- **Bundles**
  - Bundle Structure and Best Practices
  - How to use Bundle Inheritance to Override parts of a Bundle
  - How to Override any Part of a Bundle
  - How to expose a Semantic Configuration for a Bundle
- **Emailing**
  - How to send an Email
  - How to use Gmail to send Emails
  - How to Work with Emails During Development
  - How to Spool Email
- **Testing**
  - How to simulate HTTP Authentication in a Functional Test

Read the Cookbook.

CHAPTER 4

Components

4.1 The Components

4.1.1 The ClassLoader Component

The ClassLoader Component loads your project classes automatically if they follow some standard PHP conventions.

Whenever you use an undefined class, PHP uses the autoloading mechanism to delegate the loading of a file defining the class. Symfony2 provides a "universal" autoloader, which is able to load classes from files that implement one of the following conventions:

- The technical interoperability standards for PHP 5.3 namespaces and class names;
- The PEAR naming convention for classes.

If your classes and the third-party libraries you use for your project follow these standards, the Symfony2 autoloader is the only autoloader you will ever need.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/ClassLoader);
- Install it via PEAR (*pear.symfony.com/ClassLoader*);
- Install it via Composer (*symfony/class-loader* on Packagist).

Usage

New in version 2.1: The `useIncludePath` method was added in Symfony 2.1.

Registering the `Symfony\Component\ClassLoader\UniversalClassLoader` autoloader is straightforward:

```
require_once '/path/to/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';

use Symfony\Component\ClassLoader\UniversalClassLoader;

$loader = new UniversalClassLoader();
```

```
// You can search the include_path as a last resort.
$loader->useIncludePath(true);

$loader->register();
```

For minor performance gains class paths can be cached in memory using APC by registering the `Symfony\Component\ClassLoader\ApcUniversalClassLoader`:

```
require_once '/path/to/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';
require_once '/path/to/src/Symfony/Component/ClassLoader/ApcUniversalClassLoader.php';

use Symfony\Component\ClassLoader\ApcUniversalClassLoader;

$loader = new ApcUniversalClassLoader('apc.prefix.');
$loader->register();
```

The autoloader is useful only if you add some libraries to autoload.

Note: The autoloader is automatically registered in a Symfony2 application (see `app/autoload.php`).

If the classes to autoload use namespaces, use the **:method:'Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerNa** or **:method:'Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerNamespaces'** methods:

```
$loader->registerNamespace('Symfony', __DIR__.'/vendor/symfony/src');

$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/../vendor/symfony/src',
    'Monolog' => __DIR__.'/../vendor/monolog/src',
));

$loader->register();
```

For classes that follow the PEAR naming convention, use the **:method:'Symfony\\Component\\ClassLoader\\UniversalClassLoader::** or **:method:'Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerPrefixes'** methods:

```
$loader->registerPrefix('Twig_', __DIR__.'/vendor/twig/lib');

$loader->registerPrefixes(array(
    'Swift_' => __DIR__.'/vendor/swiftmailer/lib/classes',
    'Twig_'  => __DIR__.'/vendor/twig/lib',
));

$loader->register();
```

Note: Some libraries also require their root path be registered in the PHP include path (`set_include_path()`).

Classes from a sub-namespace or a sub-hierarchy of PEAR classes can be looked for in a location list to ease the vendoring of a sub-set of classes for large projects:

```
$loader->registerNamespaces(array(
    'Doctrine\\Common'           => __DIR__.'/vendor/doctrine-common/lib',
    'Doctrine\\DBAL\\Migrations' => __DIR__.'/vendor/doctrine-migrations/lib',
    'Doctrine\\DBAL'             => __DIR__.'/vendor/doctrine-dbal/lib',
    'Doctrine'                   => __DIR__.'/vendor/doctrine/lib',
));

$loader->register();
```

In this example, if you try to use a class in the `Doctrine\Common` namespace or one of its children, the autoloader will first look for the class under the `doctrine-common` directory, and it will then fallback to the default `Doctrine` directory (the last one configured) if not found, before giving up. The order of the registrations is significant in this case.

4.1.2 The Console Component

The Console component eases the creation of beautiful and testable command line interfaces.

Symfony2 ships with a Console component, which allows you to create command-line commands. Your console commands can be used for any recurring task, such as cronjobs, imports, or other batch jobs.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/Console);
- Install it via PEAR (*pear.symfony.com/Console*);
- Install it via Composer (*symfony/console* on Packagist).

Creating a basic Command

To make the console commands available automatically with Symfony2, create a `Command` directory inside your bundle and create a php file suffixed with `Command.php` for each command that you want to provide. For example, if you want to extend the `AcmeDemoBundle` (available in the Symfony Standard Edition) to greet us from the command line, create `GreetCommand.php` and add the following to it:

```
// src/Acme/DemoBundle/Command/GreetCommand.php
namespace Acme\DemoBundle\Command;

use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

class GreetCommand extends ContainerAwareCommand
{
    protected function configure()
    {
        $this
            ->setName('demo:greet')
            ->setDescription('Greet someone')
            ->addArgument('name', InputArgument::OPTIONAL, 'Who do you want to greet?')
            ->addOption('yell', null, InputOption::VALUE_NONE, 'If set, the task will yell in upperca
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $name = $input->getArgument('name');
        if ($name) {
            $text = 'Hello '.$name;
        } else {
```

```
            $text = 'Hello';
        }

        if ($input->getOption('yell')) {
            $text = strtoupper($text);
        }

        $output->writeln($text);
    }
}
```

You also need to create the file to run at the command line which creates an `Application` and adds commands to it:

Test the new console command by running the following

```
app/console demo:greet Fabien
```

This will print the following to the command line:

```
Hello Fabien
```

You can also use the `--yell` option to make everything uppercase:

```
app/console demo:greet Fabien --yell
```

This prints:

```
HELLO FABIEN
```

Coloring the Output

Whenever you output text, you can surround the text with tags to color its output. For example:

```
// green text
$output->writeln('<info>foo</info>');

// yellow text
$output->writeln('<comment>foo</comment>');

// black text on a cyan background
$output->writeln('<question>foo</question>');

// white text on a red background
$output->writeln('<error>foo</error>');
```

Using Command Arguments

The most interesting part of the commands are the arguments and options that you can make available. Arguments are the strings - separated by spaces - that come after the command name itself. They are ordered, and can be optional or required. For example, add an optional `last_name` argument to the command and make the `name` argument required:

```
$this
    // ...
    ->addArgument('name', InputArgument::REQUIRED, 'Who do you want to greet?')
```

```
    ->addArgument('last_name', InputArgument::OPTIONAL, 'Your last name?')
    // ...
```

You now have access to a `last_name` argument in your command:

```
if ($lastName = $input->getArgument('last_name')) {
    $text .= ' '.$lastName;
}
```

The command can now be used in either of the following ways:

```
app/console demo:greet Fabien
app/console demo:greet Fabien Potencier
```

Using Command Options

Unlike arguments, options are not ordered (meaning you can specify them in any order) and are specified with two dashes (e.g. `--yell` - you can also declare a one-letter shortcut that you can call with a single dash like `-y`). Options are *always* optional, and can be setup to accept a value (e.g. `dir=src`) or simply as a boolean flag without a value (e.g. `yell`).

Tip: It is also possible to make an option *optionally* accept a value (so that `--yell` or `yell=loud` work). Options can also be configured to accept an array of values.

For example, add a new option to the command that can be used to specify how many times in a row the message should be printed:

```
$this
    // ...
    ->addOption('iterations', null, InputOption::VALUE_REQUIRED, 'How many times should the message b
```

Next, use this in the command to print the message multiple times:

```
for ($i = 0; $i < $input->getOption('iterations'); $i++) {
    $output->writeln($text);
}
```

Now, when you run the task, you can optionally specify a `--iterations` flag:

```
app/console demo:greet Fabien

app/console demo:greet Fabien --iterations=5
```

The first example will only print once, since `iterations` is empty and defaults to `1` (the last argument of `addOption`). The second example will print five times.

Recall that options don't care about their order. So, either of the following will work:

```
app/console demo:greet Fabien --iterations=5 --yell
app/console demo:greet Fabien --yell --iterations=5
```

There are 4 option variants you can use:

Option	Value
InputOption::VALUE_IS_ARRAY	This option accepts multiple values
InputOption::VALUE_NONE	Do not accept input for this option (e.g. `--yell`)
InputOption::VALUE_REQUIRED	This value is required (e.g. `iterations=5`)
InputOption::VALUE_OPTIONAL	This value is optional

You can combine VALUE_IS_ARRAY with VALUE_REQUIRED or VALUE_OPTIONAL like this:

```
$this
    // ...
    ->addOption('iterations', null, InputOption::VALUE_REQUIRED | InputOption::VALUE_IS_ARRAY, 'How r
```

Asking the User for Information

When creating commands, you have the ability to collect more information from the user by asking him/her questions. For example, suppose you want to confirm an action before actually executing it. Add the following to your command:

```
$dialog = $this->getHelperSet()->get('dialog');
if (!$dialog->askConfirmation($output, '<question>Continue with this action?</question>', false)) {
    return;
}
```

In this case, the user will be asked "Continue with this action", and unless they answer with `y`, the task will stop running. The third argument to `askConfirmation` is the default value to return if the user doesn't enter any input.

You can also ask questions with more than a simple yes/no answer. For example, if you needed to know the name of something, you might do the following:

```
$dialog = $this->getHelperSet()->get('dialog');
$name = $dialog->ask($output, 'Please enter the name of the widget', 'foo');
```

Testing Commands

Symfony2 provides several tools to help you test your commands. The most useful one is the `Symfony\Component\Console\Tester\CommandTester` class. It uses special input and output classes to ease testing without a real console:

```
use Symfony\Component\Console\Tester\CommandTester;
use Symfony\Bundle\FrameworkBundle\Console\Application;
use Acme\DemoBundle\Command\GreetCommand;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
    public function testExecute()
    {
        $application = new Application();
        $application->add(new GreetCommand());

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(array('command' => $command->getName()));

        $this->assertRegExp('/.../', $commandTester->getDisplay());

        // ...
    }
}
```

The **:method:'Symfony\\Component\\Console\\Tester\\CommandTester::getDisplay'** method returns what would have been displayed during a normal call from the console.

You can test sending arguments and options to the command by passing them as an array to the **:method:'Symfony\\Component\\Console\\Tester\\CommandTester::getDisplay'** method:

```
use Symfony\Component\Console\Tester\CommandTester;
use Symfony\Bundle\FrameworkBundle\Console\Application;
use Acme\DemoBundle\Command\GreetCommand;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{

    //--

    public function testNameIsOutput()
    {
        $application = new Application();
        $application->add(new GreetCommand());

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(
            array('command' => $command->getName(), 'name' => 'Fabien')
        );

        $this->assertRegExp('/Fabien/', $commandTester->getDisplay());
    }
}
```

Tip: You can also test a whole console application by using `Symfony\Component\Console\Tester\ApplicationTester`.

Calling an existing Command

If a command depends on another one being run before it, instead of asking the user to remember the order of execution, you can call it directly yourself. This is also useful if you want to create a "meta" command that just runs a bunch of other commands (for instance, all commands that need to be run when the project's code has changed on the production servers: clearing the cache, generating Doctrine2 proxies, dumping Assetic assets, ...).

Calling a command from another one is straightforward:

```
protected function execute(InputInterface $input, OutputInterface $output)
{
    $command = $this->getApplication()->find('demo:greet');

    $arguments = array(
        'command' => 'demo:greet',
        'name'    => 'Fabien',
        '--yell'  => true,
    );

    $input = new ArrayInput($arguments);
    $returnCode = $command->run($input, $output);

    // ...
}
```

First, you **:method:'Symfony\\Component\\Console\\Command\\Command::find'** the command you want to execute by passing the command name.

Then, you need to create a new `Symfony\Component\Console\Input\ArrayInput` with the arguments and options you want to pass to the command.

Eventually, calling the `run()` method actually executes the command and returns the returned code from the command (`0` if everything went fine, any other integer otherwise).

Note: Most of the time, calling a command from code that is not executed on the command line is not a good idea for several reasons. First, the command's output is optimized for the console. But more important, you can think of a command as being like a controller; it should use the model to do something and display feedback to the user. So, instead of calling a command from the Web, refactor your code and move the logic to a new class.

4.1.3 The CssSelector Component

The CssSelector Component converts CSS selectors to XPath expressions.

Installation

You can install the component in several different ways:

- Use the official Git repository (https://github.com/symfony/CssSelector);
- Install it via PEAR (*pear.symfony.com/CssSelector*);
- Install it via Composer (*symfony/css-selector* on Packagist).

Usage

Why use CSS selectors?

When you're parsing an HTML or an XML document, by far the most powerful method is XPath.

XPath expressions are incredibly flexible, so there is almost always an XPath expression that will find the element you need. Unfortunately, they can also become very complicated, and the learning curve is steep. Even common operations (such as finding an element with a particular class) can require long and unwieldy expressions.

Many developers – particularly web developers – are more comfortable using CSS selectors to find elements. As well as working in stylesheets, CSS selectors are used in Javascript with the `querySelectorAll` function and in popular Javascript libraries such as jQuery, Prototype and MooTools.

CSS selectors are less powerful than XPath, but far easier to write, read and understand. Since they are less powerful, almost all CSS selectors can be converted to an XPath equivalent. This XPath expression can then be used with other functions and classes that use XPath to find elements in a document.

The `CssSelector` component

The component's only goal is to convert CSS selectors to their XPath equivalents:

```
use Symfony\Component\CssSelector\CssSelector;

print CssSelector::toXPath('div.item > h4 > a');
```

This gives the following output:

```
descendant-or-self::div[contains(concat(' ',normalize-space(@class), ' '), ' item ')]/h4/a
```

You can use this expression with, for instance, **:phpclass:'DOMXPath'** or **:phpclass:'SimpleXMLElement'** to find elements in a document.

Tip: The **:method:'Crawler::filter()<Symfony\\Component\\DomCrawler\\Crawler::filter>'** method uses the `CssSelector` component to find elements based on a CSS selector string. See the The DomCrawler Component for more details.

Limitations of the CssSelector component

Not all CSS selectors can be converted to XPath equivalents.

There are several CSS selectors that only make sense in the context of a web-browser.

- link-state selectors: `:link`, `:visited`, `:target`
- selectors based on user action: `:hover`, `:focus`, `:active`
- UI-state selectors: `:enabled`, `:disabled`, `:indeterminate` (however, `:checked` and `:unchecked` are available)

Pseudo-elements (`:before`, `:after`, `:first-line`, `:first-letter`) are not supported because they select portions of text rather than elements.

Several pseudo-classes are not yet supported:

- `:lang(language)`
- `root`
- `*:first-of-type`, `*:last-of-type`, `*:nth-of-type`, `*:nth-last-of-type`, `*:only-of-type`. (These work with an element name (e.g. `li:first-of-type`) but not with `*`.

4.1.4 The DomCrawler Component

The DomCrawler Component eases DOM navigation for HTML and XML documents.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/DomCrawler);
- Install it via PEAR (*pear.symfony.com/DomCrawler*);
- Install it via Composer (*symfony/dom-crawler* on Packagist).

Usage

The `Symfony\Component\DomCrawler\Crawler` class provides methods to query and manipulate HTML and XML documents.

An instance of the Crawler represents a set (**:phpclass:'SplObjectStorage'**) of **:phpclass:'DOMElement'** objects, which are basically nodes that you can traverse easily:

```
use Symfony\Component\DomCrawler\Crawler;

$html = <<<'HTML'
<html>
    <body>
        <p class="message">Hello World!</p>
        <p>Hello Crawler!</p>
    </body>
</html>
HTML;

$crawler = new Crawler($html);

foreach ($crawler as $domElement) {
    print $domElement->nodeName;
}
```

Specialized `Symfony\Component\DomCrawler\Link` and `Symfony\Component\DomCrawler\Form` classes are useful for interacting with html links and forms as you traverse through the HTML tree.

Node Filtering

Using XPath expressions is really easy:

```
$crawler = $crawler->filterXPath('descendant-or-self::body/p');
```

Tip: `DOMXPath::query` is used internally to actually perform an XPath query.

Filtering is even easier if you have the `CssSelector` Component installed. This allows you to use jQuery-like selectors to traverse:

```
$crawler = $crawler->filter('body > p');
```

Anonymous function can be used to filter with more complex criteria:

```
$crawler = $crawler->filter('body > p')->reduce(function ($node, $i) {
    // filter even nodes
    return ($i % 2) == 0;
});
```

To remove a node the anonymous function must return false.

Note: All filter methods return a new `Symfony\Component\DomCrawler\Crawler` instance with filtered content.

Node Traversing

Access node by its position on the list:

```
$crawler->filter('body > p')->eq(0);
```

Get the first or last node of the current selection:

```
$crawler->filter('body > p')->first();
$crawler->filter('body > p')->last();
```

Get the nodes of the same level as the current selection:

```
$crawler->filter('body > p')->siblings();
```

Get the same level nodes after or before the current selection:

```
$crawler->filter('body > p')->nextAll();
$crawler->filter('body > p')->previousAll();
```

Get all the child or parent nodes:

```
$crawler->filter('body')->children();
$crawler->filter('body > p')->parents();
```

Note: All the traversal methods return a new `Symfony\Component\DomCrawler\Crawler` instance.

Accessing Node Values

Access the value of the first node of the current selection:

```
$message = $crawler->filterXPath('//body/p')->text();
```

Access the attribute value of the first node of the current selection:

```
$class = $crawler->filterXPath('//body/p')->attr('class');
```

Extract attribute and/or node values from the list of nodes:

```
$attributes = $crawler->filterXpath('//body/p')->extract(array('_text', 'class'));
```

Note: Special attribute `_text` represents a node value.

Call an anonymous function on each node of the list:

```
$nodeValues = $crawler->filter('p')->each(function ($node, $i) {
    return $node->nodeValue;
});
```

The anonymous function receives the position and the node as arguments. The result is an array of values returned by the anonymous function calls.

Adding the Content

The crawler supports multiple ways of adding the content:

```
$crawler = new Crawler('<html><body /></html>');

$crawler->addHtmlContent('<html><body /></html>');
$crawler->addXmlContent('<root><node /></root>');

$crawler->addContent('<html><body /></html>');
$crawler->addContent('<root><node /></root>', 'text/xml');

$crawler->add('<html><body /></html>');
$crawler->add('<root><node /></root>');
```

As the Crawler's implementation is based on the DOM extension, it is also able to interact with native **:phpclass:'DOMDocument'**, **:phpclass:'DOMNodeList'** and **:phpclass:'DOMNode'** objects:

```
$document = new \DOMDocument();
$document->loadXml('<root><node /><node /></root>');
$nodeList = $document->getElementsByTagName('node');
$node = $document->getElementsByTagName('node')->item(0);

$crawler->addDocument($document);
$crawler->addNodeList($nodeList);
$crawler->addNodes(array($node));
$crawler->addNode($node);
$crawler->add($document);
```

Form and Link support

Special treatment is given to links and forms inside the DOM tree.

Links To find a link by name (or a clickable image by its `alt` attribute), use the `selectLink` method on an existing crawler. This returns a Crawler instance with just the selected link(s). Calling `link()` gives us a special `Symfony\Component\DomCrawler\Link` object:

```
$linksCrawler = $crawler->selectLink('Go elsewhere...');
$link = $linksCrawler->link();

// or do this all at once
$link = $crawler->selectLink('Go elsewhere...')->link();
```

The `Symfony\Component\DomCrawler\Link` object has several useful methods to get more information about the selected link itself:

```
// return the raw href value
$href = $link->getRawUri();

// return the proper URI that can be used to make another request
$uri = $link->getUri();
```

The `getUri()` is especially useful as it cleans the `href` value and transforms it into how it should really be processed. For example, for a link with `href="#foo"`, this would return the full URI of the current page suffixed with `#foo`. The return from `getUri()` is always a full URI that you can act on.

Forms Special treatment is also given to forms. A `selectButton()` method is available on the Crawler which returns another Crawler that matches a button (`input[type=submit]`, `input[type=image]`, or a `button`) with the given text. This method is especially useful because you can use it to return a `Symfony\Component\DomCrawler\Form` object that represents the form that the button lives in:

```
$form = $crawler->selectButton('validate')->form();

// or "fill" the form fields with data
$form = $crawler->selectButton('validate')->form(array(
    'name' => 'Ryan',
));
```

The `Symfony\Component\DomCrawler\Form` object has lots of very useful methods for working with forms:

$uri = $form->getUri();

$method = $form->getMethod();

The **:method:'Symfony\\Component\\DomCrawler\\Form::getUri'** method does more than just return the `action` attribute of the form. If the form method is GET, then it mimics the browser's behavior and returns the `action` attribute followed by a query string of all of the form's values.

You can virtually set and get values on the form:

```
// set values on the form internally
$form->setValues(array(
    'registration[username]' => 'symfonyfan',
    'registration[terms]'    => 1,
));

// get back an array of values - in the "flat" array like above
$values = $form->getValues();

// returns the values like PHP would see them, where "registration" is its own array
$values = $form->getPhpValues();
```

To work with multi-dimensional fields:

```
<form>
    <input name="multi[]" />
    <input name="multi[]" />
    <input name="multi[dimensional]" />
</form>
```

You must specify the fully qualified name of the field:

```
// Set a single field
$form->setValue('multi[0]', 'value');

// Set multiple fields at once
$form->setValue('multi', array(
    1             => 'value',
    'dimensional' => 'an other value'
));
```

This is great, but it gets better! The `Form` object allows you to interact with your form like a browser, selecting radio values, ticking checkboxes, and uploading files:

```
$form['registration[username]']->setValue('symfonyfan');

// check or uncheck a checkbox
$form['registration[terms]']->tick();
$form['registration[terms]']->untick();

// select an option
$form['registration[birthday][year]']->select(1984);

// select many options from a "multiple" select or checkboxes
$form['registration[interests]']->select(array('symfony', 'cookies'));

// even fake a file upload
$form['registration[photo]']->upload('/path/to/lucas.jpg');
```

What's the point of doing all of this? If you're testing internally, you can grab the information off of your form as if it had just been submitted by using the PHP values:

```
$values = $form->getPhpValues();
$files = $form->getPhpFiles();
```

If you're using an external HTTP client, you can use the form to grab all of the information you need to create a POST request for the form:

```
$uri = $form->getUri();
$method = $form->getMethod();
$values = $form->getValues();
$files = $form->getFiles();

// now use some HTTP client and post using this information
```

One great example of an integrated system that uses all of this is Goutte. Goutte understands the Symfony Crawler object and can use it to submit forms directly:

```
use Goutte\Client;

// make a real request to an external site
$client = new Client();
$crawler = $client->request('GET', 'https://github.com/login');

// select the form and fill in some values
$form = $crawler->selectButton('Log in')->form();
$form['login'] = 'symfonyfan';
$form['password'] = 'anypass';

// submit that form
$crawler = $client->submit($form);
```

4.1.5 The Finder Component

The Finder Component finds files and directories via an intuitive fluent interface.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/Finder);
- Install it via PEAR (*pear.symfony.com/Finder*);
- Install it via Composer (*symfony/finder* on Packagist).

Usage

The `Symfony\Component\Finder\Finder` class finds files and/or directories:

```
use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->files()->in(__DIR__);

foreach ($finder as $file) {
    // Print the absolute path
    print $file->getRealpath()."\n";
```

```
    // Print the relative path to the file, omitting the filename
    print $file->getRelativePath()."\n";
    // Print the relative path to the file
    print $file->getRelativePathname()."\n";
}
```

The `$file` is an instance of `Symfony\Component\Finder\SplFileInfo` which extends **:phpclass:‘SplFileInfo‘** to provide methods to work with relative paths.

The above code prints the names of all the files in the current directory recursively. The Finder class uses a fluent interface, so all methods return the Finder instance.

Tip: A Finder instance is a PHP Iterator. So, instead of iterating over the Finder with `foreach`, you can also convert it to an array with the **:phpfunction:‘iterator_to_array‘** method, or get the number of items with **:phpfunction:‘iterator_count‘**.

Criteria

Location

The location is the only mandatory criteria. It tells the finder which directory to use for the search:

```
$finder->in(__DIR__);
```

Search in several locations by chaining calls to **:method:‘Symfony\\Component\\Finder\\Finder::in‘**:

```
$finder->files()->in(__DIR__)->in('/elsewhere');
```

Exclude directories from matching with the **:method:‘Symfony\\Component\\Finder\\Finder::exclude‘** method:

```
$finder->in(__DIR__)->exclude('ruby');
```

As the Finder uses PHP iterators, you can pass any URL with a supported protocol:

```
$finder->in('ftp://example.com/pub/');
```

And it also works with user-defined streams:

```
use Symfony\Component\Finder\Finder;

$s3 = new \Zend_Service_Amazon_S3($key, $secret);
$s3->registerStreamWrapper("s3");

$finder = new Finder();
$finder->name('photos*')->size('< 100K')->date('since 1 hour ago');
foreach ($finder->in('s3://bucket-name') as $file) {
    // do something

    print $file->getFilename()."\n";
}
```

Note: Read the Streams documentation to learn how to create your own streams.

Files or Directories

By default, the Finder returns files and directories; but the **:method:‘Symfony\\Component\\Finder\\Finder::files‘** and **:method:‘Symfony\\Component\\Finder\\Finder::directories‘** methods control that:

```
$finder->files();

$finder->directories();
```

If you want to follow links, use the `followLinks()` method:

```
$finder->files()->followLinks();
```

By default, the iterator ignores popular VCS files. This can be changed with the `ignoreVCS()` method:

```
$finder->ignoreVCS(false);
```

Sorting

Sort the result by name or by type (directories first, then files):

```
$finder->sortByName();

$finder->sortByType();
```

Note: Notice that the `sort*` methods need to get all matching elements to do their jobs. For large iterators, it is slow.

You can also define your own sorting algorithm with `sort()` method:

```
$sort = function (\SplFileInfo $a, \SplFileInfo $b)
{
    return strcmp($a->getRealpath(), $b->getRealpath());
};

$finder->sort($sort);
```

File Name

Restrict files by name with the **:method:‘Symfony\\Component\\Finder\\Finder::name‘** method:

```
$finder->files()->name('*.php');
```

The `name()` method accepts globs, strings, or regexes:

```
$finder->files()->name('/\.php$/');
```

The `notNames()` method excludes files matching a pattern:

```
$finder->files()->notName('*.rb');
```

File Size

Restrict files by size with the **:method:‘Symfony\\Component\\Finder\\Finder::size‘** method:

```
$finder->files()->size('< 1.5K');
```

Restrict by a size range by chaining calls:

```
$finder->files()->size('>= 1K')->size('<= 2K');
```

The comparison operator can be any of the following: >, >=, <, '<=', '=='.

The target value may use magnitudes of kilobytes (k, ki), megabytes (m, mi), or gigabytes (g, gi). Those suffixed with an i use the appropriate 2**n version in accordance with the IEC standard.

File Date

Restrict files by last modified dates with the **:method:'Symfony\\Component\\Finder\\Finder::date'** method:

```
$finder->date('since yesterday');
```

The comparison operator can be any of the following: >, >=, <, '<=', '=='. You can also use since or after as an alias for >, and until or before as an alias for <.

The target value can be any date supported by the strtotime function.

Directory Depth

By default, the Finder recursively traverse directories. Restrict the depth of traversing with **:method:'Symfony\\Component\\Finder\\Finder::depth'**:

```
$finder->depth('== 0');
$finder->depth('< 3');
```

Custom Filtering

To restrict the matching file with your own strategy, use **:method:'Symfony\\Component\\Finder\\Finder::filter'**:

```
$filter = function (\SplFileInfo $file)
{
    if (strlen($file) > 10) {
        return false;
    }
};

$finder->files()->filter($filter);
```

The filter() method takes a Closure as an argument. For each matching file, it is called with the file as a Symfony\Component\Finder\SplFileInfo instance. The file is excluded from the result set if the Closure returns false.

4.1.6 The HttpFoundation Component

The HttpFoundation Component defines an object-oriented layer for the HTTP specification.

In PHP, the request is represented by some global variables ($_GET, $_POST, $_FILE, $_COOKIE, $_SESSION...) and the response is generated by some functions (echo, header, setcookie, ...).

The Symfony2 HttpFoundation component replaces these default PHP global variables and functions by an Object-Oriented layer.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/HttpFoundation);
- Install it via PEAR (*pear.symfony.com/HttpFoundation*);
- Install it via Composer (*symfony/http-foundation* on Packagist).

Request

The most common way to create request is to base it on the current PHP global variables with **:method:‘Symfony\\Component\\HttpFoundation\\Request::createFromGlobals‘**:

```
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
```

which is almost equivalent to the more verbose, but also more flexible, **:method:‘Symfony\\Component\\HttpFoundation\\Request::__construct‘** call:

```
$request = new Request($_GET, $_POST, array(), $_COOKIE, $_FILES, $_SERVER);
```

Accessing Request Data

A Request object holds information about the client request. This information can be accessed via several public properties:

- `request`: equivalent of `$_POST`;
- `query`: equivalent of `$_GET` (`$request->query->get('name')`);
- `cookies`: equivalent of `$_COOKIE`;
- `attributes`: no equivalent - used by your app to store other data (see *below*)
- `files`: equivalent of `$_FILE`;
- `server`: equivalent of `$_SERVER`;
- `headers`: mostly equivalent to a sub-set of `$_SERVER` (`$request->headers->get('Content-Type')`).

Each property is a `Symfony\Component\HttpFoundation\ParameterBag` instance (or a sub-class of), which is a data holder class:

- `request`: `Symfony\Component\HttpFoundation\ParameterBag`;
- `query`: `Symfony\Component\HttpFoundation\ParameterBag`;
- `cookies`: `Symfony\Component\HttpFoundation\ParameterBag`;
- `attributes`: `Symfony\Component\HttpFoundation\ParameterBag`;
- `files`: `Symfony\Component\HttpFoundation\FileBag`;
- `server`: `Symfony\Component\HttpFoundation\ServerBag`;

- `headers`: `Symfony\Component\HttpFoundation\HeaderBag`.

All `Symfony\Component\HttpFoundation\ParameterBag` instances have methods to retrieve and update its data:

- **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::all'**: Returns the parameters;
- **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::keys'**: Returns the parameter keys;
- **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::replace'**: Replaces the current parameters by a new set;
- **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::add'**: Adds parameters;
- **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::get'**: Returns a parameter by name;
- **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::set'**: Sets a parameter by name;
- **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::has'**: Returns true if the parameter is defined;
- **:method:'Symfony\\Component\\HttpFoundation\\ParameterBag::remove'**: Removes a parameter.

The `Symfony\Component\HttpFoundation\ParameterBag` instance also has some methods to filter the input values:

- **:method:'Symfony\\Component\\HttpFoundation\\Request::getAlpha'**: Returns the alphabetic characters of the parameter value;
- **:method:'Symfony\\Component\\HttpFoundation\\Request::getAlnum'**: Returns the alphabetic characters and digits of the parameter value;
- **:method:'Symfony\\Component\\HttpFoundation\\Request::getDigits'**: Returns the digits of the parameter value;
- **:method:'Symfony\\Component\\HttpFoundation\\Request::getInt'**: Returns the parameter value converted to integer;
- **:method:'Symfony\\Component\\HttpFoundation\\Request::filter'**: Filters the parameter by using the PHP `filter_var()` function.

All getters takes up to three arguments: the first one is the parameter name and the second one is the default value to return if the parameter does not exist:

```
// the query string is '?foo=bar'

$request->query->get('foo');
// returns bar

$request->query->get('bar');
// returns null

$request->query->get('bar', 'bar');
// returns 'bar'
```

When PHP imports the request query, it handles request parameters like `foo[bar]=bar` in a special way as it creates an array. So you can get the `foo` parameter and you will get back an array with a `bar` element. But sometimes, you might want to get the value for the "original" parameter name: `foo[bar]`. This is possible with all the *ParameterBag* getters like **:method:'Symfony\\Component\\HttpFoundation\\Request::get'** via the third argument:

```
// the query string is '?foo[bar]=bar'

$request->query->get('foo');
// returns array('bar' => 'bar')
```

```
$request->query->get('foo[bar]');
// returns null

$request->query->get('foo[bar]', null, true);
// returns 'bar'
```

Last, but not the least, you can also store additional data in the request, thanks to the `attributes` public property, which is also an instance of `Symfony\Component\HttpFoundation\ParameterBag`. This is mostly used to attach information that belongs to the Request and that needs to be accessed from many different points in your application. For information on how this is used in the Symfony2 framework, see *read more*.

Identifying a Request

In your application, you need a way to identify a request; most of the time, this is done via the "path info" of the request, which can be accessed via the **:method:'Symfony\\Component\\HttpFoundation\\Request::getPathInfo'** method:

```
// for a request to http://example.com/blog/index.php/post/hello-world
// the path info is "/post/hello-world"
$request->getPathInfo();
```

Simulating a Request

Instead of creating a Request based on the PHP globals, you can also simulate a Request:

```
$request = Request::create('/hello-world', 'GET', array('name' => 'Fabien'));
```

The **:method:'Symfony\\Component\\HttpFoundation\\Request::create'** method creates a request based on a path info, a method and some parameters (the query parameters or the request ones depending on the HTTP method); and of course, you an also override all other variables as well (by default, Symfony creates sensible defaults for all the PHP global variables).

Based on such a request, you can override the PHP global variables via **:method:'Symfony\\Component\\HttpFoundation\\Request::overrideGlobals'**:

```
$request->overrideGlobals();
```

Tip: You can also duplicate an existing query via **:method:'Symfony\\Component\\HttpFoundation\\Request::duplicate'** or change a bunch of parameters with a single call to **:method:'Symfony\\Component\\HttpFoundation\\Request::initialize'**.

Accessing the Session

If you have a session attached to the Request, you can access it via the **:method:'Symfony\\Component\\HttpFoundation\\Request::getSession'** method; the **:method:'Symfony\\Component\\HttpFoundation\\Request::hasPreviousSession'** method tells you if the request contains a Session which was started in one of the previous requests.

Accessing other Data

The Request class has many other methods that you can use to access the request information. Have a look at the API for more information about them.

Response

A `Symfony\Component\HttpFoundation\Response` object holds all the information that needs to be sent back to the client from a given request. The constructor takes up to three arguments: the response content, the status code, and an array of HTTP headers:

```
use Symfony\Component\HttpFoundation\Response;

$response = new Response('Content', 200, array('content-type' => 'text/html'));
```

These information can also be manipulated after the Response object creation:

```
$response->setContent('Hello World');

// the headers public attribute is a ResponseHeaderBag
$response->headers->set('Content-Type', 'text/plain');

$response->setStatusCode(404);
```

When setting the `Content-Type` of the Response, you can set the charset, but it is better to set it via the **:method:'Symfony\\Component\\HttpFoundation\\Response::setCharset'** method:

```
$response->setCharset('ISO-8859-1');
```

Note that by default, Symfony assumes that your Responses are encoded in UTF-8.

Sending the Response

Before sending the Response, you can ensure that it is compliant with the HTTP specification by calling the **:method:'Symfony\\Component\\HttpFoundation\\Response::prepare'** method:

```
$response->prepare($request);
```

Sending the response to the client is then as simple as calling **:method:'Symfony\\Component\\HttpFoundation\\Response::send'**:

```
$response->send();
```

Setting Cookies

The response cookies can be manipulated though the `headers` public attribute:

```
use Symfony\Component\HttpFoundation\Cookie;

$response->headers->setCookie(new Cookie('foo', 'bar'));
```

The **:method:'Symfony\\Component\\HttpFoundation\\ResponseHeaderBag::setCookie'** method takes an instance of `Symfony\Component\HttpFoundation\Cookie` as an argument.

You can clear a cookie via the **:method:'Symfony\\Component\\HttpFoundation\\Response::clearCookie'** method.

Managing the HTTP Cache

The `Symfony\Component\HttpFoundation\Response` class has a rich set of methods to manipulate the HTTP headers related to the cache:

- **:method:'Symfony\\Component\\HttpFoundation\\Response::setPublic'**;

- **:method:'Symfony\\Component\\HttpFoundation\\Response::setPrivate'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::expire'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::setExpires'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::setMaxAge'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::setSharedMaxAge'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::setTtl'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::setClientTtl'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::setLastModified'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::setEtag'**;
- **:method:'Symfony\\Component\\HttpFoundation\\Response::setVary'**;

The **:method:'Symfony\\Component\\HttpFoundation\\Response::setCache'** method can be used to set the most commonly used cache information in one method call:

```
$response->setCache(array(
    'etag'          => 'abcdef',
    'last_modified' => new \DateTime(),
    'max_age'       => 600,
    's_maxage'      => 600,
    'private'       => false,
    'public'        => true,
));
```

To check if the Response validators (`ETag`, `Last-Modified`) match a conditional value specified in the client Request, use the **:method:'Symfony\\Component\\HttpFoundation\\Response::isNotModified'** method:

```
if ($response->isNotModified($request)) {
    $response->send();
}
```

If the Response is not modified, it sets the status code to 304 and remove the actual response content.

Redirecting the User

To redirect the client to another URL, you can use the `Symfony\Component\HttpFoundation\RedirectResponse` class:

```
use Symfony\Component\HttpFoundation\RedirectResponse;

$response = new RedirectResponse('http://example.com/');
```

Streaming a Response

New in version 2.1: Support for streamed responses was added in Symfony 2.1.

The `Symfony\Component\HttpFoundation\StreamedResponse` class allows you to stream the Response back to the client. The response content is represented by a PHP callable instead of a string:

```
use Symfony\Component\HttpFoundation\StreamedResponse;

$response = new StreamedResponse();
```

```
$response->setCallback(function () {
    echo 'Hello World';
    flush();
    sleep(2);
    echo 'Hello World';
    flush();
});
$response->send();
```

Downloading Files

New in version 2.1: The `makeDisposition` method was added in Symfony 2.1.

When uploading a file, you must add a `Content-Disposition` header to your response. While creating this header for basic file downloads is easy, using non-ASCII filenames is more involving. The **:method:‘:Symfony\\Component\\HttpFoundation\\Response:makeDisposition‘** abstracts the hard work behind a simple API:

```
use Symfony\\Component\\HttpFoundation\\ResponseHeaderBag;

$d = $response->headers->makeDisposition(ResponseHeaderBag::DISPOSITION_ATTACHMENT, 'foo.pdf');

$response->headers->set('Content-Disposition', $d);
```

Session

TBD – This part has not been written yet as it will probably be refactored soon in Symfony 2.1.

4.1.7 The Locale Component

> Locale component provides fallback code to handle cases when the `intl` extension is missing. Additionally it extends the implementation of a native **:phpclass:‘Locale‘** class with several handy methods.

Replacement for the following functions and classes is provided:

- **:phpfunction:‘intl_is_failure()‘**
- **:phpfunction:‘intl_get_error_code()‘**
- **:phpfunction:‘intl_get_error_message()‘**
- **:phpclass:‘Collator‘**
- **:phpclass:‘IntlDateFormatter‘**
- **:phpclass:‘Locale‘**
- **:phpclass:‘NumberFormatter‘**

Note: Stub implementation only supports the `en` locale.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/Locale);
- Install it via PEAR (*pear.symfony.com/Locale*);
- Install it via Composer (*symfony/locale* on Packagist).

Usage

Taking advantage of the fallback code includes requiring function stubs and adding class stubs to the autoloader.

When using the ClassLoader component following code is sufficient to supplement missing `intl` extension:

```
if (!function_exists('intl_get_error_code')) {
    require __DIR__.'/path/to/src/Symfony/Component/Locale/Resources/stubs/functions.php';

    $loader->registerPrefixFallbacks(array(__DIR__.'/path/to/src/Symfony/Component/Locale/Resources/s
}
```

`Symfony\Component\Locale\Locale` class enriches native **:phpclass:‘Locale‘** class with additional features:

```
use Symfony\Component\Locale\Locale;

// Get the country names for a locale or get all country codes
$countries = Locale::getDisplayCountries('pl');
$countryCodes = Locale::getCountries();

// Get the language names for a locale or get all language codes
$languages = Locale::getDisplayLanguages('fr');
$languageCodes = Locale::getLanguages();

// Get the locale names for a given code or get all locale codes
$locales = Locale::getDisplayLocales('en');
$localeCodes = Locale::getLocales();

// Get ICU versions
$icuVersion = Locale::getIcuVersion();
$icuDataVersion = Locale::getIcuDataVersion();
```

4.1.8 The Process Component

The Process Component executes commands in sub-processes.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/Process);
- Install it via PEAR (*pear.symfony.com/Process*);
- Install it via Composer (*symfony/process* on Packagist).

Usage

The `Symfony\Component\Process\Process` class allows you to execute a command in a sub-process:

```
use Symfony\Component\Process\Process;

$process = new Process('ls -lsa');
$process->setTimeout(3600);
$process->run();
if (!$process->isSuccessful()) {
    throw new RuntimeException($process->getErrorOutput());
}

print $process->getOutput();
```

The **:method:'Symfony\\Component\\Process\\Process::run'** method takes care of the subtle differences between the different platforms when executing the command.

When executing a long running command (like rsync-ing files to a remote server), you can give feedback to the end user in real-time by passing an anonymous function to the **:method:'Symfony\\Component\\Process\\Process::run'** method:

```
use Symfony\Component\Process\Process;

$process = new Process('ls -lsa');
$process->run(function ($type, $buffer) {
    if ('err' === $type) {
        echo 'ERR > '.$buffer;
    } else {
        echo 'OUT > '.$buffer;
    }
});
```

If you want to execute some PHP code in isolation, use the `PhpProcess` instead:

```
use Symfony\Component\Process\PhpProcess;

$process = new PhpProcess(<<<EOF
    <?php echo 'Hello World'; ?>
EOF);
$process->run();
```

New in version 2.1: The `ProcessBuilder` class has been as of 2.1.

To make your code work better on all platforms, you might want to use the `Symfony\Component\Process\ProcessBuilder` class instead:

```
use Symfony\Component\Process\ProcessBuilder;

$builder = new ProcessBuilder(array('ls', '-lsa'));
$builder->getProcess()->run();
```

4.1.9 The Routing Component

The Routing Component maps an HTTP request to a set of configuration variables.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/Routing);

- Install it via PEAR (*pear.symfony.com/Routing*);
- Install it via Composer (*symfony/routing* on Packagist)

Usage

In order to set up a basic routing system you need three parts:

- A `Symfony\Component\Routing\RouteCollection`, which contains the route definitions (instances of the class `Symfony\Component\Routing\Route`)
- A `Symfony\Component\Routing\RequestContext`, which has information about the request
- A `Symfony\Component\Routing\Matcher\UrlMatcher`, which performs the mapping of the request to a single route

Let's see a quick example. Notice that this assumes that you've already configured your autoloader to load the Routing component:

```
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$routes = new RouteCollection();
$routes->add('route_name', new Route('/foo', array('controller' => 'MyController')));

$context = new RequestContext($_SERVER['REQUEST_URI']);

$matcher = new UrlMatcher($routes, $context);

$parameters = $matcher->match( '/foo' );
// array('controller' => 'MyController', '_route' => 'route_name')
```

Note: Be careful when using `$_SERVER['REQUEST_URI']`, as it may include any query parameters on the URL, which will cause problems with route matching. An easy way to solve this is to use the HTTPFoundation component as explained *below*.

You can add as many routes as you like to a `Symfony\Component\Routing\RouteCollection`.

The **:method:'RouteCollection::add()<Symfony\\Component\\Routing\\RouteCollection::add>'** method takes two arguments. The first is the name of the route, The second is a `Symfony\Component\Routing\Route` object, which expects a URL path and some array of custom variables in its constructor. This array of custom variables can be *anything* that's significant to your application, and is returned when that route is matched.

If no matching route can be found a `Symfony\Component\Routing\Exception\ResourceNotFoundException` will be thrown.

In addition to your array of custom variables, a `_route` key is added, which holds the name of the matched route.

Defining routes

A full route definition can contain up to four parts:

1. The URL pattern route. This is matched against the URL passed to the *RequestContext*, and can contain named wildcard placeholders (e.g. `{placeholders}`) to match dynamic parts in the URL.

2. An array of default values. This contains an array of arbitrary values that will be returned when the request matches the route.

3. An array of requirements. These define constraints for the values of the placeholders as regular expressions.

4. An array of options. These contain internal settings for the route and are the least commonly needed.

Take the following route, which combines several of these ideas:

```
$route = new Route(
    '/archive/{month}', // path
    array('controller' => 'showArchive'), // default values
    array('month' => '[0-9]{4}-[0-9]{2}'), // requirements
    array() // options
);

// ...

$parameters = $matcher->match('/archive/2012-01');
// array('controller' => 'showArchive', 'month' => 2012-01'', '_route' => '...')

$parameters = $matcher->match('/archive/foo');
// throws ResourceNotFoundException
```

In this case, the route is matched by `/archive/2012/01`, because the `{month}` wildcard matches the regular expression wildcard given. However, `/archive/foo` does *not* match, because "foo" fails the month wildcard.

Besides the regular expression constraints there are two special requirements you can define:

- `_method` enforces a certain HTTP request method (`HEAD`, `GET`, `POST`, ...)
- `_scheme` enforces a certain HTTP scheme (`http`, `https`)

For example, the following route would only accept requests to /foo with the POST method and a secure connection:

```
$route = new Route('/foo', array('_method' => 'post', '_scheme' => 'https' ));
```

Tip: If you want to match all urls which start with a certain path and end in an arbitrary suffix you can use the following route definition:

```
$route = new Route('/start/{suffix}', array('suffix' => ''), array('suffix' => '.*'));
```

Using Prefixes

You can add routes or other instances of `Symfony\Component\Routing\RouteCollection` to *another* collection. This way you can build a tree of routes. Additionally you can define a prefix, default requirements and default options to all routes of a subtree:

```
$rootCollection = new RouteCollection();

$subCollection = new RouteCollection();
$subCollection->add( /*...*/ );
$subCollection->add( /*...*/ );

$rootCollection->addCollection($subCollection, '/prefix', array('_scheme' => 'https'));
```

Set the Request Parameters

The `Symfony\Component\Routing\RequestContext` provides information about the current request. You can define all parameters of an HTTP request with this class via its constructor:

```
public function __construct($baseUrl = '', $method = 'GET', $host = 'localhost', $scheme = 'http', $h
```

Normally you can pass the values from the `$_SERVER` variable to populate the `Symfony\Component\Routing\RequestContext`. But If you use the HttpFoundation component, you can use its `Symfony\Component\HttpFoundation\Request` class to feed the `Symfony\Component\Routing\RequestContext` in a shortcut:

```
use Symfony\Component\HttpFoundation\Request;

$context = new RequestContext();
$context->fromRequest(Request::createFromGlobals());
```

Generate a URL

While the `Symfony\Component\Routing\Matcher\UrlMatcher` tries to find a route that fits the given request you can also build a URL from a certain route:

```
use Symfony\Component\Routing\Generator\UrlGenerator;

$routes = new RouteCollection();
$routes->add('show_post', new Route('/show/{slug}'));

$context = new RequestContext($_SERVER['REQUEST_URI']);

$generator = new UrlGenerator($routes, $context);

$url = $generator->generate('show_post', array(
    'slug' => 'my-blog-post'
));
// /show/my-blog-post
```

Note: If you have defined the `_scheme` requirement, an absolute URL is generated if the scheme of the current `Symfony\Component\Routing\RequestContext` does not match the requirement.

Load Routes from a File

You've already seen how you can easily add routes to a collection right inside PHP. But you can also load routes from a number of different files.

The Routing component comes with a number of loader classes, each giving you the ability to load a collection of route definitions from an external file of some format. Each loader expects a `Symfony\Component\Config\FileLocator` instance as the constructor argument. You can use the `Symfony\Component\Config\FileLocator` to define an array of paths in which the loader will look for the requested files. If the file is found, the loader returns a `Symfony\Component\Routing\RouteCollection`.

If you're using the `YamlFileLoader`, then route definitions look like this:

```
# routes.yml
route1:
    pattern: /foo
```

```
    defaults: { controller: 'MyController::fooAction' }

route2:
    pattern: /foo/bar
    defaults: { controller: 'MyController::foobarAction' }
```

To load this file, you can use the following code. This assumes that your `routes.yml` file is in the same directory as the below code:

```
use Symfony\Component\Config\FileLocator;
use Symfony\Component\Routing\Loader\YamlFileLoader;

// look inside *this* directory
$locator = new FileLocator(array(__DIR__));
$loader = new YamlFileLoader($locator);
$collection = $loader->load('routes.yml');
```

Besides `Symfony\Component\Routing\Loader\YamlFileLoader` there are two other loaders that work the same way:

- `Symfony\Component\Routing\Loader\XmlFileLoader`
- `Symfony\Component\Routing\Loader\PhpFileLoader`

If you use the `Symfony\Component\Routing\Loader\PhpFileLoader` you have to provide the name of a php file which returns a `Symfony\Component\Routing\RouteCollection`:

```
// RouteProvider.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('route_name', new Route('/foo', array('controller' => 'ExampleController')));
// ...

return $collection;
```

Routes as Closures There is also the `Symfony\Component\Routing\Loader\ClosureLoader`, which calls a closure and uses the result as a `Symfony\Component\Routing\RouteCollection`:

```
use Symfony\Component\Routing\Loader\ClosureLoader;

$closure = function() {
    return new RouteCollection();
};

$loader = new ClosureLoader();
$collection = $loader->load($closure);
```

Routes as Annotations Last but not least there are `Symfony\Component\Routing\Loader\AnnotationDirectoryLoad` and `Symfony\Component\Routing\Loader\AnnotationFileLoader` to load route definitions from class annotations. The specific details are left out here.

The all-in-one Router

The `Symfony\Component\Routing\Router` class is a all-in-one package to quickly use the Routing component. The constructor expects a loader instance, a path to the main route definition and some other settings:

```
public function __construct(LoaderInterface $loader, $resource, array $options = array(), RequestCont
```

With the `cache_dir` option you can enable route caching (if you provide a path) or disable caching (if it's set to `null`). The caching is done automatically in the background if you want to use it. A basic example of the `Symfony\Component\Routing\Router` class would look like:

```
$locator = new FileLocator(array(__DIR__));
$requestContext = new RequestContext($_SERVER['REQUEST_URI']);

$router = new Router(
    new YamlFileLoader($locator),
    "routes.yml",
    array('cache_dir' => __DIR__.'/cache'),
    $requestContext,
);
$router->match('/foo/bar');
```

Note: If you use caching, the Routing component will compile new classes which are saved in the `cache_dir`. This means your script must have write permissions for that location.

4.1.10 The YAML Component

The YAML Component loads and dumps YAML files.

What is it?

The Symfony2 YAML Component parses YAML strings to convert them to PHP arrays. It is also able to convert PHP arrays to YAML strings.

YAML, *YAML Ain't Markup Language*, is a human friendly data serialization standard for all programming languages. YAML is a great format for your configuration files. YAML files are as expressive as XML files and as readable as INI files.

The Symfony2 YAML Component implements the YAML 1.2 version of the specification.

Installation

You can install the component in many different ways:

- Use the official Git repository (https://github.com/symfony/Yaml);
- Install it via PEAR (*pear.symfony.com/Yaml*);
- Install it via Composer (*symfony/yaml* on Packagist).

Why?

Fast

One of the goal of Symfony YAML is to find the right balance between speed and features. It supports just the needed feature to handle configuration files.

Real Parser

It sports a real parser and is able to parse a large subset of the YAML specification, for all your configuration needs. It also means that the parser is pretty robust, easy to understand, and simple enough to extend.

Clear error messages

Whenever you have a syntax problem with your YAML files, the library outputs a helpful message with the filename and the line number where the problem occurred. It eases the debugging a lot.

Dump support

It is also able to dump PHP arrays to YAML with object support, and inline level configuration for pretty outputs.

Types Support

It supports most of the YAML built-in types like dates, integers, octals, booleans, and much more...

Full merge key support

Full support for references, aliases, and full merge key. Don't repeat yourself by referencing common configuration bits.

Using the Symfony2 YAML Component

The Symfony2 YAML Component is very simple and consists of two main classes: one parses YAML strings (`Symfony\Component\Yaml\Parser`), and the other dumps a PHP array to a YAML string (`Symfony\Component\Yaml\Dumper`).

On top of these two classes, the `Symfony\Component\Yaml\Yaml` class acts as a thin wrapper that simplifies common uses.

Reading YAML Files

The **:method:'Symfony\\Component\\Yaml\\Parser::parse'** method parses a YAML string and converts it to a PHP array:

```
use Symfony\Component\Yaml\Parser;

$yaml = new Parser();

$value = $yaml->parse(file_get_contents('/path/to/file.yml'));
```

If an error occurs during parsing, the parser throws a `Symfony\Component\Yaml\Exception\ParseException` exception indicating the error type and the line in the original YAML string where the error occurred:

```
use Symfony\Component\Yaml\Exception\ParseException;

try {
    $value = $yaml->parse(file_get_contents('/path/to/file.yml'));
} catch (ParseException $e) {
    printf("Unable to parse the YAML string: %s", $e->getMessage());
}
```

Tip: As the parser is re-entrant, you can use the same parser object to load different YAML strings.

When loading a YAML file, it is sometimes better to use the **:method:'Symfony\\Component\\Yaml\\Yaml::parse'** wrapper method:

```
use Symfony\Component\Yaml\Yaml;

$loader = Yaml::parse('/path/to/file.yml');
```

The **:method:'Symfony\\Component\\Yaml\\Yaml::parse'** static method takes a YAML string or a file containing YAML. Internally, it calls the **:method:'Symfony\\Component\\Yaml\\Parser::parse'** method, but with some added bonuses:

- It executes the YAML file as if it was a PHP file, so that you can embed PHP commands in YAML files;
- When a file cannot be parsed, it automatically adds the file name to the error message, simplifying debugging when your application is loading several YAML files.

Writing YAML Files

The **:method:'Symfony\\Component\\Yaml\\Dumper::dump'** method dumps any PHP array to its YAML representation:

```
use Symfony\Component\Yaml\Dumper;

$array = array('foo' => 'bar', 'bar' => array('foo' => 'bar', 'bar' => 'baz'));

$dumper = new Dumper();

$yaml = $dumper->dump($array);

file_put_contents('/path/to/file.yml', $yaml);
```

Note: Of course, the Symfony2 YAML dumper is not able to dump resources. Also, even if the dumper is able to dump PHP objects, it is considered to be a not supported feature.

If an error occurs during the dump, the parser throws a `Symfony\Component\Yaml\Exception\DumpException` exception.

If you only need to dump one array, you can use the **:method:'Symfony\\Component\\Yaml\\Yaml::dump'** static method shortcut:

```
use Symfony\Component\Yaml\Yaml;

$yaml = Yaml::dump($array, $inline);
```

The YAML format supports two kind of representation for arrays, the expanded one, and the inline one. By default, the dumper uses the inline representation:

```
{ foo: bar, bar: { foo: bar, bar: baz } }
```

The second argument of the **:method:'Symfony\\Component\\Yaml\\Dumper::dump'** method customizes the level at which the output switches from the expanded representation to the inline one:

```
echo $dumper->dump($array, 1);
```

```
foo: bar
bar: { foo: bar, bar: baz }
```

```
echo $dumper->dump($array, 2);
```

```
foo: bar
bar:
    foo: bar
    bar: baz
```

The YAML Format

According to the official YAML website, YAML is "a human friendly data serialization standard for all programming languages".

Even if the YAML format can describe complex nested data structure, this chapter only describes the minimum set of features needed to use YAML as a configuration file format.

YAML is a simple language that describes data. As PHP, it has a syntax for simple types like strings, booleans, floats, or integers. But unlike PHP, it makes a difference between arrays (sequences) and hashes (mappings).

Scalars

The syntax for scalars is similar to the PHP syntax.

Strings

```
A string in YAML
```

```
'A singled-quoted string in YAML'
```

Tip: In a single quoted string, a single quote `'` must be doubled:

```
'A single quote '' in a single-quoted string'
```

```
"A double-quoted string in YAML\n"
```

Quoted styles are useful when a string starts or ends with one or more relevant spaces.

Tip: The double-quoted style provides a way to express arbitrary strings, by using `\` escape sequences. It is very useful when you need to embed a `\n` or a unicode character in a string.

When a string contains line breaks, you can use the literal style, indicated by the pipe (`|`), to indicate that the string will span several lines. In literals, newlines are preserved:

```
|
  \/ /| |\/| |
  / / | |  | |__
```

Alternatively, strings can be written with the folded style, denoted by >, where each line break is replaced by a space:

```
>
  This is a very long sentence
  that spans several lines in the YAML
  but which will be rendered as a string
  without carriage returns.
```

Note: Notice the two spaces before each line in the previous examples. They won't appear in the resulting PHP strings.

Numbers

```
# an integer
12
```

```
# an octal
014
```

```
# an hexadecimal
0xC
```

```
# a float
13.4
```

```
# an exponential number
1.2e+34
```

```
# infinity
.inf
```

Nulls Nulls in YAML can be expressed with `null` or ~.

Booleans Booleans in YAML are expressed with `true` and `false`.

Dates YAML uses the ISO-8601 standard to express dates:

```
2001-12-14t21:59:43.10-05:00
```

```
# simple date
2002-12-14
```

Collections

A YAML file is rarely used to describe a simple scalar. Most of the time, it describes a collection. A collection can be a sequence or a mapping of elements. Both sequences and mappings are converted to PHP arrays.

Sequences use a dash followed by a space (‘‘- ‘‘):

```
- PHP
- Perl
- Python
```

The previous YAML file is equivalent to the following PHP code:

```
array('PHP', 'Perl', 'Python');
```

Mappings use a colon followed by a space (“: “) to mark each key/value pair:

```
PHP: 5.2
MySQL: 5.1
Apache: 2.2.20
```

which is equivalent to this PHP code:

```
array('PHP' => 5.2, 'MySQL' => 5.1, 'Apache' => '2.2.20');
```

Note: In a mapping, a key can be any valid scalar.

The number of spaces between the colon and the value does not matter:

```
PHP:    5.2
MySQL:  5.1
Apache: 2.2.20
```

YAML uses indentation with one or more spaces to describe nested collections:

```
"symfony 1.0":
  PHP:    5.0
  Propel: 1.2
"symfony 1.2":
  PHP:    5.2
  Propel: 1.3
```

The following YAML is equivalent to the following PHP code:

```
array(
  'symfony 1.0' => array(
    'PHP'    => 5.0,
    'Propel' => 1.2,
  ),
  'symfony 1.2' => array(
    'PHP'    => 5.2,
    'Propel' => 1.3,
  ),
);
```

There is one important thing you need to remember when using indentation in a YAML file: *Indentation must be done with one or more spaces, but never with tabulations*.

You can nest sequences and mappings as you like:

```
'Chapter 1':
  - Introduction
  - Event Types
'Chapter 2':
  - Introduction
  - Helpers
```

YAML can also use flow styles for collections, using explicit indicators rather than indentation to denote scope.

A sequence can be written as a comma separated list within square brackets (`[]`):

```
[PHP, Perl, Python]
```

A mapping can be written as a comma separated list of key/values within curly braces (*{}*):

```
{ PHP: 5.2, MySQL: 5.1, Apache: 2.2.20 }
```

You can mix and match styles to achieve a better readability:

```
'Chapter 1': [Introduction, Event Types]
'Chapter 2': [Introduction, Helpers]
```

```
"symfony 1.0": { PHP: 5.0, Propel: 1.2 }
"symfony 1.2": { PHP: 5.2, Propel: 1.3 }
```

Comments

Comments can be added in YAML by prefixing them with a hash mark (#):

```
# Comment on a line
"symfony 1.0": { PHP: 5.0, Propel: 1.2 } # Comment at the end of a line
"symfony 1.2": { PHP: 5.2, Propel: 1.3 }
```

Note: Comments are simply ignored by the YAML parser and do not need to be indented according to the current level of nesting in a collection.

Read the Components documentation.

CHAPTER 5

Reference Documents

Get answers quickly with reference documents:

5.1 Reference Documents

5.1.1 FrameworkBundle Configuration ("framework")

This reference document is a work in progress. It should be accurate, but all options are not yet fully covered.

The `FrameworkBundle` contains most of the "base" framework functionality and can be configured under the `framework` key in your application configuration. This includes settings related to sessions, translation, forms, validation, routing and more.

Configuration

- *charset*
- *secret*
- *ide*
- *test*
- ***form***
  - enabled
- ***csrf_protection***
  - enabled
  - *field_name*
- ***session***
  - *lifetime*
- ***templating***
  - *assets_base_urls*
  - *assets_version*
  - *assets_version_format*

charset

type: `string` **default**: `UTF-8`

The character set that's used throughout the framework. It becomes the service container parameter named `kernel.charset`.

secret

type: `string` **required**

This is a string that should be unique to your application. In practice, it's used for generating the CSRF tokens, but it could be used in any other context where having a unique string is useful. It becomes the service container parameter named `kernel.secret`.

ide

type: `string` **default**: `null`

If you're using an IDE like TextMate or Mac Vim, then Symfony can turn all of the file paths in an exception message into a link, which will open that file in your IDE.

If you use TextMate or Mac Vim, you can simply use one of the following built-in values:

- `textmate`
- `macvim`

You can also specify a custom file link string. If you do this, all percentage signs (`%`) must be doubled to escape that character. For example, the full TextMate string would look like this:

```
framework:
    ide:  "txmt://open?url=file://%%f&line=%%l"
```

Of course, since every developer uses a different IDE, it's better to set this on a system level. This can be done by setting the `xdebug.file_link_format` PHP.ini value to the file link string. If this configuration value is set, then the `ide` option does not need to be specified.

test

type: `Boolean`

If this configuration parameter is present (and not `false`), then the services related to testing your application (e.g. `test.client`) are loaded. This setting should be present in your `test` environment (usually via `app/config/config_test.yml`). For more information, see Testing.

form

csrf_protection

session

lifetime **type**: `integer` **default**: `0`

This determines the lifetime of the session - in seconds. By default it will use `0`, which means the cookie is valid for the length of the browser session.

templating

assets_base_urls **default**: `{ http: [], https: [] }`

This option allows you to define base URL's to be used for assets referenced from `http` and `https` pages. A string value may be provided in lieu of a single-element array. If multiple base URL's are provided, Symfony2 will select one from the collection each time it generates an asset's path.

For your convenience, `assets_base_urls` can be set directly with a string or array of strings, which will be automatically organized into collections of base URL's for `http` and `https` requests. If a URL starts with `https://` or is protocol-relative (i.e. starts with `//`) it will be added to both collections. URL's starting with `http://` will only be added to the `http` collection.

New in version 2.1: Unlike most configuration blocks, successive values for `assets_base_urls` will overwrite each other instead of being merged. This behavior was chosen because developers will typically define base URL's for each environment. Given that most projects tend to inherit configurations (e.g. `config_test.yml` imports `config_dev.yml`) and/or share a common base configuration (i.e. `config.yml`), merging could yield a set of base URL's for multiple environments.

assets_version **type**: `string`

This option is used to *bust* the cache on assets by globally adding a query parameter to all rendered asset paths (e.g. `/images/logo.png?v2`). This applies only to assets rendered via the Twig `asset` function (or PHP equivalent) as well as assets rendered with Assetic.

For example, suppose you have the following:

- *Twig*

```
<img src="{{ asset('images/logo.png') }}" alt="Symfony!" />
```

- *PHP*

```
<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" alt="Symfony!" />
```

By default, this will render a path to your image such as `/images/logo.png`. Now, activate the `assets_version` option:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    templating: { engines: ['twig'], assets_version: v2 }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:templating assets-version="v2">
    <framework:engine id="twig" />
</framework:templating>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'templating'      => array(
        'engines' => array('twig'),
        'assets_version' => 'v2',
    ),
));
```

Now, the same asset will be rendered as `/images/logo.png?v2` If you use this feature, you **must** manually increment the `assets_version` value before each deployment so that the query parameters change.

You can also control how the query string works via the *assets_version_format* option.

assets_version_format **type**: `string` **default**: `%%s?%%s`

This specifies a sprintf() pattern that will be used with the *assets_version* option to construct an asset's path. By default, the pattern adds the asset's version as a query string. For example, if `assets_version_format` is set to `%%s?version=%%s` and `assets_version` is set to `5`, the asset's path would be `/images/logo.png?version=5`.

Note: All percentage signs (`%`) in the format string must be doubled to escape the character. Without escaping, values might inadvertently be interpretted as *Service Parameters*.

Tip: Some CDN's do not support cache-busting via query strings, so injecting the version into the actual file path is necessary. Thankfully, `assets_version_format` is not limited to producing versioned query strings.

The pattern receives the asset's original path and version as its first and second parameters, respectively. Since the asset's path is one parameter, we cannot modify it in-place (e.g. `/images/logo-v5.png`); however, we can prefix the asset's path using a pattern of `version-%%2$s/%%1$s`, which would result in the path `version-5/images/logo.png`.

URL rewrite rules could then be used to disregard the version prefix before serving the asset. Alternatively, you could copy assets to the appropriate version path as part of your deployment process and forgot any URL rewriting. The latter option is useful if you would like older asset versions to remain accessible at their original URL.

Full Default Configuration

- *YAML*

```
framework:

    # general configuration
    charset:              ~
    secret:               ~ # Required
    ide:                  ~
    test:                 ~
    default_locale:       en
    trust_proxy_headers:  false

    # form configuration
    form:
        enabled:              true
    csrf_protection:
```

```
            enabled:              true
            field_name:           _token

        # esi configuration
        esi:
            enabled:              true

        # profiler configuration
        profiler:
            only_exceptions:      false
            only_master_requests:  false
            dsn:                  sqlite:%kernel.cache_dir%/profiler.db
            username:
            password:
            lifetime:             86400
            matcher:
                ip:                   ~
                path:                 ~
                service:              ~

        # router configuration
        router:
            resource:             ~ # Required
            type:                 ~
            http_port:            80
            https_port:           443

        # session configuration
        session:
            auto_start:           ~
            storage_id:           session.storage.native
            name:                 ~
            lifetime:             86400
            path:                 ~
            domain:               ~
            secure:               ~
            httponly:             ~

        # templating configuration
        templating:
            assets_version:       ~
            assets_version_format:  "%%s?%%s"
            assets_base_urls:
                http:                 []
                ssl:                  []
            cache:                ~
            engines:              # Required
            form:
                resources:        [FrameworkBundle:Form]

                # Example:
                - twig
            loaders:              []
            packages:

                # Prototype
                name:
                    version:              ~
```

```
                version_format:       ~
                base_urls:
                    http:                 []
                    ssl:                  []

        # translator configuration
        translator:
            enabled:              true
            fallback:             en

        # validation configuration
        validation:
            enabled:              true
            cache:                ~
            enable_annotations:   false

        # annotation configuration
        annotations:
            cache:                file
            file_cache_dir:       %kernel.cache_dir%/annotations
            debug:                true
```

5.1.2 AsseticBundle Configuration Reference

Full Default Configuration

- *YAML*

```
    assetic:
        debug:                true
        use_controller:       true
        read_from:            %kernel.root_dir%/../web
        write_to:             %assetic.read_from%
        java:                 /usr/bin/java
        node:                 /usr/bin/node
        sass:                 /usr/bin/sass
        bundles:

            # Defaults (all currently registered bundles):
            - FrameworkBundle
            - SecurityBundle
            - TwigBundle
            - MonologBundle
            - SwiftmailerBundle
            - DoctrineBundle
            - AsseticBundle
            - ...

        assets:

            # Prototype
            name:
                inputs:               []
                filters:              []
                options:
```

```
                # Prototype
                name:                 []
        filters:

            # Prototype
            name:                 []
        twig:
            functions:

                # Prototype
                name:                 []
```

5.1.3 Configuration Reference

- *YAML*

```
    doctrine:
        dbal:
            default_connection:   default
            connections:
                default:
                    dbname:               database
                    host:                 localhost
                    port:                 1234
                    user:                 user
                    password:             secret
                    driver:               pdo_mysql
                    driver_class:         MyNamespace\MyDriverImpl
                    options:
                        foo: bar
                    path:                 %kernel.data_dir%/data.sqlite
                    memory:               true
                    unix_socket:          /tmp/mysql.sock
                    wrapper_class:        MyDoctrineDbalConnectionWrapper
                    charset:              UTF8
                    logging:              %kernel.debug%
                    platform_service:     MyOwnDatabasePlatformService
                    mapping_types:
                        enum: string
                conn1:
                    # ...
            types:
                custom: Acme\HelloBundle\MyCustomType
        orm:
            auto_generate_proxy_classes:    false
            proxy_namespace:                Proxies
            proxy_dir:                      %kernel.cache_dir%/doctrine/orm/Proxies
            default_entity_manager:         default # The first defined is used if not set
            entity_managers:
                default:
                    # The name of a DBAL connection (the one marked as default is used if not set)
                    connection:                     conn1
                    mappings: # Required
                        AcmeHelloBundle: ~
                    class_metadata_factory_name:    Doctrine\ORM\Mapping\ClassMetadataFactory
                    # All cache drivers have to be array, apc, xcache or memcache
                    metadata_cache_driver:          array
```

```
                    query_cache_driver:             array
                    result_cache_driver:
                        type:           memcache
                        host:           localhost
                        port:           11211
                        instance_class: Memcache
                        class:          Doctrine\Common\Cache\MemcacheCache
                    dql:
                        string_functions:
                            test_string: Acme\HelloBundle\DQL\StringFunction
                        numeric_functions:
                            test_numeric: Acme\HelloBundle\DQL\NumericFunction
                        datetime_functions:
                            test_datetime: Acme\HelloBundle\DQL\DatetimeFunction
                    hydrators:
                        custom: Acme\HelloBundle\Hydrators\CustomHydrator
                em2:
                    # ...
```

- *XML*

```
    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                            http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

        <doctrine:config>
            <doctrine:dbal default-connection="default">
                <doctrine:connection
                    name="default"
                    dbname="database"
                    host="localhost"
                    port="1234"
                    user="user"
                    password="secret"
                    driver="pdo_mysql"
                    driver-class="MyNamespace\MyDriverImpl"
                    path="%kernel.data_dir%/data.sqlite"
                    memory="true"
                    unix-socket="/tmp/mysql.sock"
                    wrapper-class="MyDoctrineDbalConnectionWrapper"
                    charset="UTF8"
                    logging="%kernel.debug%"
                    platform-service="MyOwnDatabasePlatformService"
                >
                    <doctrine:option key="foo">bar</doctrine:option>
                    <doctrine:mapping-type name="enum">string</doctrine:mapping-type>
                </doctrine:connection>
                <doctrine:connection name="conn1" />
                <doctrine:type name="custom">Acme\HelloBundle\MyCustomType</doctrine:type>
            </doctrine:dbal>

            <doctrine:orm default-entity-manager="default" auto-generate-proxy-classes="false" proxy
                <doctrine:entity-manager name="default" query-cache-driver="array" result-cache-driv
                    <doctrine:metadata-cache-driver type="memcache" host="localhost" port="11211" in
                    <doctrine:mapping name="AcmeHelloBundle" />
                    <doctrine:dql>
                        <doctrine:string-function name="test_string>Acme\HelloBundle\DQL\StringFunct
```

```
                        <doctrine:numeric-function name="test_numeric>Acme\HelloBundle\DQL\NumericFu
                        <doctrine:datetime-function name="test_datetime>Acme\HelloBundle\DQL\Datetim
                    </doctrine:dql>
                </doctrine:entity-manager>
                <doctrine:entity-manager name="em2" connection="conn2" metadata-cache-driver="apc">
                    <doctrine:mapping
                        name="DoctrineExtensions"
                        type="xml"
                        dir="%kernel.root_dir%/../src/vendor/DoctrineExtensions/lib/DoctrineExtensic
                        prefix="DoctrineExtensions\Entity"
                        alias="DExt"
                    />
                </doctrine:entity-manager>
            </doctrine:orm>
        </doctrine:config>
    </container>
```

Configuration Overview

This following configuration example shows all the configuration defaults that the ORM resolves to:

```
doctrine:
    orm:
        auto_mapping: true
        # the standard distribution overrides this to be true in debug, false otherwise
        auto_generate_proxy_classes: false
        proxy_namespace: Proxies
        proxy_dir: %kernel.cache_dir%/doctrine/orm/Proxies
        default_entity_manager: default
        metadata_cache_driver: array
        query_cache_driver: array
        result_cache_driver: array
```

There are lots of other configuration options that you can use to overwrite certain classes, but those are for very advanced use-cases only.

Caching Drivers

For the caching drivers you can specify the values “array”, “apc”, “memcache” or “xcache”.

The following example shows an overview of the caching configurations:

```
doctrine:
    orm:
        auto_mapping: true
        metadata_cache_driver: apc
        query_cache_driver: xcache
        result_cache_driver:
            type: memcache
            host: localhost
            port: 11211
            instance_class: Memcache
```

Mapping Configuration

Explicit definition of all the mapped entities is the only necessary configuration for the ORM and there are several configuration options that you can control. The following configuration options exist for a mapping:

- `type` One of `annotation`, `xml`, `yml`, `php` or `staticphp`. This specifies which type of metadata type your mapping uses.
- `dir` Path to the mapping or entity files (depending on the driver). If this path is relative it is assumed to be relative to the bundle root. This only works if the name of your mapping is a bundle name. If you want to use this option to specify absolute paths you should prefix the path with the kernel parameters that exist in the DIC (for example %kernel.root_dir%).
- `prefix` A common namespace prefix that all entities of this mapping share. This prefix should never conflict with prefixes of other defined mappings otherwise some of your entities cannot be found by Doctrine. This option defaults to the bundle namespace + `Entity`, for example for an application bundle called `AcmeHelloBundle` prefix would be `Acme\HelloBundle\Entity`.
- `alias` Doctrine offers a way to alias entity namespaces to simpler, shorter names to be used in DQL queries or for Repository access. When using a bundle the alias defaults to the bundle name.
- `is_bundle` This option is a derived value from `dir` and by default is set to true if dir is relative proved by a `file_exists()` check that returns false. It is false if the existence check returns true. In this case an absolute path was specified and the metadata files are most likely in a directory outside of a bundle.

Doctrine DBAL Configuration

Note: DoctrineBundle supports all parameters that default Doctrine drivers accept, converted to the XML or YAML naming standards that Symfony enforces. See the Doctrine DBAL documentation for more information.

Besides default Doctrine options, there are some Symfony-related ones that you can configure. The following block shows all possible configuration keys:

- *YAML*

```
doctrine:
    dbal:
        dbname:               database
        host:                 localhost
        port:                 1234
        user:                 user
        password:             secret
        driver:               pdo_mysql
        driver_class:         MyNamespace\MyDriverImpl
        options:
            foo: bar
        path:                 %kernel.data_dir%/data.sqlite
        memory:               true
        unix_socket:          /tmp/mysql.sock
        wrapper_class:        MyDoctrineDbalConnectionWrapper
        charset:              UTF8
        logging:              %kernel.debug%
        platform_service:     MyOwnDatabasePlatformService
        mapping_types:
            enum: string
        types:
            custom: Acme\HelloBundle\MyCustomType
```

- *XML*

```xml
<!-- xmlns:doctrine="http://symfony.com/schema/dic/doctrine" -->
<!-- xsi:schemaLocation="http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/do

<doctrine:config>
    <doctrine:dbal
        name="default"
        dbname="database"
        host="localhost"
        port="1234"
        user="user"
        password="secret"
        driver="pdo_mysql"
        driver-class="MyNamespace\MyDriverImpl"
        path="%kernel.data_dir%/data.sqlite"
        memory="true"
        unix-socket="/tmp/mysql.sock"
        wrapper-class="MyDoctrineDbalConnectionWrapper"
        charset="UTF8"
        logging="%kernel.debug%"
        platform-service="MyOwnDatabasePlatformService"
    >
        <doctrine:option key="foo">bar</doctrine:option>
        <doctrine:mapping-type name="enum">string</doctrine:mapping-type>
        <doctrine:type name="custom">Acme\HelloBundle\MyCustomType</doctrine:type>
    </doctrine:dbal>
</doctrine:config>
```

If you want to configure multiple connections in YAML, put them under the `connections` key and give them a unique name:

```yaml
doctrine:
    dbal:
        default_connection:       default
        connections:
            default:
                dbname:           Symfony2
                user:             root
                password:         null
                host:             localhost
            customer:
                dbname:           customer
                user:             root
                password:         null
                host:             localhost
```

The `database_connection` service always refers to the *default* connection, which is the first one defined or the one configured via the `default_connection` parameter.

Each connection is also accessible via the `doctrine.dbal.[name]_connection` service where `[name]` if the name of the connection.

5.1.4 Security Configuration Reference

The security system is one of the most powerful parts of Symfony2, and can largely be controlled via its configuration.

Full Default Configuration

The following is the full default configuration for the security system. Each part will be explained in the next section.

- *YAML*

```
# app/config/security.yml
security:
    access_denied_url: /foo/error403

    always_authenticate_before_granting: false

    # whether or not to call eraseCredentials on the token
    erase_credentials: true

    # strategy can be: none, migrate, invalidate
    session_fixation_strategy: migrate

    access_decision_manager:
        strategy: affirmative
        allow_if_all_abstain: false
        allow_if_equal_granted_denied: true

    acl:
        connection: default # any name configured in doctrine.dbal section
        tables:
            class: acl_classes
            entry: acl_entries
            object_identity: acl_object_identities
            object_identity_ancestors: acl_object_identity_ancestors
            security_identity: acl_security_identities
        cache:
            id: service_id
            prefix: sf2_acl_
        voter:
            allow_if_object_identity_unavailable: true

    encoders:
        somename:
            class: Acme\DemoBundle\Entity\User
        Acme\DemoBundle\Entity\User: sha512
        Acme\DemoBundle\Entity\User: plaintext
        Acme\DemoBundle\Entity\User:
            algorithm: sha512
            encode_as_base64: true
            iterations: 5000
        Acme\DemoBundle\Entity\User:
            id: my.custom.encoder.service.id

    providers:
        memory_provider_name:
            memory:
                users:
                    foo: { password: foo, roles: ROLE_USER }
                    bar: { password: bar, roles: [ROLE_USER, ROLE_ADMIN] }
        entity_provider_name:
            entity: { class: SecurityBundle:User, property: username }

    factories:
```

```
            MyFactory: %kernel.root_dir%/../src/Acme/DemoBundle/Resources/config/security_factories.

        firewalls:
            somename:
                pattern: .*
                request_matcher: some.service.id
                access_denied_url: /foo/error403
                access_denied_handler: some.service.id
                entry_point: some.service.id
                provider: some_provider_key_from_above
                context: name
                stateless: false
                x509:
                    provider: some_provider_key_from_above
                http_basic:
                    provider: some_provider_key_from_above
                http_digest:
                    provider: some_provider_key_from_above
                form_login:
                    check_path: /login_check
                    login_path: /login
                    use_forward: false
                    always_use_default_target_path: false
                    default_target_path: /
                    target_path_parameter: _target_path
                    use_referer: false
                    failure_path: /foo
                    failure_forward: false
                    failure_handler: some.service.id
                    success_handler: some.service.id
                    username_parameter: _username
                    password_parameter: _password
                    csrf_parameter: _csrf_token
                    intention: authenticate
                    csrf_provider: my.csrf_provider.id
                    post_only: true
                    remember_me: false
                remember_me:
                    token_provider: name
                    key: someS3cretKey
                    name: NameOfTheCookie
                    lifetime: 3600 # in seconds
                    path: /foo
                    domain: somedomain.foo
                    secure: true
                    httponly: true
                    always_remember_me: false
                    remember_me_parameter: _remember_me
                logout:
                    path:   /logout
                    target: /
                    invalidate_session: false
                    delete_cookies:
                        a: { path: null, domain: null }
                        b: { path: null, domain: null }
                    handlers: [some.service.id, another.service.id]
                    success_handler: some.service.id
                anonymous: ~
```

```
        access_control:
            -
                path: ^/foo
                host: mydomain.foo
                ip: 192.0.0.0/8
                roles: [ROLE_A, ROLE_B]
                requires_channel: https

        role_hierarchy:
            ROLE_SUPERADMIN: ROLE_ADMIN
            ROLE_SUPERADMIN: 'ROLE_ADMIN, ROLE_USER'
            ROLE_SUPERADMIN: [ROLE_ADMIN, ROLE_USER]
            anything: { id: ROLE_SUPERADMIN, value: 'ROLE_USER, ROLE_ADMIN' }
            anything: { id: ROLE_SUPERADMIN, value: [ROLE_USER, ROLE_ADMIN] }
```

Form Login Configuration

When using the `form_login` authentication listener beneath a firewall, there are several common options for configuring the "form login" experience:

The Login Form and Process

- `login_path` (type: `string`, default: `/login`) This is the URL that the user will be redirected to (unless `use_forward` is set to `true`) when he/she tries to access a protected resource but isn't fully authenticated.

  This URL **must** be accessible by a normal, un-authenticated user, else you may create a redirect loop. For details, see "*Avoid Common Pitfalls*".

- `check_path` (type: `string`, default: `/login_check`) This is the URL that your login form must submit to. The firewall will intercept any requests (`POST` requests only, be default) to this URL and process the submitted login credentials.

  Be sure that this URL is covered by your main firewall (i.e. don't create a separate firewall just for `check_path` URL).

- `use_forward` (type: `Boolean`, default: `false`) If you'd like the user to be forwarded to the login form instead of being redirected, set this option to `true`.

- `username_parameter` (type: `string`, default: `_username`) This is the field name that you should give to the username field of your login form. When you submit the form to `check_path`, the security system will look for a POST parameter with this name.

- `password_parameter` (type: `string`, default: `_password`) This is the field name that you should give to the password field of your login form. When you submit the form to `check_path`, the security system will look for a POST parameter with this name.

- `post_only` (type: `Boolean`, default: `true`) By default, you must submit your login form to the `check_path` URL as a POST request. By setting this option to `false`, you can send a GET request to the `check_path` URL.

Redirecting after Login

- `always_use_default_target_path` (type: `Boolean`, default: `false`)

- `default_target_path` (type: `string`, default: `/`)

- `target_path_parameter` (type: `string`, default: `_target_path`)
- `use_referer` (type: `Boolean`, default: `false`)

5.1.5 SwiftmailerBundle Configuration ("swiftmailer")

This reference document is a work in progress. It should be accurate, but all options are not yet fully covered. For a full list of the default configuration options, see *Full Default Configuration*

The `swiftmailer` key configures Symfony's integration with Swiftmailer, which is responsible for creating and delivering email messages.

Configuration

- *transport*
- *username*
- *password*
- *host*
- *port*
- *encryption*
- *auth_mode*
- ***spool***
  - *type*
  - *path*
- *sender_address*
- ***antiflood***
  - *threshold*
  - *sleep*
- *delivery_address*
- *disable_delivery*
- *logging*

transport

type: `string` **default**: `smtp`

The exact transport method to use to deliver emails. Valid values are:

- smtp
- gmail (see How to use Gmail to send Emails)
- mail
- sendmail
- null (same as setting *disable_delivery* to `true`)

username

type: `string`

The username when using `smtp` as the transport.

password

type: `string`

The password when using `smtp` as the transport.

host

type: `string` **default**: `localhost`

The host to connect to when using `smtp` as the transport.

port

type: `string` **default**: 25 or 465 (depending on *encryption*)

The port when using `smtp` as the transport. This defaults to 465 if encryption is `ssl` and 25 otherwise.

encryption

type: `string`

The encryption mode to use when using `smtp` as the transport. Valid values are `tls`, `ssl`, or `null` (indicating no encryption).

auth_mode

type: `string`

The authentication mode to use when using `smtp` as the transport. Valid values are `plain`, `login`, `cram-md5`, or `null`.

spool

For details on email spooling, see How to Spool Email.

type **type**: `string` **default**: `file`

The method used to store spooled messages. Currently only `file` is supported. However, a custom spool should be possible by creating a service called `swiftmailer.spool.myspool` and setting this value to `myspool`.

path **type**: `string` **default**: `%kernel.cache_dir%/swiftmailer/spool`

When using the `file` spool, this is the path where the spooled messages will be stored.

sender_address

type: `string`

If set, all messages will be delivered with this address as the "return path" address, which is where bounced messages should go. This is handled internally by Swiftmailer's `Swift_Plugins_ImpersonatePlugin` class.

antiflood

threshold **type**: `string` **default**: `99`

Used with `Swift_Plugins_AntiFloodPlugin`. This is the number of emails to send before restarting the transport.

sleep **type**: `string` **default**: `0`

Used with `Swift_Plugins_AntiFloodPlugin`. This is the number of seconds to sleep for during a transport restart.

delivery_address

type: `string`

If set, all email messages will be sent to this address instead of being sent to their actual recipients. This is often useful when developing. For example, by setting this in the `config_dev.yml` file, you can guarantee that all emails sent during development go to a single account.

This uses `Swift_Plugins_RedirectingPlugin`. Original recipients are available on the `X-Swift-To`, `X-Swift-Cc` and `X-Swift-Bcc` headers.

disable_delivery

type: `Boolean` **default**: `false`

If true, the `transport` will automatically be set to `null`, and no emails will actually be delivered.

logging

type: `Boolean` **default**: `%kernel.debug%`

If true, Symfony's data collector will be activated for Swiftmailer and the information will be available in the profiler.

Full Default Configuration

- *YAML*

```
swiftmailer:
    transport:            smtp
    username:             ~
    password:             ~
    host:                 localhost
    port:                 false
    encryption:           ~
```

```
        auth_mode:            ~
        spool:
            type:                 file
            path:                 %kernel.cache_dir%/swiftmailer/spool
        sender_address:       ~
        antiflood:
            threshold:            99
            sleep:                0
        delivery_address:     ~
        disable_delivery:     ~
        logging:              %kernel.debug%
```

5.1.6 TwigBundle Configuration Reference

- *YAML*

```
    twig:
        form:
            resources:

                # Default:
                - div_layout.html.twig

                # Example:
                - MyBundle::form.html.twig
        globals:

            # Examples:
            foo:                  "@bar"
            pi:                   3.14

            # Prototype
            key:
                id:                   ~
                type:                 ~
                value:                ~
        autoescape:           ~
        base_template_class:  ~ # Example: Twig_Template
        cache:                %kernel.cache_dir%/twig
        charset:              %kernel.charset%
        debug:                %kernel.debug%
        strict_variables:     ~
        auto_reload:          ~
        exception_controller:  Symfony\Bundle\TwigBundle\Controller\ExceptionController::showAction
```

- *XML*

```
    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:twig="http://symfony.com/schema/dic/twig"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                            http://symfony.com/schema/dic/twig http://symfony.com/schema/dic/doctrin

        <twig:config auto-reload="%kernel.debug%" autoescape="true" base-template-class="Twig_Templa
            <twig:form>
                <twig:resource>MyBundle::form.html.twig</twig:resource>
            </twig:form>
```

```
            <twig:global key="foo" id="bar" type="service" />
            <twig:global key="pi">3.14</twig:global>
        </twig:config>
    </container>
```

- *PHP*

```
    $container->loadFromExtension('twig', array(
        'form' => array(
            'resources' => array(
                'MyBundle::form.html.twig',
            )
        ),
        'globals' => array(
            'foo' => '@bar',
            'pi'  => 3.14,
        ),
        'auto_reload'         => '%kernel.debug%',
        'autoescape'          => true,
        'base_template_class' => 'Twig_Template',
        'cache'               => '%kernel.cache_dir%/twig',
        'charset'             => '%kernel.charset%',
        'debug'               => '%kernel.debug%',
        'strict_variables'    => false,
    ));
```

Configuration

exception_controller

type: `string` **default**: `Symfony\\Bundle\\TwigBundle\\Controller\\ExceptionController::showAction`

This is the controller that is activated after an exception is thrown anywhere in your application. The default controller (`Symfony\Bundle\TwigBundle\Controller\ExceptionController`) is what's responsible for rendering specific templates under different error conditions (see How to customize Error Pages). Modifying this option is advanced. If you need to customize an error page you should use the previous link. If you need to perform some behavior on an exception, you should add a listener to the `kernel.exception` event (see *Enabling Custom Listeners*).

5.1.7 Configuration Reference

- *YAML*

```
    monolog:
        handlers:

            # Examples:
            syslog:
                type:               stream
                path:               /var/log/symfony.log
                level:              ERROR
                bubble:             false
                formatter:          my_formatter
            main:
                type:               fingers_crossed
```

```
                action_level:         WARNING
                buffer_size:          30
                handler:              custom
            custom:
                type:                 service
                id:                   my_handler

            # Prototype
            name:
                type:                  ~ # Required
                id:                    ~
                priority:              0
                level:                 DEBUG
                bubble:                true
                path:                  %kernel.logs_dir%/%kernel.environment%.log
                ident:                 false
                facility:              user
                max_files:             0
                action_level:          WARNING
                stop_buffering:        true
                buffer_size:           0
                handler:               ~
                members:               []
                from_email:            ~
                to_email:              ~
                subject:               ~
                email_prototype:
                    id:      ~ # Required (when the email_prototype is used)
                    method: ~
                formatter:             ~
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <monolog:config>
        <monolog:handler
            name="syslog"
            type="stream"
            path="/var/log/symfony.log"
            level="error"
            bubble="false"
            formatter="my_formatter"
        />
        <monolog:handler
            name="main"
            type="fingers_crossed"
            action-level="warning"
            handler="custom"
        />
        <monolog:handler
            name="custom"
            type="service"
            id="my_handler"
        />
```

```
        </monolog:config>
    </container>
```

Note: When the profiler is enabled, a handler is added to store the logs' messages in the profiler. The profiler uses the name "debug" so it is reserved and cannot be used in the configuration.

5.1.8 WebProfilerBundle Configuration

Full Default Configuration

- *YAML*

```
web_profiler:

    # display secondary information to make the toolbar shorter
    verbose:             true

    # display the web debug toolbar at the bottom of pages with a summary of profiler info
    toolbar:             false

    # gives you the opportunity to look at the collected data before following the redirect
    intercept_redirects: false
```

5.1.9 Form Types Reference

birthday Field Type

A date field that specializes in handling birthdate data.

Can be rendered as a single text box, three text boxes (month, day, and year), or three select boxes.

This type is essentially the same as the date type, but with a more appropriate default for the *years* option. The *years* option defaults to 120 years ago to the current year.

Underlying Data Type	can be `DateTime`, `string`, `timestamp`, or `array` (see the *input option*)
Rendered as	can be three select boxes or 1 or 3 text boxes, based on the *widget* option
Options	• *years*
Inherited options	• *widget* • *input* • *months* • *days* • *format* • *pattern* • *data_timezone* • *user_timezone*
Parent type	date
Class	`Symfony\Component\Form\Extension\Core\Type\Birthda`

Field Options

years **type**: `array` **default**: 120 years ago to the current year

List of years available to the year field type. This option is only relevant when the `widget` option is set to `choice`.

Inherited options

These options inherit from the date type:

widget **type**: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders three select inputs. The order of the selects is defined in the *pattern* option.
- `text`: renders a three field input of type text (month, day, year).
- `single_text`: renders a single input of type text. User's input is validated based on the *format* option.

input **type**: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array('year' => 2011, 'month' => 06, 'day' => 05)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

months **type**: `array` **default**: 1 to 12

List of months available to the month field type. This option is only relevant when the `widget` option is set to `choice`.

days **type**: `array` **default**: 1 to 31

List of days available to the day field type. This option is only relevant when the `widget` option is set to `choice`:

```
'days' => range(1,31)
```

format **type**: `integer` or `string` **default**: `IntlDateFormatter::MEDIUM`

Option passed to the `IntlDateFormatter` class, used to transform user input into the proper format. This is critical when the *widget* option is set to `single_text`, and will define how the user will input the data. By default, the format is determined based on the current user locale; you can override it by passing the format as a string.

For more information on valid formats, see Date/Time Format Syntax. For example, to render a single text box that expects the user to end `yyyy-MM-dd`, use the following options:

```
$builder->add('date_created', 'date', array(
    'widget' => 'single_text',
    'format' => 'yyyy-MM-dd',
));
```

pattern **type**: `string`

This option is only relevant when the *widget* is set to `choice`. The default pattern is based off the *format* option, and tries to match the characters `M`, `d`, and `y` in the format pattern. If no match is found, the default is the string `{{ year }}-{{ month }}-{{ day }}`. Tokens for this option include:

- `{{ year }}`: Replaced with the `year` widget
- `{{ month }}`: Replaced with the `month` widget
- `{{ day }}`: Replaced with the `day` widget

data_timezone **type**: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the PHP supported timezones

user_timezone **type**: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the PHP supported timezones

checkbox Field Type

Creates a single input checkbox. This should always be used for a field that has a Boolean value: if the box is checked, the field will be set to true, if the box is unchecked, the value will be set to false.

Rendered as	`input text` field
Options	• *value*
Inherited options	• *required* • *label* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\Checkbo`

Example Usage

```
$builder->add('public', 'checkbox', array(
    'label'     => 'Show this entry publicly?',
    'required'  => false,
));
```

Field Options

value **type**: `mixed` **default**: `1`

The value that's actually used as the value for the checkbox. This does not affect the value that's set on your object.

Inherited options

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

choice Field Type

A multi-purpose field used to allow the user to "choose" one or more options. It can be rendered as a `select` tag, radio buttons, or checkboxes.

To use this field, you must specify *either* the `choice_list` or `choices` option.

Rendered as	can be various tags (see below)
Options	• *choices* • *choice_list* • *multiple* • *expanded* • *preferred_choices* • *empty_value* • *empty_data*
Inherited options	• *required* • *label* • *read_only* • *error_bubbling*
Parent type	form (if expanded), `field` otherwise
Class	`Symfony\Component\Form\Extension\Core\Type\ChoiceT`

Example Usage

The easiest way to use this field is to specify the choices directly via the `choices` option. The key of the array becomes the value that's actually set on your underlying object (e.g. `m`), while the value is what the user sees on the form (e.g. `Male`).

```
$builder->add('gender', 'choice', array(
    'choices'   => array('m' => 'Male', 'f' => 'Female'),
    'required'  => false,
));
```

By setting `multiple` to true, you can allow the user to choose multiple values. The widget will be rendered as a multiple `select` tag or a series of checkboxes depending on the `expanded` option:

```
$builder->add('availability', 'choice', array(
    'choices'   => array(
        'morning'   => 'Morning',
        'afternoon' => 'Afternoon',
        'evening'   => 'Evening',
    ),
    'multiple'  => true,
));
```

You can also use the `choice_list` option, which takes an object that can specify the choices for your widget.

Select tag, Checkboxes or Radio Buttons

This field may be rendered as one of several different HTML fields, depending on the `expanded` and `multiple` options:

element type	expanded	multiple
select tag	false	false
select tag (with `multiple` attribute)	false	true
radio buttons	true	false
checkboxes	true	true

Field Options

choices **type**: `array` **default**: `array()`

This is the most basic way to specify the choices that should be used by this field. The `choices` option is an array, where the array key is the item value and the array value is the item's label:

```
$builder->add('gender', 'choice', array(
    'choices' => array('m' => 'Male', 'f' => 'Female')
));
```

choice_list **type**: `Symfony\Component\Form\Extension\Core\ChoiceList\ChoiceListInterface`

This is one way of specifying the options to be used for this field. The `choice_list` option must be an instance of the `ChoiceListInterface`. For more advanced cases, a custom class that implements the interface can be created to supply the choices.

multiple **type**: `Boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

expanded **type**: `Boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

preferred_choices **type**: `array` **default**: `array()`

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

- *Twig*

```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['foo_choices'], array('separator' => '=====')) ?>
```

empty_value **type**: `string` or `Boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if both the `expanded` and `multiple` options are set to false.

- Add an empty value with "Choose an option" as the text:

```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

empty_data **type**: `mixed` **default**: `array()` if `multiple` or `expanded`, `''` otherwise

This option determines what value the field will return when the `empty_value` choice is selected.

For example, if you want the `gender` field to be set to `null` when no value is selected, you can do it like this:

```
$builder->add('gender', 'choice', array(
    'choices' => array(
        'm' => 'Male',
        'f' => 'Female'
    ),
    'required'    => false,
    'empty_value' => 'Choose your gender',
    'empty_data'  => null
));
```

Inherited options

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

collection Field Type

This field type is used to render a "collection" of some field or form. In the easiest sense, it could be an array of `text` fields that populate an array `emails` field. In more complex examples, you can embed entire forms, which is useful when creating forms that expose one-to-many relationships (e.g. a product from where you can manage many related product photos).

Rendered as	depends on the *type* option
Options	• *type* • *options* • *allow_add* • *allow_delete* • *prototype*
Inherited options	• *label* • *error_bubbling* • *by_reference*
Parent type	form
Class	`Symfony\Component\Form\Extension\Core\Type\Collect`

Basic Usage

This type is used when you want to manage a collection of similar items in a form. For example, suppose you have an `emails` field that corresponds to an array of email addresses. In the form, you want to expose each email address as its own input text box:

```
$builder->add('emails', 'collection', array(
    // each item in the array will be an "email" field
    'type'   => 'email',
    // these options are passed to each "email" type
    'options'  => array(
        'required'  => false,
        'attr'      => array('class' => 'email-box')
    ),
));
```

The simplest way to render this is all at once:

- *Twig*

```
{{ form_row(form.emails) }}
```

- *PHP*

```
<?php echo $view['form']->row($form['emails]) ?>
```

A much more flexible method would look like this:

- *Twig*

```
{{ form_label(form.emails) }}
{{ form_errors(form.emails) }}

<ul>
{% for emailField in form.emails %}
    <li>
        {{ form_errors(emailField) }}
        {{ form_widget(emailField) }}
    </li>
{% endfor %}
</ul>
```

- *PHP*

```
    <?php echo $view['form']->label($form['emails']) ?>
    <?php echo $view['form']->errors($form['emails']) ?>

    <ul>
    {% for emailField in form.emails %}
    <?php foreach ($form['emails'] as $emailField): ?>
        <li>
            <?php echo $view['form']->errors($emailField) ?>
            <?php echo $view['form']->widget($emailField) ?>
        </li>
    <?php endforeach; ?>
    </ul>
```

In both cases, no input fields would render unless your `emails` data array already contained some emails.

In this simple example, it's still impossible to add new addresses or remove existing addresses. Adding new addresses is possible by using the *allow_add* option (and optionally the *prototype* option) (see example below). Removing emails from the `emails` array is possible with the *allow_delete* option.

Adding and Removing items If *allow_add* is set to `true`, then if any unrecognized items are submitted, they'll be added seamlessly to the array of items. This is great in theory, but takes a little bit more effort in practice to get the client-side JavaScript correct.

Following along with the previous example, suppose you start with two emails in the `emails` data array. In that case, two input fields will be rendered that will look something like this (depending on the name of your form):

```
<input type="email" id="form_emails_1" name="form[emails][0]" value="foo@foo.com" />
<input type="email" id="form_emails_1" name="form[emails][1]" value="bar@bar.com" />
```

To allow your user to add another email, just set *allow_add* to `true` and - via JavaScript - render another field with the name `form[emails][2]` (and so on for more and more fields).

To help make this easier, setting the *prototype* option to `true` allows you to render a "template" field, which you can then use in your JavaScript to help you dynamically create these new fields. A rendered prototype field will look like this:

```
<input type="email" id="form_emails_$$name$$" name="form[emails][$$name$$]" value="" />
```

By replacing `$$name$$` with some unique value (e.g. `2`), you can build and insert new HTML fields into your form.

Using jQuery, a simple example might look like this. If you're rendering your collection fields all at once (e.g. `form_row(form.emails)`), then things are even easier because the `data-prototype` attribute is rendered automatically for you (with a slight difference - see note below) and all you need is the JavaScript:

- *Twig*

```
    <form action="..." method="POST" {{ form_enctype(form) }}>
        {# ... #}

        {# store the prototype on the data-prototype attribute #}
        <ul id="email-fields-list" data-prototype="{{ form_widget(form.emails.get('prototype')) | e
        {% for emailField in form.emails %}
            <li>
                {{ form_errors(emailField) }}
                {{ form_widget(emailField) }}
            </li>
        {% endfor %}
        </ul>
```

```
        <a href="#" id="add-another-email">Add another email</a>

        {# ... #}
    </form>

    <script type="text/javascript">
        // keep track of how many email fields have been rendered
        var emailCount = '{{ form.emails | length }}';

        jQuery(document).ready(function() {
            jQuery('#add-another-email').click(function() {
                var emailList = jQuery('#email-fields-list');

                // grab the prototype template
                var newWidget = emailList.attr('data-prototype');
                // replace the "$$name$$" used in the id and name of the prototype
                // with a number that's unique to our emails
                // end name attribute looks like name="contact[emails][2]"
                newWidget = newWidget.replace(/\$\$name\$\$/g, emailCount);
                emailCount++;

                // create a new list element and add it to our list
                var newLi = jQuery('<li></li>').html(newWidget);
                newLi.appendTo(jQuery('#email-fields-list'));

                return false;
            });
        })
    </script>
```

Tip: If you're rendering the entire collection at once, then the prototype is automatically available on the `data-prototype` attribute of the element (e.g. `div` or `table`) that surrounds your collection. The only difference is that the entire "form row" is rendered for you, meaning you wouldn't have to wrap it in any container element like we've done above.

Field Options

type **type**: `string` or `Symfony\Component\Form\FormTypeInterface` **required**

This is the field type for each item in this collection (e.g. `text`, `choice`, etc). For example, if you have an array of email addresses, you'd use the :doc`email</reference/forms/types/email>` type. If you want to embed a collection of some other form, create a new instance of your form type and pass it as this option.

options **type**: `array` **default**: `array()`

This is the array that's passed to the form type specified in the *type* option. For example, if you used the :doc`choice</reference/forms/types/choice>` type as your *type* option (e.g. for a collection of drop-down menus), then you'd need to at least pass the `choices` option to the underlying type:

```
$builder->add('favorite_cities', 'collection', array(
    'type'   => 'choice',
    'options'  => array(
        'choices'  => array(
            'nashville' => 'Nashville',
            'paris'     => 'Paris',
```

```
            'berlin'    => 'Berlin',
            'london'    => 'London',
        ),
    ),
));
```

allow_add **type**: `Boolean` **default**: `false`

If set to `true`, then if unrecognized items are submitted to the collection, they will be added as new items. The ending array will contain the existing items as well as the new item that was in the submitted data. See the above example for more details.

The *prototype* option can be used to help render a prototype item that can be used - with JavaScript - to create new form items dynamically on the client side. For more information, see the above example and *Allowing "new" todos with the "prototype"*.

Caution: If you're embedding entire other forms to reflect a one-to-many database relationship, you may need to manually ensure that the foreign key of these new objects is set correctly. If you're using Doctrine, this won't happen automatically. See the above link for more details.

allow_delete **type**: `Boolean` **default**: `false`

If set to `true`, then if an existing item is not contained in the submitted data, it will be correctly absent from the final array of items. This means that you can implement a "delete" button via JavaScript which simply removes a form element from the DOM. When the user submits the form, its absence from the submitted data will mean that it's removed from the final array.

For more information, see *Allowing todos to be removed*.

Caution: Be careful when using this option when you're embedding a collection of objects. In this case, if any embedded forms are removed, they *will* correctly be missing from the final array of objects. However, depending on your application logic, when one of those objects is removed, you may want to delete it or at least remove its foreign key reference to the main object. None of this is handled automatically. For more information, see *Allowing todos to be removed*.

prototype **type**: `Boolean` **default**: `true`

This option is useful when using the *allow_add* option. If `true` (and if *allow_add* is also `true`), a special "prototype" attribute will be available so that you can render a "template" example on your page of what a new element should look like. The `name` attribute given to this element is `$$name$$`. This allows you to add a "add another" button via JavaScript which reads the prototype, replaces `$$name$$` with some unique name or number, and render it inside your form. When submitted, it will be added to your underlying array due to the *allow_add* option.

The prototype field can be rendered via the `prototype` variable in the collection field:

- *Twig*

```
{{ form_row(form.emails.get('prototype')) }}
```

- *PHP*

```
<?php echo $view['form']->row($form['emails']->get('prototype')) ?>
```

Note that all you really need is the "widget", but depending on how you're rendering your form, having the entire "form row" may be easier for you.

Tip: If you're rendering the entire collection field at once, then the prototype form row is automatically available on the `data-prototype` attribute of the element (e.g. `div` or `table`) that surrounds your collection.

For details on how to actually use this option, see the above example as well as *Allowing "new" todos with the "prototype"*.

Inherited options

These options inherit from the field type. Not all options are listed here - only the most applicable to this type:

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

error_bubbling **type**: `Boolean` **default**: `true`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

by_reference **type**: `Boolean` **default**: `true`

If the underlying value of a field is an object and this option is set to `true`, then the resulting object won't actually be set when binding the form. For example, if you have a protected `author` field on your underlying object which is an instance of some `Author` object, then if `by_reference` is `false`, that `Author` object will be updated with the submitted data, but `setAuthor` will not actually be called on the main object. Since the `Author` object is a reference, this only really makes a difference if you have some custom logic in your `setAuthor` method that you want to guarantee will be run. In that case, set this option to `false`.

country Field Type

The `country` type is a subset of the `ChoiceType` that displays countries of the world. As an added bonus, the country names are displayed in the language of the user.

The "value" for each country is the two-letter country code.

Note: The locale of your user is guessed using Locale::getDefault()

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses all of the countries of the world. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select tag, Checkboxes or Radio Buttons*)
Inherited options	• *multiple* • *expanded* • *preferred_choices* • *empty_value* • *error_bubbling* • *required* • *label* • *read_only*
Parent type	choice
Class	`Symfony\Component\Form\Extension\Core\Type\Country`

Inherited options

These options inherit from the choice type:

multiple **type**: `Boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

expanded **type**: `Boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

preferred_choices **type**: `array` **default**: `array()`

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

- *Twig*

```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['foo_choices'], array('separator' => '=====')) ?>
```

empty_value **type**: `string` or `Boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if both the `expanded` and `multiple` options are set to false.

- Add an empty value with "Choose an option" as the text:

```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

csrf Field Type

The `csrf` type is a hidden input field containing a CSRF token.

Rendered as	`input hidden` field
Options	• `csrf_provider` • `intention` • `property_path`
Parent type	`hidden`
Class	`Symfony\Component\Form\Extension\Csrf\Type\CsrfTyp`

Field Options

csrf_provider **type**: `Symfony\Component\Form\CsrfProvider\CsrfProviderInterface`

The `CsrfProviderInterface` object that should generate the CSRF token. If not set, this defaults to the default provider.

intention **type**: `string`

An optional unique identifier used to generate the CSRF token.

property_path **type**: `any` **default**: `the field's value`

Fields display a property value of the form's domain object by default. When the form is submitted, the submitted value is written back into the object.

If you want to override the property that a field reads from and writes to, you can set the `property_path` option. Its default value is the field's name.

If you wish the field to be ignored when reading or writing to the object you can set the `property_path` option to `false`

date Field Type

A field that allows the user to modify date information via a variety of different HTML elements.

The underlying data used for this field type can be a `DateTime` object, a string, a timestamp or an array. As long as the *input* option is set correctly, the field will take care of all of the details.

The field can be rendered as a single text box, three text boxes (month, day, and year) or three select boxes (see the *widget_* option).

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	single text box or three select fields
Options	• *widget* • *input* • *empty_value* • *years* • *months* • *days* • *format* • *pattern* • *data_timezone* • *user_timezone*
Parent type	`field` (if text), `form` otherwise
Class	`Symfony\Component\Form\Extension\Core\Type\DateTyp`

Basic Usage

This field type is highly configurable, but easy to use. The most important options are `input` and `widget`.

Suppose that you have a `publishedAt` field whose underlying date is a `DateTime` object. The following configures the `date` type for that field as three different choice fields:

```
$builder->add('publishedAt', 'date', array(
    'input'  => 'datetime',
    'widget' => 'choice',
));
```

The `input` option *must* be changed to match the type of the underlying date data. For example, if the `publishedAt` field's data were a unix timestamp, you'd need to set `input` to `timestamp`:

```
$builder->add('publishedAt', 'date', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

The field also supports an `array` and `string` as valid `input` option values.

Field Options

widget **type**: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders three select inputs. The order of the selects is defined in the *pattern* option.
- `text`: renders a three field input of type text (month, day, year).
- `single_text`: renders a single input of type text. User's input is validated based on the *format* option.

input **type**: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05`)

- `datetime` (a `DateTime` object)
- `array` (e.g. `array('year' => 2011, 'month' => 06, 'day' => 05)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

empty_value **type**: `string` or `array`

If your widget option is set to `choice`, then this field will be represented as a series of `select` boxes. The `empty_value` option can be used to add a "blank" entry to the top of each select box:

```
$builder->add('dueDate', 'date', array(
    'empty_value' => '',
));
```

Alternatively, you can specify a string to be displayed for the "blank" value:

```
$builder->add('dueDate', 'date', array(
    'empty_value' => array('year' => 'Year', 'month' => 'Month', 'day' => 'Day')
));
```

years **type**: `array` **default**: five years before to five years after the current year

List of years available to the year field type. This option is only relevant when the `widget` option is set to `choice`.

months **type**: `array` **default**: 1 to 12

List of months available to the month field type. This option is only relevant when the `widget` option is set to `choice`.

days **type**: `array` **default**: 1 to 31

List of days available to the day field type. This option is only relevant when the `widget` option is set to `choice`:

```
'days' => range(1,31)
```

format **type**: `integer` or `string` **default**: `IntlDateFormatter::MEDIUM`

Option passed to the `IntlDateFormatter` class, used to transform user input into the proper format. This is critical when the *widget* option is set to `single_text`, and will define how the user will input the data. By default, the format is determined based on the current user locale; you can override it by passing the format as a string.

For more information on valid formats, see Date/Time Format Syntax. For example, to render a single text box that expects the user to end `yyyy-MM-dd`, use the following options:

```
$builder->add('date_created', 'date', array(
    'widget' => 'single_text',
    'format' => 'yyyy-MM-dd',
));
```

pattern **type**: `string`

This option is only relevant when the *widget* is set to `choice`. The default pattern is based off the *format* option, and tries to match the characters `M`, `d`, and `y` in the format pattern. If no match is found, the default is the string `{{ year }}-{{ month }}-{{ day }}`. Tokens for this option include:

- `{{ year }}`: Replaced with the `year` widget
- `{{ month }}`: Replaced with the `month` widget
- `{{ day }}`: Replaced with the `day` widget

data_timezone **type**: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the PHP supported timezones

user_timezone **type**: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the PHP supported timezones

datetime Field Type

This field type allows the user to modify data that represents a specific date and time (e.g. `1984-06-05 12:15:30`).

Can be rendered as a text input or select tags. The underlying format of the data can be a `DateTime` object, a string, a timestamp or an array.

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	single text box or three select fields
Options	• *date_widget* • *time_widget* • *input* • *date_format* • *hours* • *minutes* • *seconds* • *years* • *months* • *days* • *with_seconds* • *data_timezone* • *user_timezone*
Parent type	form
Class	`Symfony\Component\Form\Extension\Core\Type\DateTim`

Field Options

date_widget **type**: `string` **default**: `choice`

Defines the `widget` option for the date type

time_widget **type**: `string` **default**: `choice`

Defines the `widget` option for the time type

input **type**: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `2011-06-05 12:15:00`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array(2011, 06, 05, 12, 15, 0)`)
- `timestamp` (e.g. `1307276100`)

The value that comes back from the form will also be normalized back into this format.

date_format **type**: `integer` or `string` **default**: `IntlDateFormatter::MEDIUM`

Defines the `format` option that will be passed down to the date field.

hours **type**: `integer` **default**: 1 to 23

List of hours available to the hours field type. This option is only relevant when the `widget` option is set to `choice`.

minutes **type**: `integer` **default**: 1 to 59

List of minutes available to the minutes field type. This option is only relevant when the `widget` option is set to `choice`.

seconds **type**: `integer` **default**: 1 to 59

List of seconds available to the seconds field type. This option is only relevant when the `widget` option is set to `choice`.

years **type**: `array` **default**: five years before to five years after the current year

List of years available to the year field type. This option is only relevant when the `widget` option is set to `choice`.

months **type**: `array` **default**: 1 to 12

List of months available to the month field type. This option is only relevant when the `widget` option is set to `choice`.

days **type**: `array` **default**: 1 to 31

List of days available to the day field type. This option is only relevant when the `widget` option is set to `choice`:

```
'days' => range(1,31)
```

with_seconds **type**: `Boolean` **default**: `false`

Whether or not to include seconds in the input. This will result in an additional input to capture seconds.

data_timezone **type**: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the PHP supported timezones

user_timezone **type**: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the PHP supported timezones

email Field Type

The `email` field is a text field that is rendered using the HTML5 `<input type="email" />` tag.

Rendered as	`input email` field (a text box)
Inherited options	• *max_length* • *required* • *label* • *trim* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\EmailTy`

Inherited Options

These options inherit from the field type:

max_length **type**: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

trim **type**: `Boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the `trim()` function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

entity Field Type

A special `choice` field that's designed to load options from a Doctrine entity. For example, if you have a `Category` entity, you could use this field to display a `select` field of all, or some, of the `Category` objects from the database.

Rendered as	can be various tags (see *Select tag, Checkboxes or Radio Buttons*)
Options	• *class* • *property* • *query_builder* • *em*
Inherited options	• *required* • *label* • *multiple* • *expanded* • *preferred_choices* • *empty_value* • *read_only* • *error_bubbling*
Parent type	choice
Class	`Symfony\Bridge\Doctrine\Form\Type\EntityType`

Basic Usage

The `entity` type has just one required option: the entity which should be listed inside the choice field:

```
$builder->add('users', 'entity', array(
    'class' => 'AcmeHelloBundle:User',
));
```

In this case, all `User` objects will be loaded from the database and rendered as either a `select` tag, a set or radio buttons or a series of checkboxes (this depends on the `multiple` and `expanded` values).

Using a Custom Query for the Entities If you need to specify a custom query to use when fetching the entities (e.g. you only want to return some entities, or need to order them), use the `query_builder` option. The easiest way to use the option is as follows:

```
use Doctrine\ORM\EntityRepository;
// ...

$builder->add('users', 'entity', array(
```

```
    'class' => 'AcmeHelloBundle:User',
    'query_builder' => function(EntityRepository $er) {
        return $er->createQueryBuilder('u')
            ->orderBy('u.username', 'ASC');
    },
));
```

Select tag, Checkboxes or Radio Buttons

This field may be rendered as one of several different HTML fields, depending on the `expanded` and `multiple` options:

element type	expanded	multiple
select tag	false	false
select tag (with `multiple` attribute)	false	true
radio buttons	true	false
checkboxes	true	true

Field Options

class **type**: `string` **required**

The class of your entity (e.g. `AcmeStoreBundle:Category`). This can be a fully-qualified class name (e.g. `Acme\StoreBundle\Entity\Category`) or the short alias name (as shown prior).

property **type**: `string`

This is the property that should be used for displaying the entities as text in the HTML element. If left blank, the entity object will be cast into a string and so must have a `__toString()` method.

query_builder **type**: `Doctrine\ORM\QueryBuilder` or a Closure

If specified, this is used to query the subset of options (and their order) that should be used for the field. The value of this option can either be a `QueryBuilder` object or a Closure. If using a Closure, it should take a single argument, which is the `EntityRepository` of the entity.

em **type**: `string` **default**: the default entity manager

If specified, the specified entity manager will be used to load the choices instead of the default entity manager.

Inherited options

These options inherit from the choice type:

multiple **type**: `Boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

expanded **type**: `Boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

preferred_choices **type**: `array` **default**: `array()`

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `-------------------`). This can be customized when rendering the field:

- *Twig*

```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['foo_choices'], array('separator' => '=====')) ?>
```

empty_value **type**: `string` or `Boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if both the `expanded` and `multiple` options are set to false.

- Add an empty value with "Choose an option" as the text:

```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

file Field Type

The `file` type represents a file input in your form.

Rendered as	`input file` field
Inherited options	• *required* • *label* • *read_only* • *error_bubbling*
Parent type	form
Class	`Symfony\Component\Form\Extension\Core\Type\FileTyp`

Basic Usage

Let's say you have this form definition:

```
$builder->add('attachment', 'file');
```

Caution: Don't forget to add the `enctype` attribute in the form tag: `<form action="#" method="post" {{ form_enctype(form) }}>`.

When the form is submitted, the `attachment` field will be an instance of `Symfony\Component\HttpFoundation\File\UploadedFile`. It can be used to move the `attachment` file to a permanent location:

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

public function uploadAction()
{
    // ...

    if ($form->isValid()) {
        $someNewFilename = ...
```

```
        $form['attachment']->getData()->move($dir, $someNewFilename);

        // ...
    }

    // ...
}
```

The `move()` method takes a directory and a file name as its arguments. You might calculate the filename in one of the following ways:

```
// use the original file name
$file->move($dir, $file->getClientOriginalName());

// compute a random name and try to guess the extension (more secure)
$extension = $file->guessExtension();
if (!$extension) {
    // extension cannot be guessed
    $extension = 'bin';
}
$file->move($dir, rand(1, 99999).'.'.$extension);
```

Using the original name via `getClientOriginalName()` is not safe as it could have been manipulated by the end-user. Moreover, it can contain characters that are not allowed in file names. You should sanitize the name before using it directly.

Read the cookbook for an example of how to manage a file upload associated with a Doctrine entity.

Inherited options

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

The Abstract "field" Type

The `field` form type is not an actual field type you use, but rather functions as the parent field type for many other fields.

The `field` type predefines a couple of options:

data

type: mixed **default**: Defaults to field of the underlying object (if there is one)

When you create a form, each field initially displays the value of the corresponding property of the form's domain object (if an object is bound to the form). If you want to override the initial value for the form or just an individual field, you can set it in the data option:

```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

required

type: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

- `disabled` [type: Boolean, default: false] If you don't want a user to modify the value of a field, you can set the disabled option to true. Any submitted value will be ignored.

```
    use Symfony\Component\Form\TextField

    $field = new TextField('status', array(
        'data' => 'Old data',
        'disabled' => true,
    ));
    $field->submit('New data');

    // prints "Old data"
    echo $field->getData();
```

trim

type: `Boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the `trim()` function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

property_path

type: `any` **default**: `the field's value`

Fields display a property value of the form's domain object by default. When the form is submitted, the submitted value is written back into the object.

If you want to override the property that a field reads from and writes to, you can set the `property_path` option. Its default value is the field's name.

If you wish the field to be ignored when reading or writing to the object you can set the `property_path` option to `false`

attr

type: array **default**: Empty array

If you want to add extra attributes to HTML field representation you can use `attr` option. It's an associative array with HTML attribute as a key. This can be useful when you need to set a custom class for some widget:

```
$builder->add('body', 'textarea', array(
    'attr' => array('class' => 'tinymce'),
));
```

translation_domain

type: `string` **default**: `messages`

This is the translation domain that will be used for any labels or options that are rendered for this field.

form Field Type

See `Symfony\Component\Form\Extension\Core\Type\FormType`.

hidden Field Type

The hidden type represents a hidden input field.

Rendered as	`input hidden` field
Inherited options	• `data` • `property_path`
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\HiddenT`

Inherited Options

These options inherit from the field type:

data **type**: mixed **default**: Defaults to field of the underlying object (if there is one)

When you create a form, each field initially displays the value of the corresponding property of the form's domain object (if an object is bound to the form). If you want to override the initial value for the form or just an individual field, you can set it in the data option:

```
$builder->add('token', 'hidden', array(
    'data' => 'abcdef',
));
```

property_path **type**: `any` **default**: `the field's value`

Fields display a property value of the form's domain object by default. When the form is submitted, the submitted value is written back into the object.

If you want to override the property that a field reads from and writes to, you can set the `property_path` option. Its default value is the field's name.

If you wish the field to be ignored when reading or writing to the object you can set the `property_path` option to `false`

integer Field Type

Renders an input "number" field. Basically, this is a text field that's good at handling data that's in an integer form. The input `number` field looks like a text box, except that - if the user's browser supports HTML5 - it will have some extra frontend functionality.

This field has different options on how to handle input values that aren't integers. By default, all non-integer values (e.g. 6.78) will round down (e.g. 6).

Rendered as	`input text` field
Options	• *rounding_mode* • *grouping*
Inherited options	• *required* • *label* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\Integer`

Field Options

rounding_mode **type**: `integer` **default**: `IntegerToLocalizedStringTransformer::ROUND_DOWN`

By default, if the user enters a non-integer number, it will be rounded down. There are several other rounding methods, and each is a constant on the `Symfony\Component\Form\Extension\Core\DataTransformer\IntegerToLocalizedStringTransformer`:

- `IntegerToLocalizedStringTransformer::ROUND_DOWN` Rounding mode to round towards zero.
- `IntegerToLocalizedStringTransformer::ROUND_FLOOR` Rounding mode to round towards negative infinity.
- `IntegerToLocalizedStringTransformer::ROUND_UP` Rounding mode to round away from zero.
- `IntegerToLocalizedStringTransformer::ROUND_CEILING` Rounding mode to round towards positive infinity.

grouping **type**: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class. Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

Inherited options

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

language Field Type

The `language` type is a subset of the `ChoiceType` that allows the user to select from a large list of languages. As an added bonus, the language names are displayed in the language of the user.

The "value" for each locale is either the two letter ISO639-1 *language* code (e.g. `fr`).

Note: The locale of your user is guessed using Locale::getDefault()

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of languages. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select tag, Checkboxes or Radio Buttons*)
Inherited options	• *multiple* • *expanded* • *preferred_choices* • *empty_value* • *error_bubbling* • *required* • *label* • *read_only*
Parent type	choice
Class	`Symfony\Component\Form\Extension\Core\Type\Languag`

Inherited Options

These options inherit from the choice type:

multiple **type**: `Boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

expanded **type**: `Boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

preferred_choices **type**: `array` **default**: `array()`

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

- *Twig*

```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['foo_choices'], array('separator' => '=====')) ?>
```

empty_value **type**: `string` or `Boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if both the `expanded` and `multiple` options are set to false.

- Add an empty value with "Choose an option" as the text:

```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

locale Field Type

The `locale` type is a subset of the `ChoiceType` that allows the user to select from a large list of locales (language+country). As an added bonus, the locale names are displayed in the language of the user.

The "value" for each locale is either the two letter ISO639-1 *language* code (e.g. `fr`), or the language code followed by an underscore (_), then the ISO3166 *country* code (e.g. `fr_FR` for French/France).

Note: The locale of your user is guessed using Locale::getDefault()

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of locales. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select tag, Checkboxes or Radio Buttons*)
Inherited options	• *multiple* • *expanded* • *preferred_choices* • *empty_value* • *error_bubbling* • *required* • *label* • *read_only*
Parent type	choice
Class	`Symfony\Component\Form\Extension\Core\Type\Languag`

Inherited options

These options inherit from the choice type:

multiple **type**: `Boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

expanded **type**: `Boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

preferred_choices **type**: `array` **default**: `array()`

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `----------`). This can be customized when rendering the field:

- *Twig*

```
    {{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['foo_choices'], array('separator' => '=====')) ?>
```

empty_value **type**: `string` or `Boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if both the `expanded` and `multiple` options are set to false.

- Add an empty value with "Choose an option" as the text:

```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

money Field Type

Renders an input text field and specializes in handling submitted "money" data.

This field type allows you to specify a currency, whose symbol is rendered next to the text field. There are also several other options for customizing how the input and output of the data is handled.

Rendered as	`input text` field
Options	• *currency* • *divisor* • *precision* • *grouping*
Inherited options	• *required* • *label* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\MoneyTy`

Field Options

currency **type**: `string` **default**: `EUR`

Specifies the currency that the money is being specified in. This determines the currency symbol that should be shown by the text box. Depending on the currency - the currency symbol may be shown before or after the input text field.

This can also be set to false to hide the currency symbol.

divisor **type**: `integer` **default**: `1`

If, for some reason, you need to divide your starting value by a number before rendering it to the user, you can use the `divisor` option. For example:

```
$builder->add('price', 'money', array(
    'divisor' => 100,
));
```

In this case, if the `price` field is set to `9900`, then the value `99` will actually be rendered to the user. When the user submits the value `99`, it will be multiplied by `100` and `9900` will ultimately be set back on your object.

precision **type**: `integer` **default**: `2`

For some reason, if you need some precision other than 2 decimal places, you can modify this value. You probably won't need to do this unless, for example, you want to round to the nearest dollar (set the precision to `0`).

grouping **type**: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class. Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

Inherited Options

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

number Field Type

Renders an input text field and specializes in handling number input. This type offers different options for the precision, rounding, and grouping that you want to use for your number.

Rendered as	`input text` field
Options	• *rounding_mode* • *precision* • *grouping*
Inherited options	• *required* • *label* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\NumberT`

Field Options

precision **type**: `integer` **default**: Locale-specific (usually around `3`)

This specifies how many decimals will be allowed until the field rounds the submitted value (via `rounding_mode`). For example, if `precision` is set to `2`, a submitted value of `20.123` will be rounded to, for example, `20.12` (depending on your `rounding_mode`).

rounding_mode **type**: `integer` **default**: `IntegerToLocalizedStringTransformer::ROUND_HALFUP`

If a submitted number needs to be rounded (based on the `precision` option), you have several configurable options for that rounding. Each option is a constant on the `Symfony\Component\Form\Extension\Core\DataTransformer\IntegerToLocalizedStringTransformer`:

- `IntegerToLocalizedStringTransformer::ROUND_DOWN` Rounding mode to round towards zero.
- `IntegerToLocalizedStringTransformer::ROUND_FLOOR` Rounding mode to round towards negative infinity.
- `IntegerToLocalizedStringTransformer::ROUND_UP` Rounding mode to round away from zero.
- `IntegerToLocalizedStringTransformer::ROUND_CEILING` Rounding mode to round towards positive infinity.
- `IntegerToLocalizedStringTransformer::ROUND_HALFDOWN` Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down.
- `IntegerToLocalizedStringTransformer::ROUND_HALFEVEN` Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor.
- `IntegerToLocalizedStringTransformer::ROUND_HALFUP` Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up.

grouping **type**: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class. Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

Inherited Options

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

password Field Type

The `password` field renders an input password text box.

Rendered as	`input password` field
Options	• *always_empty*
Inherited options	• *max_length* • *required* • *label* • *trim* • *read_only* • *error_bubbling*
Parent type	text
Class	`Symfony\Component\Form\Extension\Core\Type\Passwor`

Field Options

always_empty **type**: `Boolean` **default**: `true`

If set to true, the field will *always* render blank, even if the corresponding field has a value. When set to false, the password field will be rendered with the `value` attribute set to its true value.

Put simply, if for some reason you want to render your password field *with* the password value already entered into the box, set this to false.

Inherited Options

These options inherit from the field type:

max_length **type**: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

trim **type**: `Boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the `trim()` function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

percent Field Type

The `percent` type renders an input text field and specializes in handling percentage data. If your percentage data is stored as a decimal (e.g. `.95`), you can use this field out-of-the-box. If you store your data as a number (e.g. `95`), you should set the `type` option to `integer`.

This field adds a percentage sign "`%`" after the input box.

Rendered as	`input text` field
Options	• *type* • *precision*
Inherited options	• *required* • *label* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\Percent`

Options

type **type**: `string` **default**: `fractional`

This controls how your data is stored on your object. For example, a percentage corresponding to "55%", might be stored as `.55` or `55` on your object. The two "types" handle these two cases:

- `fractional` If your data is stored as a decimal (e.g. `.55`), use this type. The data will be multiplied by `100` before being shown to the user (e.g. `55`). The submitted data will be divided by `100` on form submit so that the decimal value is stored (`.55`);
- `integer` If your data is stored as an integer (e.g. 55), then use this option. The raw value (`55`) is shown to the user and stored on your object. Note that this only works for integer values.

precision **type**: `integer` **default**: `0`

By default, the input numbers are are rounded. To allow for more decimal places, use this option.

Inherited Options

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

radio Field Type

Creates a single radio button. This should always be used for a field that has a Boolean value: if the radio button is selected, the field will be set to true, if the button is not selected, the value will be set to false.

The `radio` type isn't usually used directly. More commonly it's used internally by other types such as choice. If you want to have a Boolean field, use checkbox.

Rendered as	`input radio` field
Options	• *value*
Inherited options	• *required* • *label* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\RadioTy`

Field Options

value **type**: `mixed` **default**: `1`

The value that's actually used as the value for the radio button. This does not affect the value that's set on your object.

Inherited Options

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

repeated Field Type

This is a special field "group", that creates two identical fields whose values must match (or a validation error is thrown). The most common use is when you need the user to repeat his or her password or email to verify accuracy.

Rendered as	input `text` field by default, but see *type* option
Options	• *type* • *options* • *first_name* • *second_name*
Inherited options	• *invalid_message* • *invalid_message_parameters* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\Repeate`

Example Usage

```
$builder->add('password', 'repeated', array(
    'type' => 'password',
    'invalid_message' => 'The password fields must match.',
```

```
    'options' => array('label' => 'Password'),
));
```

Upon a successful form submit, the value entered into both of the "password" fields becomes the data of the `password` key. In other words, even though two fields are actually rendered, the end data from the form is just the single value (usually a string) that you need.

The most important option is `type`, which can be any field type and determines the actual type of the two underlying fields. The `options` option is passed to each of those individual fields, meaning - in this example - any option supported by the `password` type can be passed in this array.

Validation One of the key features of the `repeated` field is internal validation (you don't need to do anything to set this up) that forces the two fields to have a matching value. If the two fields don't match, an error will be shown to the user.

The `invalid_message` is used to customize the error that will be displayed when the two fields do not match each other.

Field Options

type **type**: `string` **default**: `text`

The two underlying fields will be of this field type. For example, passing a type of `password` will render two password fields.

options **type**: `array` **default**: `array()`

This options array will be passed to each of the two underlying fields. In other words, these are the options that customize the individual field types. For example, if the `type` option is set to `password`, this array might contain the options `always_empty` or `required` - both options that are supported by the `password` field type.

first_name **type**: `string` **default**: `first`

This is the actual field name to be used for the first field. This is mostly meaningless, however, as the actual data entered into both of the fields will be available under the key assigned to the `repeated` field itself (e.g. `password`). However, if you don't specify a label, this field name is used to "guess" the label for you.

second_name **type**: `string` **default**: `second`

The same as `first_name`, but for the second field.

Inherited options

These options inherit from the field type:

invalid_message **type**: `string` **default**: `This value is not valid`

This is the validation error message that's used when the data entered is determined by internal validation of a field type. This might happen, for example, if the user enters a string into a time field that cannot be converted into a real time. For normal validation messages (such as when setting a minimum length for a field), set the validation messages with your validation rules (*reference*).

invalid_message_parameters **type**: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

search Field Type

This renders an `<input type="search" />` field, which is a text box with special functionality supported by some browsers.

Read about the input search field at DiveIntoHTML5.info

Rendered as	`input search` field
Inherited options	• *max_length* • *required* • *label* • *trim* • *read_only* • *error_bubbling*
Parent type	text
Class	`Symfony\Component\Form\Extension\Core\Type\SearchT`

Inherited Options

These options inherit from the field type:

max_length **type**: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

trim **type**: `Boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the `trim()` function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

text Field Type

The text field represents the most basic input text field.

Rendered as	`input text` field
Inherited options	• *max_length* • *required* • *label* • *trim* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\TextTyp`

Inherited Options

These options inherit from the field type:

max_length **type**: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

trim **type**: `Boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the `trim()` function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

textarea Field Type

Renders a `textarea` HTML element.

Rendered as	`textarea` tag
Inherited options	• *max_length* • *required* • *label* • *trim* • *read_only* • *error_bubbling*
Parent type	field
Class	`Symfony\Component\Form\Extension\Core\Type\Textare`

Inherited Options

These options inherit from the field type:

max_length **type**: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

trim **type**: `Boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the `trim()` function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

time Field Type

A field to capture time input.

This can be rendered as a text field, a series of text fields (e.g. hour, minute, second) or a series of select fields. The underlying data can be stored as a `DateTime` object, a string, a timestamp or an array.

Underlying Data Type	can be `DateTime`, string, timestamp, or array (see the `input` option)
Rendered as	can be various tags (see below)
Options	• *widget* • *input* • *with_seconds* • *hours* • *minutes* • *seconds* • *data_timezone* • *user_timezone*
Parent type	form
Class	`Symfony\Component\Form\Extension\Core\Type\TimeTyp`

Basic Usage

This field type is highly configurable, but easy to use. The most important options are `input` and `widget`.

Suppose that you have a `startTime` field whose underlying time data is a `DateTime` object. The following configures the `time` type for that field as three different choice fields:

```
$builder->add('startTime', 'time', array(
    'input'  => 'datetime',
    'widget' => 'choice',
));
```

The `input` option *must* be changed to match the type of the underlying date data. For example, if the `startTime` field's data were a unix timestamp, you'd need to set `input` to `timestamp`:

```
$builder->add('startTime', 'time', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

The field also supports an `array` and `string` as valid `input` option values.

Field Options

widget **type**: `string` **default**: `choice`

The basic way in which this field should be rendered. Can be one of the following:

- `choice`: renders two (or three if *with_seconds* is true) select inputs.
- `text`: renders a two or three text inputs (hour, minute, second).
- `single_text`: renders a single input of type text. User's input will be validated against the form `hh:mm` (or `hh:mm:ss` if using seconds).

input **type**: `string` **default**: `datetime`

The format of the *input* data - i.e. the format that the date is stored on your underlying object. Valid values are:

- `string` (e.g. `12:17:26`)
- `datetime` (a `DateTime` object)
- `array` (e.g. `array('hour' => 12, 'minute' => 17, 'second' => 26)`)
- `timestamp` (e.g. `1307232000`)

The value that comes back from the form will also be normalized back into this format.

with_seconds **type**: `Boolean` **default**: `false`

Whether or not to include seconds in the input. This will result in an additional input to capture seconds.

hours **type**: `integer` **default**: 1 to 23

List of hours available to the hours field type. This option is only relevant when the `widget` option is set to `choice`.

minutes **type**: `integer` **default**: 1 to 59

List of minutes available to the minutes field type. This option is only relevant when the `widget` option is set to `choice`.

seconds **type**: `integer` **default**: 1 to 59

List of seconds available to the seconds field type. This option is only relevant when the `widget` option is set to `choice`.

data_timezone **type**: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the PHP supported timezones

user_timezone **type**: `string` **default**: system default timezone

Timezone for how the data should be shown to the user (and therefore also the data that the user submits). This must be one of the PHP supported timezones

timezone Field Type

The `timezone` type is a subset of the `ChoiceType` that allows the user to select from all possible timezones.

The "value" for each timezone is the full timezone name, such as `America/Chicago` or `Europe/Istanbul`.

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of locales. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Rendered as	can be various tags (see *Select tag, Checkboxes or Radio Buttons*)
Inherited options	• *multiple* • *expanded* • *preferred_choices* • *empty_value* • *error_bubbling* • *required* • *label* • *read_only*
Parent type	choice
Class	`Symfony\Component\Form\Extension\Core\Type\Timezon`

Inherited options

These options inherit from the choice type:

multiple **type**: `Boolean` **default**: `false`

If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the `expanded` option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false. The returned value will be an array.

expanded **type**: `Boolean` **default**: `false`

If set to true, radio buttons or checkboxes will be rendered (depending on the `multiple` value). If false, a select element will be rendered.

preferred_choices **type**: `array` **default**: `array()`

If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu. The following would move the "Baz" option to the top, with a visual separator between it and the rest of the options:

```
$builder->add('foo_choices', 'choice', array(
    'choices' => array('foo' => 'Foo', 'bar' => 'Bar', 'baz' => 'Baz'),
    'preferred_choices' => array('baz'),
));
```

Note that preferred choices are only meaningful when rendering as a `select` element (i.e. `expanded` is false). The preferred choices and normal choices are separated visually by a set of dotted lines (i.e. `-------------------`). This can be customized when rendering the field:

- *Twig*

```
{{ form_widget(form.foo_choices, { 'separator': '=====' }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['foo_choices'], array('separator' => '=====')) ?>
```

empty_value **type**: `string` or `Boolean`

This option determines whether or not a special "empty" option (e.g. "Choose an option") will appear at the top of a select widget. This option only applies if both the `expanded` and `multiple` options are set to false.

- Add an empty value with "Choose an option" as the text:

```
$builder->add('states', 'choice', array(
    'empty_value' => 'Choose an option',
));
```

- Guarantee that no "empty" value option is displayed:

```
$builder->add('states', 'choice', array(
    'empty_value' => false,
));
```

If you leave the `empty_value` option unset, then a blank (with no text) option will automatically be added if and only if the `required` option is false:

```
// a blank (with no text) option will be added
$builder->add('states', 'choice', array(
    'required' => false,
));
```

These options inherit from the field type:

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

url Field Type

The `url` field is a text field that prepends the submitted value with a given protocol (e.g. `http://`) if the submitted value doesn't already have a protocol.

Rendered as	`input url` field
Options	• *default_protocol*
Inherited options	• *max_length* • *required* • *label* • *trim* • *read_only* • *error_bubbling*
Parent type	text
Class	`Symfony\Component\Form\Extension\Core\Type\UrlType`

Field Options

default_protocol **type**: `string` **default**: `http`

If a value is submitted that doesn't begin with some protocol (e.g. `http://`, `ftp://`, etc), this protocol will be prepended to the string when the data is bound to the form.

Inherited Options

These options inherit from the field type:

max_length **type**: `integer`

This option is used to add a `max_length` attribute, which is used by some browsers to limit the amount of text in a field.

required **type**: `Boolean` **default**: `true`

If true, an HTML5 required attribute will be rendered. The corresponding `label` will also render with a `required` class.

This is superficial and independent from validation. At best, if you let Symfony guess your field type, then the value of this option will be guessed from your validation information.

label **type**: `string` **default**: The label is "guessed" from the field name

Sets the label that will be used when rendering the field. The label can also be directly set inside the template:

```
{{ form_label(form.name, 'Your name') }}
```

trim **type**: `Boolean` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the `trim()` function when the data is bound. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is merged back onto the underlying object.

read_only **type**: `Boolean` **default**: `false`

If this option is true, the field will be rendered with the `disabled` attribute so that the field is not editable.

error_bubbling **type**: `Boolean` **default**: `false`

If true, any errors for this field will be passed to the parent field or form. For example, if set to true on a normal field, any errors for that field will be attached to the main form, not to the specific field.

A form is composed of *fields*, each of which are built with the help of a field *type* (e.g. a `text` type, `choice` type, etc). Symfony2 comes standard with a large list of field types that can be used in your application.

Supported Field Types

The following field types are natively available in Symfony2:

Text Fields

- text
- textarea
- email
- integer
- money
- number
- password
- percent
- search
- url

Choice Fields

- choice
- entity
- country
- language
- locale

- timezone

Date and Time Fields

- date
- datetime
- time
- birthday

Other Fields

- checkbox
- file
- radio

Field Groups

- collection
- repeated

Hidden Fields

- hidden
- csrf

Base Fields

- field
- form

5.1.10 Twig Template Form Function Reference

This reference manual covers all the possible Twig functions available for rendering forms. There are several different functions available, and each is responsible for rendering a different part of a form (e.g. labels, errors, widgets, etc).

form_label(form.name, label, variables)

Renders the label for the given field. You can optionally pass the specific label you want to display as the second argument.

```
{{ form_label(form.name) }}

{# The two following syntaxes are equivalent #}
{{ form_label(form.name, 'Your Name', { 'attr': {'class': 'foo'} }) }}
{{ form_label(form.name, null, { 'label': 'Your name', 'attr': {'class': 'foo'} }) }}
```

form_errors(form.name)

Renders any errors for the given field.

```
{{ form_errors(form.name) }}

{# render any "global" errors #}
{{ form_errors(form) }}
```

form_widget(form.name, variables)

Renders the HTML widget of a given field. If you apply this to an entire form or collection of fields, each underlying form row will be rendered.

```
{# render a widget, but add a "foo" class to it #}
{{ form_widget(form.name, { 'attr': {'class': 'foo'} }) }}
```

The second argument to `form_widget` is an array of variables. The most common variable is `attr`, which is an array of HTML attributes to apply to the HTML widget. In some cases, certain types also have other template-related options that can be passed. These are discussed on a type-by-type basis.

form_row(form.name, variables)

Renders the "row" of a given field, which is the combination of the field's label, errors and widget.

```
{# render a field row, but display a label with text "foo" #}
{{ form_row(form.name, { 'label': 'foo' }) }}
```

The second argument to `form_row` is an array of variables. The templates provided in Symfony only allow to override the label as shown in the example above.

form_rest(form, variables)

This renders all fields that have not yet been rendered for the given form. It's a good idea to always have this somewhere inside your form as it'll render hidden fields for you and make any fields you forgot to render more obvious (since it'll render the field for you).

```
{{ form_rest(form) }}
```

form_enctype(form)

If the form contains at least one file upload field, this will render the required `enctype="multipart/form-data"` form attribute. It's always a good idea to include this in your form tag:

```
<form action="{{ path('form_submit') }}" method="post" {{ form_enctype(form) }}>
```

5.1.11 Validation Constraints Reference

NotBlank

Validates that a value is not blank, defined as not equal to a blank string and also not equal to `null`. To force that a value is simply not equal to `null`, see the NotNull constraint.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\NotBlank`
Validator	`Symfony\Component\Validator\Constraints\NotBlankVa`

Basic Usage

If you wanted to ensure that the `firstName` property of an `Author` class were not blank, you could do the following:

- *YAML*

```
properties:
    firstName:
        - NotBlank: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     */
    protected $firstName;
}
```

Options

message **type**: `string` **default**: `This value should not be blank`

This is the message that will be shown if the value is blank.

Blank

Validates that a value is blank, defined as equal to a blank string or equal to `null`. To force that a value strictly be equal to `null`, see the Null constraint. To force that a value is *not* blank, see NotBlank.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\Blank`
Validator	`Symfony\Component\Validator\Constraints\BlankValid`

Basic Usage

If, for some reason, you wanted to ensure that the `firstName` property of an `Author` class were blank, you could do the following:

- *YAML*

```
    properties:
        firstName:
            - Blank: ~
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        /**
         * @Assert\Blank()
         */
        protected $firstName;
    }
```

Options

message **type**: `string` **default**: `This value should be blank`

This is the message that will be shown if the value is not blank.

NotNull

Validates that a value is not strictly equal to `null`. To ensure that a value is simply not blank (not a blank string), see the NotBlank constraint.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\NotNull`
Validator	`Symfony\Component\Validator\Constraints\NotNullVal`

Basic Usage

If you wanted to ensure that the `firstName` property of an `Author` class were not strictly equal to `null`, you would:

- *YAML*

```
    properties:
        firstName:
            - NotNull: ~
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        /**
         * @Assert\NotNull()
         */
```

```
        protected $firstName;
    }
```

Options

message **type**: `string` **default**: `This value should not be null`

This is the message that will be shown if the value is `null`.

Null

Validates that a value is exactly equal to `null`. To force that a property is simply blank (blank string or `null`), see the Blank constraint. To ensure that a property is not null, see NotNull.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\Null`
Validator	`Symfony\Component\Validator\Constraints\NullValida`

Basic Usage

If, for some reason, you wanted to ensure that the `firstName` property of an `Author` class exactly equal to `null`, you could do the following:

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Author:
        properties:
            firstName:
                - Null: ~
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    namespace Acme\BlogBundle\Entity;

    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        /**
         * @Assert\Null()
         */
        protected $firstName;
    }
```

Options

message **type**: `string` **default**: `This value should be null`

This is the message that will be shown if the value is not `null`.

True

Validates that a value is `true`. Specifically, this checks to see if the value is exactly `true`, exactly the integer `1`, or exactly the string "`1`".

Also see False.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\True`
Validator	`Symfony\Component\Validator\Constraints\TrueValida`

Basic Usage

This constraint can be applied to properties (e.g. a `termsAccepted` property on a registration model) or to a "getter" method. It's most powerful in the latter case, where you can assert that a method returns a true value. For example, suppose you have the following method:

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    protected $token;

    public function isTokenValid()
    {
        return $this->token == $this->generateToken();
    }
}
```

Then you can constrain this method with `True`.

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Author:
        getters:
            tokenValid:
                - "True": { message: "The token is invalid" }
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        protected $token;

        /**
         * @Assert\True(message = "The token is invalid")
         */
        public function isTokenValid()
        {
            return $this->token == $this->generateToken();
```

```
        }
    }
```

- *XML*

```
    <?xml version="1.0" encoding="UTF-8" ?>
    <!-- src/Acme/Blogbundle/Resources/config/validation.xml -->

    <constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

        <class name="Acme\BlogBundle\Entity\Author">
            <getter property="tokenValid">
                <constraint name="True">
                    <option name="message">The token is invalid...</option>
                </constraint>
            </getter>
        </class>
    </constraint-mapping>
```

- *PHP*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\True;

    class Author
    {
        protected $token;

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addGetterConstraint('tokenValid', new True(array(
                'message' => 'The token is invalid',
            )));
        }

        public function isTokenValid()
        {
            return $this->token == $this->generateToken();
        }
    }
```

If the `isTokenValid()` returns false, the validation will fail.

Options

message **type**: `string` **default**: `This value should be true`

This message is shown if the underlying data is not true.

False

Validates that a value is `false`. Specifically, this checks to see if the value is exactly `false`, exactly the integer `0`, or exactly the string "`0`".

Also see True.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\False`
Validator	`Symfony\Component\Validator\Constraints\FalseValid`

Basic Usage

The `False` constraint can be applied to a property or a "getter" method, but is most commonly useful in the latter case. For example, suppose that you want to guarantee that some `state` property is *not* in a dynamic `invalidStates` array. First, you'd create a "getter" method:

```
protected $state;

protectd $invalidStates = array();

public function isStateInvalid()
{
    return in_array($this->state, $this->invalidStates);
}
```

In this case, the underlying object is only valid if the `isStateInvalid` method returns **false**:

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author
    getters:
        stateInvalid:
            - "False":
                message: You've entered an invalid state.
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\False()
     */
     public function isStateInvalid($message = "You've entered an invalid state.")
     {
        // ...
     }
}
```

Caution: When using YAML, be sure to surround `False` with quotes (`"False"`) or else YAML will convert this into a Boolean value.

Options

message **type**: `string` **default**: `This value should be false`

This message is shown if the underlying data is not false.

Type

Validates that a value is of a specific data type. For example, if a variable should be an array, you can use this constraint with the `array` type option to validate this.

Applies to	*property or method*
Options	• *type* • *message*
Class	`Symfony\Component\Validator\Constraints\Type`
Validator	`Symfony\Component\Validator\Constraints\TypeValida`

Basic Usage

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        age:
            - Type:
                type: integer
                message: The value {{ value }} is not a valid {{ type }}.
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Type(type="integer", message="The value {{ value }} is not a valid {{ type }}.")
     */
     protected $age;
}
```

Options

type **type**: `string` [*default option*]

This required option is the fully qualified class name or one of the PHP datatypes as determined by PHP's `is_` functions.

- array
- bool
- callable
- float

- double
- int
- integer
- long
- null
- numeric
- object
- real
- resource
- scalar
- string

message **type**: `string` **default**: `This value should be of type {{ type }}`

The message if the underlying data is not of the given type.

Email

Validates that a value is a valid email address. The underlying value is cast to a string before being validated.

Applies to	*property or method*
Options	• *message* • *checkMX*
Class	`Symfony\Component\Validator\Constraints\Email`
Validator	`Symfony\Component\Validator\Constraints\EmailValid`

Basic Usage

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        email:
            - Email:
                message: The email "{{ value }}" is not a valid email.
                checkMX: true
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
```

```
         * @Assert\Email(
         *     message = "The email '{{ value }}' is not a valid email.",
         *     checkMX = true
         * )
         */
        protected $email;
    }
```

Options

message **type**: `string` **default**: `This value is not a valid email address`

This message is shown if the underlying data is not a valid email address.

checkMX **type**: `Boolean` **default**: `false`

If true, then the checkdnsrr PHP function will be used to check the validity of the MX record of the host of the given email.

MinLength

Validates that the length of a string is at least as long as the given limit.

Applies to	*property or method*
Options	• *limit* • *message* • *charset*
Class	`Symfony\Component\Validator\Constraints\MinLength`
Validator	`Symfony\Component\Validator\Constraints\MinLengthV`

Basic Usage

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Blog:
        properties:
            firstName:
                - MinLength: { limit: 3, message: "Your name must have at least {{ limit }} characte
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Blog.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Blog
    {
        /**
         * @Assert\MinLength(
         *     limit=3,
         *     message="Your name must have at least {{ limit }} characters."
         * )
```

```
         */
        protected $summary;
    }
```

- *XML*

```
    <!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
    <class name="Acme\BlogBundle\Entity\Blog">
        <property name="summary">
            <constraint name="MinLength">
                <option name="limit">3</option>
                <option name="message">Your name must have at least {{ limit }} characters.</option>
            </constraint>
        </property>
    </class>
```

Options

limit **type**: `integer` [*default option*]

This required option is the "min" value. Validation will fail if the length of the give string is **less** than this number.

message **type**: `string` **default**: `This value is too short. It should have {{ limit }} characters or more`

The message that will be shown if the underlying string has a length that is shorter than the *limit* option.

charset **type**: `charset` **default**: `UTF-8`

If the PHP extension "mbstring" is installed, then the PHP function mb_strlen will be used to calculate the length of the string. The value of the `charset` option is passed as the second argument to that function.

MaxLength

Validates that the length of a string is not larger than the given limit.

Applies to	*property or method*
Options	• *limit* • *message* • *charset*
Class	`Symfony\Component\Validator\Constraints\MaxLength`
Validator	`Symfony\Component\Validator\Constraints\MaxLengthV`

Basic Usage

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Blog:
        properties:
            summary:
                - MaxLength: 100
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Blog.php
use Symfony\Component\Validator\Constraints as Assert;

class Blog
{
    /**
     * @Assert\MaxLength(100)
     */
    protected $summary;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Blog">
    <property name="summary">
        <constraint name="MaxLength">
            <value>100</value>
        </constraint>
    </property>
</class>
```

Options

limit **type**: `integer` [*default option*]

This required option is the "max" value. Validation will fail if the length of the give string is **greater** than this number.

message **type**: `string` **default**: `This value is too long. It should have {{ limit }} characters or less`

The message that will be shown if the underlying string has a length that is longer than the *limit* option.

charset **type**: `charset` **default**: `UTF-8`

If the PHP extension "mbstring" is installed, then the PHP function mb_strlen will be used to calculate the length of the string. The value of the `charset` option is passed as the second argument to that function.

Url

Validates that a value is a valid URL string.

Applies to	*property or method*
Options	• *message* • *protocols*
Class	`Symfony\Component\Validator\Constraints\Url`
Validator	`Symfony\Component\Validator\Constraints\UrlValidat`

Basic Usage

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        bioUrl:
            - Url:
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Url()
     */
     protected $bioUrl;
}
```

Options

message **type**: `string` **default**: `This value is not a valid URL`

This message is shown if the URL is invalid.

protocols **type**: `array` **default**: `array('http', 'https')`

The protocols that will be considered to be valid. For example, if you also needed `ftp://` type URLs to be valid, you'd redefine the `protocols` array, listing `http`, `https`, and also `ftp`.

Regex

Validates that a value matches a regular expression.

Applies to	*property or method*
Options	• *pattern* • *match* • *message*
Class	`Symfony\Component\Validator\Constraints\Regex`
Validator	`Symfony\Component\Validator\Constraints\RegexValid`

Basic Usage

Suppose you have a `description` field and you want to verify that it begins with a valid word character. The regular expression to test for this would be `/^\w+/`, indicating that you're looking for at least one or more word characters at the beginning of your string:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        description:
            - Regex: "/^\w+/"
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Regex("/^\w+/")
     */
    protected $description;
}
```

Alternatively, you can set the *match* option to `false` in order to assert that a given string does *not* match. In the following example, you'll assert that the `firstName` field does not contain any numbers and give it a custom message:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        firstName:
            - Regex:
                pattern: "/\d/"
                match:   false
                message: Your name cannot contain a number
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Regex(
     *     pattern="/\d/",
     *     match=false,
     *     message="Your name cannot contain a number"
     * )
     */
    protected $firstName;
}
```

Options

pattern **type**: `string` [*default option*]

This required option is the regular expression pattern that the input will be matched against. By default, this validator will fail if the input string does *not* match this regular expression (via the preg_match PHP function). However, if *match* is set to false, then validation will fail if the input string *does* match this pattern.

match **type**: `Boolean` default: `true`

If `true` (or not set), this validator will pass if the given string matches the given *pattern* regular expression. However, when this option is set to `false`, the opposite will occur: validation will pass only if the given string does **not** match the *pattern* regular expression.

message **type**: `string` **default**: `This value is not valid`

This is the message that will be shown if this validator fails.

Ip

Validates that a value is a valid IP address. By default, this will validate the value as IPv4, but a number of different options exist to validate as IPv6 and many other combinations.

Applies to	*property or method*
Options	• *version* • *message*
Class	`Symfony\Component\Validator\Constraints\Ip`
Validator	`Symfony\Component\Validator\Constraints\IpValidato`

Basic Usage

- *YAML*

```
# src/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        ipAddress:
            - Ip:
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Ip
     */
     protected $ipAddress;
}
```

Options

version **type**: `string` **default**: `4`

This determines exactly *how* the ip address is validated and can take one of a variety of different values:

All ranges

- `4` - Validates for IPv4 addresses
- `6` - Validates for IPv6 addresses
- `all` - Validates all IP formats

No private ranges

- `4_no_priv` - Validates for IPv4 but without private IP ranges
- `6_no_priv` - Validates for IPv6 but without private IP ranges
- `all_no_priv` - Validates for all IP formats but without private IP ranges

No reserved ranges

- `4_no_res` - Validates for IPv4 but without reserved IP ranges
- `6_no_res` - Validates for IPv6 but without reserved IP ranges
- `all_no_res` - Validates for all IP formats but without reserved IP ranges

Only public ranges

- `4_public` - Validates for IPv4 but without private and reserved ranges
- `6_public` - Validates for IPv6 but without private and reserved ranges
- `all_public` - Validates for all IP formats but without private and reserved ranges

message **type**: `string` **default**: `This is not a valid IP address`

This message is shown if the string is not a valid IP address.

Max

Validates that a given number is *less* than some maximum number.

Applies to	*property or method*
Options	• *limit* • *message* • *invalidMessage*
Class	`Symfony\Component\Validator\Constraints\Max`
Validator	`Symfony\Component\Validator\Constraints\MaxValidat`

Basic Usage

To verify that the "age" field of a class is not greater than "50", you might add the following:

- *YAML*

```
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Participant:
    properties:
        age:
            - Max: { limit: 50, message: You must be 50 or under to enter. }
```

- *Annotations*

```
// src/Acme/EventBundle/Entity/Participant.php
use Symfony\Component\Validator\Constraints as Assert;

class Participant
{
    /**
     * @Assert\Max(limit = 50, message = "You must be 50 or under to enter.")
     */
     protected $age;
}
```

Options

limit **type**: `integer` [*default option*]

This required option is the "max" value. Validation will fail if the given value is **greater** than this max value.

message **type**: `string` **default**: `This value should be {{ limit }} or less`

The message that will be shown if the underlying value is greater than the *limit* option.

invalidMessage **type**: `string` **default**: `This value should be a valid number`

The message that will be shown if the underlying value is not a number (per the is_numeric PHP function).

Min

Validates that a given number is *greater* than some minimum number.

Applies to	*property or method*
Options	• *limit* • *message* • *invalidMessage*
Class	`Symfony\Component\Validator\Constraints\Min`
Validator	`Symfony\Component\Validator\Constraints\MinValidat`

Basic Usage

To verify that the "age" field of a class is "18" or greater, you might add the following:

- *YAML*

```
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Participant:
    properties:
        age:
            - Min: { limit: 18, message: You must be 18 or older to enter. }
```

- *Annotations*

```
// src/Acme/EventBundle/Entity/Participant.php
use Symfony\Component\Validator\Constraints as Assert;

class Participant
{
    /**
     * @Assert\Min(limit = "18", message = "You must be 18 or older to enter")
     */
     protected $age;
}
```

Options

limit **type**: `integer` [*default option*]

This required option is the "min" value. Validation will fail if the given value is **less** than this min value.

message **type**: `string` **default**: `This value should be {{ limit }} or more`

The message that will be shown if the underlying value is less than the *limit* option.

invalidMessage **type**: `string` **default**: `This value should be a valid number`

The message that will be shown if the underlying value is not a number (per the is_numeric PHP function).

Date

Validates that a value is a valid date, meaning either a `DateTime` object or a string (or an object that can be cast into a string) that follows a valid YYYY-MM-DD format.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\Date`
Validator	`Symfony\Component\Validator\Constraints\DateValida`

Basic Usage

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        birthday:
            - Date: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Date()
     */
     protected $birthday;
}
```

Options

message **type**: `string` **default**: `This value is not a valid date`

This message is shown if the underlying data is not a valid date.

DateTime

Validates that a value is a valid "datetime", meaning either a `DateTime` object or a string (or an object that can be cast into a string) that follows a valid YYYY-MM-DD HH:MM:SS format.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\DateTime`
Validator	`Symfony\Component\Validator\Constraints\DateTimeVa`

Basic Usage

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        createdAt:
            - DateTime: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\DateTime()
     */
     protected $createdAt;
}
```

Options

message **type**: `string` **default**: `This value is not a valid datetime`

This message is shown if the underlying data is not a valid datetime.

Time

Validates that a value is a valid time, meaning either a `DateTime` object or a string (or an object that can be cast into a string) that follows a valid "HH:MM:SS" format.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\Time`
Validator	`Symfony\Component\Validator\Constraints\TimeValida`

Basic Usage

Suppose you have an Event class, with a `startAt` field that is the time of the day when the event starts:

- *YAML*

```
# src/Acme/EventBundle/Resources/config/validation.yml
Acme\EventBundle\Entity\Event:
    properties:
        startsAt:
            - Time: ~
```

- *Annotations*

```
// src/Acme/EventBundle/Entity/Event.php
namespace Acme\EventBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Event
{
    /**
     * @Assert\Time()
     */
     protected $startsAt;
}
```

Options

message **type**: `string` **default**: `This value is not a valid time`

This message is shown if the underlying data is not a valid time.

Choice

This constraint is used to ensure that the given value is one of a given set of *valid* choices. It can also be used to validate that each item in an array of items is one of those valid choices.

Applies to	*property or method*
Options	• *choices* • *callback* • *multiple* • *min* • *max* • *message* • *multipleMessage* • *minMessage* • *maxMessage* • *strict*
Class	`Symfony\Component\Validator\Constraints\Choice`
Validator	`Symfony\Component\Validator\Constraints\ChoiceVali`

Basic Usage

The basic idea of this constraint is that you supply it with an array of valid values (this can be done in several ways) and it validates that the value of the given property exists in that array.

If your valid choice list is simple, you can pass them in directly via the *choices* option:

- *YAML*

```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice:
                choices:  [male, female]
                message:  Choose a valid gender.
```

- *XML*

```xml
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\EntityAuthor">
    <property name="gender">
        <constraint name="Choice">
            <option name="choices">
                <value>male</value>
                <value>female</value>
            </option>
            <option name="message">Choose a valid gender.</option>
        </constraint>
    </property>
</class>
```

- *Annotations*

```php
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice(choices = {"male", "female"}, message = "Choose a valid gender.")
     */
```

```
        protected $gender;
    }
```

- *PHP*

```
    // src/Acme/BlogBundle/EntityAuthor.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\Choice;

    class Author
    {
        protected $gender;

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('gender', new Choice(array(
                'choices' => array('male', 'female'),
                'message' => 'Choose a valid gender',
            )));
        }
    }
```

Supplying the Choices with a Callback Function

You can also use a callback function to specify your options. This is useful if you want to keep your choices in some central location so that, for example, you can easily access those choices for validation or for building a select form element.

```
// src/Acme/BlogBundle/Entity/Author.php
class Author
{
    public static function getGenders()
    {
        return array('male', 'female');
    }
}
```

You can pass the name of this method to the *callback_* option of the `Choice` constraint.

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Author:
        properties:
            gender:
                - Choice: { callback: getGenders }
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        /**
         * @Assert\Choice(callback = "getGenders")
         */
```

```
        protected $gender;
    }
```

- *XML*

```
    <!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
    <class name="Acme\BlogBundle\Entity\Author">
        <property name="gender">
            <constraint name="Choice">
                <option name="callback">getGenders</option>
            </constraint>
        </property>
    </class>
```

If the static callback is stored in a different class, for example `Util`, you can pass the class name and the method as an array.

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Author:
        properties:
            gender:
                - Choice: { callback: [Util, getGenders] }
```

- *XML*

```
    <!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
    <class name="Acme\BlogBundle\Entity\Author">
        <property name="gender">
            <constraint name="Choice">
                <option name="callback">
                    <value>Util</value>
                    <value>getGenders</value>
                </option>
            </constraint>
        </property>
    </class>
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        /**
         * @Assert\Choice(callback = {"Util", "getGenders"})
         */
        protected $gender;
    }
```

Available Options

choices **type**: `array` [*default option*]

A required option (unless *callback* is specified) - this is the array of options that should be considered in the valid set. The input value will be matched against this array.

callback **type**: `string|array|Closure`

This is a callback method that can be used instead of the *choices* option to return the choices array. See *Supplying the Choices with a Callback Function* for details on its usage.

multiple **type**: `Boolean` **default**: `false`

If this option is true, the input value is expected to be an array instead of a single, scalar value. The constraint will check that each value of the input array can be found in the array of valid choices. If even one of the input values cannot be found, the validation will fail.

min **type**: `integer`

If the `multiple` option is true, then you can use the `min` option to force at least XX number of values to be selected. For example, if `min` is 3, but the input array only contains 2 valid items, the validation will fail.

max **type**: `integer`

If the `multiple` option is true, then you can use the `max` option to force no more than XX number of values to be selected. For example, if `max` is 3, but the input array contains 4 valid items, the validation will fail.

message **type**: `string` **default**: `The value you selected is not a valid choice`

This is the message that you will receive if the `multiple` option is set to `false`, and the underlying value is not in the valid array of choices.

multipleMessage **type**: `string` **default**: `One or more of the given values is invalid`

This is the message that you will receive if the `multiple` option is set to `true`, and one of the values on the underlying array being checked is not in the array of valid choices.

minMessage **type**: `string` **default**: `You must select at least {{ limit }} choices`

This is the validation error message that's displayed when the user chooses too few choices per the *min* option.

maxMessage **type**: `string` **default**: `You must select at most {{ limit }} choices`

This is the validation error message that's displayed when the user chooses too many options per the *max* option.

strict **type**: `Boolean` **default**: `false`

If true, the validator will also check the type of the input value. Specifically, this value is passed to as the third argument to the PHP in_array method when checking to see if a value is in the valid choices array.

Collection

This constraint is used when the underlying data is a collection (i.e. an array or an object that implements `Traversable` and `ArrayAccess`), but you'd like to validate different keys of that collection in different ways. For example, you might validate the `email` key using the `Email` constraint and the `inventory` key of the collection with the `Min` constraint.

This constraint can also make sure that certain collection keys are present and that extra keys are not present.

Applies to	*property or method*
Options	• *fields* • *allowExtraFields* • *extraFieldsMessage* • *allowMissingFields* • *missingFieldsMessage*
Class	`Symfony\Component\Validator\Constraints\Collection`
Validator	`Symfony\Component\Validator\Constraints\Collection`

Basic Usage

The `Collection` constraint allows you to validate the different keys of a collection individually. Take the following example:

```
namespace Acme\BlogBundle\Entity;

class Author
{
    protected $profileData = array(
        'personal_email',
        'short_bio',
    );

    public function setProfileData($key, $value)
    {
        $this->profileData[$key] = $value;
    }
}
```

To validate that the `personal_email` element of the `profileData` array property is a valid email address and that the `short_bio` element is not blank but is no longer than 100 characters in length, you would do the following:

- *YAML*

```
    properties:
        profileData:
            - Collection:
                fields:
                    personal_email: Email
                    short_bio:
                        - NotBlank
                        - MaxLength:
                            limit:   100
                            message: Your short bio is too long!
                allowMissingfields: true
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        /**
         * @Assert\Collection(
         *     fields = {
```

```
         *          "personal_email" = @Assert\Email,
         *          "short_bio" = {
         *              @Assert\NotBlank(),
         *              @Assert\MaxLength(
         *                  limit = 100,
         *                  message = "Your bio is too long!"
         *              )
         *          }
         *      },
         *      allowMissingfields = true
         * )
         */
         protected $profileData = array(
             'personal_email',
             'short_bio',
         );
    }
```

- *XML*

```
    <!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
    <class name="Acme\BlogBundle\Entity\Author">
        <property name="profileData">
            <constraint name="Collection">
                <option name="fields">
                    <value key="personal_email">
                        <constraint name="Email" />
                    </value>
                    <value key="short_bio">
                        <constraint name="NotBlank" />
                        <constraint name="MaxLength">
                            <option name="limit">100</option>
                            <option name="message">Your bio is too long!</option>
                        </constraint>
                    </value>
                </option>
                <option name="allowMissingFields">true</option>
            </constraint>
        </property>
    </class>
```

- *PHP*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\Collection;
    use Symfony\Component\Validator\Constraints\Email;
    use Symfony\Component\Validator\Constraints\MaxLength;

    class Author
    {
        private $options = array();

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('profileData', new Collection(array(
                'fields' => array(
                    'personal_email' => new Email(),
                    'lastName' => array(new NotBlank(), new MaxLength(100)),
```

```
                ),
                'allowMissingFields' => true,
            )));
        }
    }
```

Presence and Absence of Fields By default, this constraint validates more than simply whether or not the individual fields in the collection pass their assigned constraints. In fact, if any keys of a collection are missing or if there are any unrecognized keys in the collection, validation errors will be thrown.

If you would like to allow for keys to be absent from the collection or if you would like "extra" keys to be allowed in the collection, you can modify the *allowMissingFields* and *allowExtraFields* options respectively. In the above example, the `allowMissingFields` option was set to true, meaning that if either of the `personal_email` or `short_bio` elements were missing from the `$personalData` property, no validation error would occur.

Options

fields **type**: `array` [*default option*]

This option is required, and is an associative array defining all of the keys in the collection and, for each key, exactly which validator(s) should be executed against that element of the collection.

allowExtraFields **type**: `Boolean` **default**: false

If this option is set to `false` and the underlying collection contains one or more elements that are not included in the *fields* option, a validation error will be returned. If set to `true`, extra fields are ok.

extraFieldsMessage **type**: `Boolean` **default**: `The fields {{ fields }} were not expected`

The message shown if *allowExtraFields* is false and an extra field is detected.

allowMissingFields **type**: `Boolean` **default**: false

If this option is set to `false` and one or more fields from the *fields* option are not present in the underlying collection, a validation error will be returned. If set to `true`, it's ok if some fields in the *fields_* option are not present in the underlying collection.

missingFieldsMessage **type**: `Boolean` **default**: `The fields {{ fields }} are missing`

The message shown if *allowMissingFields* is false and one or more fields are missing from the underlying collection.

UniqueEntity

Validates that a particular field (or fields) in a Doctrine entity are unique. This is commonly used, for example, to prevent a new user to register using an email address that already exists in the system.

Applies to	*class*
Options	• *fields* • *message* • *em*
Class	`Symfony\Bridge\Doctrine\Validator\Constraints\Uniq`
Validator	`Symfony\Bridge\Doctrine\Validator\Constraints\Uniq`

Basic Usage

Suppose you have an `AcmeUserBundle` with a `User` entity that has an `email` field. You can use the `Unique` constraint to guarantee that the `email` field remains unique between all of the constrains in your user table:

- *Annotations*

```
// Acme/UserBundle/Entity/User.php
use Symfony\Component\Validator\Constraints as Assert;
use Doctrine\ORM\Mapping as ORM;

// DON'T forget this use statement!!!
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Entity
 * @UniqueEntity("email")
 */
class Author
{
    /**
     * @var string $email
     *
     * @ORM\Column(name="email", type="string", length=255, unique=true)
     * @Assert\Email()
     */
    protected $email;

    // ...
}
```

- *YAML*

```
# src/Acme/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Entity\Author:
    constraints:
        - Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity: email
    properties:
        email:
            - Email: ~
```

- *XML*

```
<class name="Acme\UserBundle\Entity\Author">
    <constraint name="Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity">
        <option name="fields">email</option>
        <option name="message">This email already exists.</option>
    </constraint>
     <property name="email">
```

```
            <constraint name="Email" />
        </property>
    </class>
```

Options

fields **type**: `array`` | ``string` [*default option*]

This required option is the field (or list of fields) on which this entity should be unique. For example, you could specify that both the email and name fields in the User example above should be unique.

message **type**: `string` **default**: `This value is already used.`

The message that's displayed with this constraint fails.

em **type**: `string`

The name of the entity manager to use for making the query to determine the uniqueness. If left blank, the correct entity manager will determined for this class. For that reason, this option should probably not need to be used.

Language

Validates that a value is a valid language code.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\Language`
Validator	`Symfony\Component\Validator\Constraints\LanguageVa`

Basic Usage

- *YAML*

```
    # src/UserBundle/Resources/config/validation.yml
    Acme\UserBundle\Entity\User:
        properties:
            preferredLanguage:
                - Language:
```

- *Annotations*

```
    // src/Acme/UserBundle/Entity/User.php
    namespace Acme\UserBundle\Entity;

    use Symfony\Component\Validator\Constraints as Assert;

    class User
    {
        /**
         * @Assert\Language
         */
```

```
        protected $preferredLanguage;
    }
```

Options

message **type**: `string` **default**: `This value is not a valid language`

This message is shown if the string is not a valid language code.

Locale

Validates that a value is a valid locale.

The "value" for each locale is either the two letter ISO639-1 *language* code (e.g. `fr`), or the language code followed by an underscore (_), then the ISO3166 *country* code (e.g. `fr_FR` for French/France).

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\Locale`
Validator	`Symfony\Component\Validator\Constraints\LocaleVali`

Basic Usage

- *YAML*

```
    # src/UserBundle/Resources/config/validation.yml
    Acme\UserBundle\Entity\User:
        properties:
            locale:
                - Locale:
```

- *Annotations*

```
    // src/Acme/UserBundle/Entity/User.php
    namespace Acme\UserBundle\Entity;

    use Symfony\Component\Validator\Constraints as Assert;

    class User
    {
        /**
         * @Assert\Locale
         */
         protected $locale;
    }
```

Options

message **type**: `string` **default**: `This value is not a valid locale`

This message is shown if the string is not a valid locale.

Country

Validates that a value is a valid two-letter country code.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\Country`
Validator	`Symfony\Component\Validator\Constraints\CountryVal`

Basic Usage

- *YAML*

```
# src/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Entity\User:
    properties:
        country:
            - Country:
```

- *Annotations*

```
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\Country
     */
     protected $country;
}
```

Options

message **type**: `string` **default**: `This value is not a valid country`

This message is shown if the string is not a valid country code.

File

Validates that a value is a valid "file", which can be one of the following:

- A string (or object with a `__toString()` method) path to an existing file;
- A valid `Symfony\Component\HttpFoundation\File\File` object (including objects of class `Symfony\Component\HttpFoundation\File\UploadedFile`).

This constraint is commonly used in forms with the file form type.

Tip: If the file you're validating is an image, try the Image constraint.

Applies to	*property or method*
Options	• *maxSize* • *mimeTypes* • *maxSizeMessage* • *mimeTypesMessage* • *notFoundMessage* • *notReadableMessage* • *uploadIniSizeErrorMessage* • *uploadFormSizeErrorMessage* • *uploadErrorMessage*
Class	`Symfony\Component\Validator\Constraints\File`
Validator	`Symfony\Component\Validator\Constraints\FileValida`

Basic Usage

This constraint is most commonly used on a property that will be rendered in a form as a file form type. For example, suppose you're creating an author form where you can upload a "bio" PDF for the author. In your form, the `bioFile` property would be a `file` type. The `Author` class might look as follows:

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\HttpFoundation\File\File;

class Author
{
    protected $bioFile;

    public function setBioFile(File $file = null)
    {
        $this->bioFile = $file;
    }

    public function getBioFile()
    {
        return $this->bioFile;
    }
}
```

To guarantee that the `bioFile` `File` object is valid, and that it is below a certain file size and a valid PDF, add the following:

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Author
        properties:
            bioFile:
                - File:
                    maxSize: 1024k
                    mimeTypes: [application/pdf, application/x-pdf]
                    mimeTypesMessage: Please upload a valid PDF
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\File(
     *     maxSize = "1024k",
     *     mimeTypes = {"application/pdf", "application/x-pdf"},
     *     mimeTypesMessage = "Please upload a valid PDF"
     * )
     */
    protected $bioFile;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <property name="bioFile">
        <constraint name="File">
            <option name="maxSize">1024k</option>
            <option name="mimeTypes">
                <value>application/pdf</value>
                <value>application/x-pdf</value>
            </option>
            <option name="mimeTypesMessage">Please upload a valid PDF</option>
        </constraint>
    </property>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php
// ...

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\File;

class Author
{
    // ...

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('bioFile', new File(array(
            'maxSize' => '1024k',
            'mimeTypes' => array(
                'application/pdf',
                'application/x-pdf',
            ),
            'mimeTypesMessage' => 'Please upload a valid PDF',
        )));
    }
}
```

The `bioFile` property is validated to guarantee that it is a real file. Its size and mime type are also validated because the appropriate options have been specified.

Options

maxSize **type**: `mixed`

If set, the size of the underlying file must be below this file size in order to be valid. The size of the file can be given in one of the following formats:

- **bytes**: To specify the `maxSize` in bytes, pass a value that is entirely numeric (e.g. `4096`);
- **kilobytes**: To specify the `maxSize` in kilobytes, pass a number and suffix it with a lowercase "k" (e.g. `200k`);
- **megabytes**: To specify the `maxSize` in megabytes, pass a number and suffix it with a capital "M" (e.g. `4M`).

mimeTypes **type**: `array` or `string`

If set, the validator will check that the mime type of the underlying file is equal to the given mime type (if a string) or exists in the collection of given mime types (if an array).

maxSizeMessage **type**: `string` **default**: `The file is too large ({{ size }}). Allowed maximum size is {{ limit }}`

The message displayed if the file is larger than the *maxSize* option.

mimeTypesMessage **type**: `string` **default**: `The mime type of the file is invalid ({{ type }}). Allowed mime types are {{ types }}`

The message displayed if the mime type of the file is not a valid mime type per the *mimeTypes* option.

notFoundMessage **type**: `string` **default**: `The file could not be found`

The message displayed if no file can be found at the given path. This error is only likely if the underlying value is a string path, as a `File` object cannot be constructed with an invalid file path.

notReadableMessage **type**: `string` **default**: `The file is not readable`

The message displayed if the file exists, but the PHP `is_readable` function fails when passed the path to the file.

uploadIniSizeErrorMessage **type**: `string` **default**: `The file is too large. Allowed maximum size is {{ limit }}`

The message that is displayed if the uploaded file is larger than the `upload_max_filesize` PHP.ini setting.

uploadFormSizeErrorMessage **type**: `string` **default**: `The file is too large`

The message that is displayed if the uploaded file is larger than allowed by the HTML file input field.

uploadErrorMessage **type**: `string` **default**: `The file could not be uploaded`

The message that is displayed if the uploaded file could not be uploaded for some unknown reason, such as the file upload failed or it couldn't be written to disk.

Image

The Image constraint works exactly like the File constraint, except that its *mimeTypes* and *mimeTypesMessage* options are automatically setup to work for image files specifically.

Additionally, as of Symfony 2.1, it has options so you can validate against the width and height of the image.

See the File constraint for the bulk of the documentation on this constraint.

Applies to	*property or method*
Options	• *mimeTypes* • *minWidth* • *maxWidth* • *maxHeight* • *minHeight* • *mimeTypesMessage* • *sizeNotDetectedMessage* • *maxWidthMessage* • *minWidthMessage* • *maxHeightMessage* • *minHeightMessage* • See File for inherited options
Class	`Symfony\Component\Validator\Constraints\File`
Validator	`Symfony\Component\Validator\Constraints\FileValida`

Basic Usage

This constraint is most commonly used on a property that will be rendered in a form as a file form type. For example, suppose you're creating an author form where you can upload a "headshot" image for the author. In your form, the `headshot` property would be a `file` type. The `Author` class might look as follows:

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\HttpFoundation\File\File;

class Author
{
    protected $headshot;

    public function setHeadshot(File $file = null)
    {
        $this->headshot = $file;
    }

    public function getHeadshot()
    {
        return $this->headshot;
    }
}
```

To guarantee that the `headshot File` object is a valid image and that it is between a certain size, add the following:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author
    properties:
        headshot:
            - Image:
                minWidth: 200
                maxWidth: 400
                minHeight: 200
                maxHeight: 400
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\File(
     *     minWidth = 200,
     *     maxWidth = 400,
     *     minHeight = 200,
     *     maxHeight = 400,
     * )
     */
    protected $headshot;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <property name="headshot">
        <constraint name="File">
            <option name="minWidth">200</option>
            <option name="maxWidth">400</option>
            <option name="minHeight">200</option>
            <option name="maxHeight">400</option>
        </constraint>
    </property>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php
// ...

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\File;

class Author
{
    // ...

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('headshot', new File(array(
            'minWidth' => 200,
            'maxWidth' => 400,
```

```
                'minHeight' => 200,
                'maxHeight' => 400,
            )));
        }
    }
```

The `headshot` property is validated to guarantee that it is a real image and that it is between a certain width and height.

Options

This constraint shares all of its options with the File constraint. It does, however, modify two of the default option values and add several other options.

mimeTypes **type**: `array` or `string` **default**: `image/*`

mimeTypesMessage **type**: `string` **default**: `This file is not a valid image`

New in version 2.1: All of the min/max width/height options are new to Symfony 2.1.

minWidth **type**: `integer`

If set, the width of the image file must be greater than or equal to this value in pixels.

maxWidth **type**: `integer`

If set, the width of the image file must be less than or equal to this value in pixels.

minHeight **type**: `integer`

If set, the height of the image file must be greater than or equal to this value in pixels.

maxHeight **type**: `integer`

If set, the height of the image file must be less than or equal to this value in pixels.

sizeNotDetectedMessage **type**: `string` **default**: `The size of the image could not be detected`

If the system is unable to determine the size of the image, this error will be displayed. This will only occur when at least one of the four size constraint options has been set.

maxWidthMessage **type**: `string` **default**: `The image width is too big ({{ width }}px). Allowed maximum width is {{ max_width }}px`

The error message if the width of the image exceeds *maxWidth*.

minWidthMessage **type**: `string` **default**: `The image width is too small ({{ width }}px). Minimum width expected is {{ min_width }}px`

The error message if the width of the image is less than *minWidth*.

maxHeightMessage **type**: `string` **default**: `The image height is too big ({{ height }}px). Allowed maximum height is {{ max_height }}px`

The error message if the height of the image exceeds *maxHeight*.

minHeightMessage **type**: `string` **default**: `The image height is too small ({{ height }}px). Minimum height expected is {{ min_height }}px`

The error message if the height of the image is less than *minHeight*.

Callback

The purpose of the Callback assertion is to let you create completely custom validation rules and to assign any validation errors to specific fields on your object. If you're using validation with forms, this means that you can make these custom errors display next to a specific field, instead of simply at the top of your form.

This process works by specifying one or more *callback* methods, each of which will be called during the validation process. Each of those methods can do anything, including creating and assigning validation errors.

Note: A callback method itself doesn't *fail* or return any value. Instead, as you'll see in the example, a callback method has the ability to directly add validator "violations".

Applies to	*class*
Options	• *methods*
Class	`Symfony\Component\Validator\Constraints\Callback`
Validator	`Symfony\Component\Validator\Constraints\CallbackVa`

Setup

- *YAML*

```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    constraints:
        - Callback:
            methods:    [isAuthorValid]
```

- *XML*

```xml
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <constraint name="Callback">
        <option name="methods">
            <value>isAuthorValid</value>
        </option>
    </constraint>
</class>
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

/**
```

```
     * @Assert\Callback(methods={"isAuthorValid"})
     */
    class Author
    {
    }
```

The Callback Method

The callback method is passed a special `ExecutionContext` object. You can set "violations" directly on this object and determine to which field those errors should be attributed:

```
// ...
use Symfony\Component\Validator\ExecutionContext;

class Author
{
    // ...
    private $firstName;

    public function isAuthorValid(ExecutionContext $context)
    {
        // somehow you have an array of "fake names"
        $fakeNames = array();

        // check if the name is actually a fake name
        if (in_array($this->getFirstName(), $fakeNames)) {
            $propertyPath = $context->getPropertyPath() . '.firstName';
            $context->setPropertyPath($propertyPath);
            $context->addViolation('This name sounds totally fake!', array(), null);
        }
    }
```

Options

methods **type**: `array` **default**: `array()` [*default option*]

This is an array of the methods that should be executed during the validation process. Each method can be one of the following formats:

1. **String method name**

   If the name of a method is a simple string (e.g. `isAuthorValid`), that method will be called on the same object that's being validated and the `ExecutionContext` will be the only argument (see the above example).

2. **Static array callback**

   Each method can also be specified as a standard array callback:

   - *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    constraints:
        - Callback:
            methods:
                -    [Acme\BlogBundle\MyStaticValidatorClass, isAuthorValid]
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @Assert\Callback(methods={
 *     { "Acme\BlogBundle\MyStaticValidatorClass", "isAuthorValid"}
 * })
 */
class Author
{
}
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Callback;

class Author
{
    public $name;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addConstraint(new Callback(array(
            'methods' => array('isAuthorValid'),
        )));
    }
}
```

In this case, the static method `isAuthorValid` will be called on the `Acme\BlogBundle\MyStaticValidatorClass` class. It's passed both the original object being validated (e.g. `Author`) as well as the `ExecutionContext`:

```
namespace Acme\BlogBundle;

use Symfony\Component\Validator\ExecutionContext;
use Acme\BlogBundle\Entity\Author;

class MyStaticValidatorClass
{
    static public function isAuthorValid(Author $author, ExecutionContext $context)
    {
        // ...
    }
}
```

Tip: If you specify your `Callback` constraint via PHP, then you also have the option to make your callback either a PHP closure or a non-static callback. It is *not* currently possible, however, to specify a service as a constraint. To validate using a service, you should create a custom validation constraint and add that new constraint to your class.

All

When applied to an array (or Traversable object), this constraint allows you to apply a collection of constraints to each element of the array.

Applies to	*property or method*
Options	• *constraints*
Class	`Symfony\Component\Validator\Constraints\All`
Validator	`Symfony\Component\Validator\Constraints\AllValidat`

Basic Usage

Suppose that you have an array of strings, and you want to validate each entry in that array:

- *YAML*

```
# src/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Entity\User:
    properties:
        favoriteColors:
            - All:
                - NotBlank:  ~
                - MinLength: 5
```

- *Annotations*

```
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class User
{
    /**
     * @Assert\All({
     *     @Assert\NotBlank
     *     @Assert\MinLength(5),
     * })
     */
     protected $favoriteColors = array();
}
```

Now, each entry in the `favoriteColors` array will be validated to not be blank and to be at least 5 characters long.

Options

constraints **type**: `array` [*default option*]

This required option is the array of validation constraints that you want to apply to each element of the underlying array.

UserPassword

New in version 2.1: This constraint is new in version 2.1.

This validates that an input value is equal to the current authenticated user's password. This is useful in a form where a user can change his password, but needs to enter his old password for security.

Note: This should **not** be used validate a login form, since this is done automatically by the security system.

When applied to an array (or Traversable object), this constraint allows you to apply a collection of constraints to each element of the array.

Applies to	*property or method*
Options	• *message*
Class	`Symfony\Component\Validator\Constraints\UserPasswo`
Validator	`Symfony\Bundle\SecurityBundle\Validator\Constraint`

Basic Usage

Suppose you have a *PasswordChange* class, that's used in a form where the user can change his password by entering his old password and a new password. This constraint will validate that the old password matches the user's current password:

- *YAML*

```
# src/UserBundle/Resources/config/validation.yml
Acme\UserBundle\Form\Model\ChangePassword:
    properties:
        oldPassword:
            - UserPassword:
                message: "Wrong value for your current password"
```

- *Annotations*

```
// src/Acme/UserBundle/Form/Model/ChangePassword.php
namespace Acme\UserBundle\Form\Model;

use Symfony\Component\Validator\Constraints as Assert;

class ChangePassword
{
    /**
     * @Assert\UserPassword(
     *     message = "Wrong value for your current password"
     * )
     */
     protected $oldPassword;
}
```

Options

message **type**: `message` **default**: `This value should be the user current password`

This is the message that's displayed when the underlying string does *not* match the current user's password.

Valid

This constraint is used to enable validation on objects that are embedded as properties on an object being validated. This allows you to validate an object and all sub-objects associated with it.

Applies to	*property or method*
Options	• *traverse*
Class	`Symfony\Component\Validator\Constraints\Type`

Basic Usage

In the following example, we create two classes `Author` and `Address` that both have constraints on their properties. Furthermore, `Author` stores an `Address` instance in the `$address` property.

```
// src/Acme/HelloBundle/Address.php
class Address
{
    protected $street;
    protected $zipCode;
}
```

```
// src/Acme/HelloBundle/Author.php
class Author
{
    protected $firstName;
    protected $lastName;
    protected $address;
}
```

- *YAML*

```
    # src/Acme/HelloBundle/Resources/config/validation.yml
    Acme\HelloBundle\Address:
        properties:
            street:
                - NotBlank: ~
            zipCode:
                - NotBlank: ~
                - MaxLength: 5

    Acme\HelloBundle\Author:
        properties:
            firstName:
                - NotBlank: ~
                - MinLength: 4
            lastName:
                - NotBlank: ~
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/validation.xml -->
    <class name="Acme\HelloBundle\Address">
        <property name="street">
            <constraint name="NotBlank" />
        </property>
        <property name="zipCode">
```

```
            <constraint name="NotBlank" />
            <constraint name="MaxLength">5</constraint>
        </property>
    </class>

    <class name="Acme\HelloBundle\Author">
        <property name="firstName">
            <constraint name="NotBlank" />
            <constraint name="MinLength">4</constraint>
        </property>
        <property name="lastName">
            <constraint name="NotBlank" />
        </property>
    </class>
```

- *Annotations*

```
    // src/Acme/HelloBundle/Address.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Address
    {
        /**
         * @Assert\NotBlank()
         */
        protected $street;

        /**
         * @Assert\NotBlank
         * @Assert\MaxLength(5)
         */
        protected $zipCode;
    }

    // src/Acme/HelloBundle/Author.php
    class Author
    {
        /**
         * @Assert\NotBlank
         * @Assert\MinLength(4)
         */
        protected $firstName;

        /**
         * @Assert\NotBlank
         */
        protected $lastName;

        protected $address;
    }
```

- *PHP*

```
    // src/Acme/HelloBundle/Address.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\NotBlank;
    use Symfony\Component\Validator\Constraints\MaxLength;

    class Address
```

```
    {
        protected $street;

        protected $zipCode;

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('street', new NotBlank());
            $metadata->addPropertyConstraint('zipCode', new NotBlank());
            $metadata->addPropertyConstraint('zipCode', new MaxLength(5));
        }
    }

    // src/Acme/HelloBundle/Author.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\NotBlank;
    use Symfony\Component\Validator\Constraints\MinLength;

    class Author
    {
        protected $firstName;

        protected $lastName;

        protected $address;

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('firstName', new NotBlank());
            $metadata->addPropertyConstraint('firstName', new MinLength(4));
            $metadata->addPropertyConstraint('lastName', new NotBlank());
        }
    }
```

With this mapping, it is possible to successfully validate an author with an invalid address. To prevent that, add the `Valid` constraint to the `$address` property.

- *YAML*

```
    # src/Acme/HelloBundle/Resources/config/validation.yml
    Acme\HelloBundle\Author:
        properties:
            address:
                - Valid: ~
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/validation.xml -->
    <class name="Acme\HelloBundle\Author">
        <property name="address">
            <constraint name="Valid" />
        </property>
    </class>
```

- *Annotations*

```
    // src/Acme/HelloBundle/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
```

```
    {
        /* ... */

        /**
         * @Assert\Valid
         */
        protected $address;
    }
```

- *PHP*

```
    // src/Acme/HelloBundle/Author.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\Valid;

    class Author
    {
        protected $address;

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('address', new Valid());
        }
    }
```

If you validate an author with an invalid address now, you can see that the validation of the `Address` fields failed.

AcmeHelloBundleAuthor.address.zipCode: This value is too long. It should have 5 characters or less

Options

traverse **type**: `string` **default**: `true`

If this constraint is applied to a property that holds an array of objects, then each object in that array will be validated only if this option is set to `true`.

The Validator is designed to validate objects against *constraints*. In real life, a constraint could be: "The cake must not be burned". In Symfony2, constraints are similar: They are assertions that a condition is true.

Supported Constraints

The following constraints are natively available in Symfony2:

Basic Constraints

These are the basic constraints: use them to assert very basic things about the value of properties or the return value of methods on your object.

- NotBlank
- Blank
- NotNull
- Null
- True

- False
- Type

String Constraints

- Email
- MinLength
- MaxLength
- Url
- Regex
- Ip

Number Constraints

- Max
- Min

Date Constraints

- Date
- DateTime
- Time

Collection Constraints

- Choice
- Collection
- UniqueEntity
- Language
- Locale
- Country

File Constraints

- File
- Image

Other Constraints

- Callback
- All
- UserPassword
- Valid

5.1.12 The Dependency Injection Tags

Tags:

- `data_collector`
- `form.type`
- `form.type_extension`
- `form.type_guesser`
- `kernel.cache_warmer`
- `kernel.event_listener`
- `kernel.event_subscriber`
- `monolog.logger`
- `monolog.processor`
- `templating.helper`
- `routing.loader`
- `translation.loader`
- `twig.extension`
- `validator.initializer`

Enabling Custom PHP Template Helpers

To enable a custom template helper, add it as a regular service in one of your configuration, tag it with `templating.helper` and define an `alias` attribute (the helper will be accessible via this alias in the templates):

- *YAML*

```
services:
    templating.helper.your_helper_name:
        class: Fully\Qualified\Helper\Class\Name
        tags:
            - { name: templating.helper, alias: alias_name }
```

- *XML*

```
<service id="templating.helper.your_helper_name" class="Fully\Qualified\Helper\Class\Name">
    <tag name="templating.helper" alias="alias_name" />
</service>
```

- *PHP*

```
    $container
        ->register('templating.helper.your_helper_name', 'Fully\Qualified\Helper\Class\Name')
        ->addTag('templating.helper', array('alias' => 'alias_name'))
    ;
```

Enabling Custom Twig Extensions

To enable a Twig extension, add it as a regular service in one of your configuration, and tag it with `twig.extension`:

- *YAML*

```
    services:
        twig.extension.your_extension_name:
            class: Fully\Qualified\Extension\Class\Name
            tags:
                - { name: twig.extension }
```

- *XML*

```
    <service id="twig.extension.your_extension_name" class="Fully\Qualified\Extension\Class\Name">
        <tag name="twig.extension" />
    </service>
```

- *PHP*

```
    $container
        ->register('twig.extension.your_extension_name', 'Fully\Qualified\Extension\Class\Name')
        ->addTag('twig.extension')
    ;
```

For information on how to create the actual Twig Extension class, see Twig's documentation on the topic.

Before writing your own extensions, have a look at the Twig official extension repository which already includes several useful extensions. For example `Intl` and its `localizeddate` filter that formats a date according to user's locale. These official Twig extensions also have to be added as regular services:

- *YAML*

```
    services:
        twig.extension.intl:
            class: Twig_Extensions_Extension_Intl
            tags:
                - { name: twig.extension }
```

- *XML*

```
    <service id="twig.extension.intl" class="Twig_Extensions_Extension_Intl">
        <tag name="twig.extension" />
    </service>
```

- *PHP*

```
    $container
        ->register('twig.extension.intl', 'Twig_Extensions_Extension_Intl')
        ->addTag('twig.extension')
    ;
```

Enabling Custom Listeners

To enable a custom listener, add it as a regular service in one of your configuration, and tag it with `kernel.event_listener`. You must provide the name of the event your service listens to, as well as the method that will be called:

- *YAML*

```
services:
    kernel.listener.your_listener_name:
        class: Fully\Qualified\Listener\Class\Name
        tags:
            - { name: kernel.event_listener, event: xxx, method: onXxx }
```

- *XML*

```
<service id="kernel.listener.your_listener_name" class="Fully\Qualified\Listener\Class\Name">
    <tag name="kernel.event_listener" event="xxx" method="onXxx" />
</service>
```

- *PHP*

```
$container
    ->register('kernel.listener.your_listener_name', 'Fully\Qualified\Listener\Class\Name')
    ->addTag('kernel.event_listener', array('event' => 'xxx', 'method' => 'onXxx'))
;
```

Note: You can also specify priority as an attribute of the kernel.event_listener tag (much like the method or event attributes), with either a positive or negative integer. This allows you to make sure your listener will always be called before or after another listener listening for the same event.

Enabling Custom Subscribers

New in version 2.1: The ability to add kernel event subscribers is new to 2.1.

To enable a custom subscriber, add it as a regular service in one of your configuration, and tag it with `kernel.event_subscriber`:

- *YAML*

```
services:
    kernel.subscriber.your_subscriber_name:
        class: Fully\Qualified\Subscriber\Class\Name
        tags:
            - { name: kernel.event_subscriber }
```

- *XML*

```
<service id="kernel.subscriber.your_subscriber_name" class="Fully\Qualified\Subscriber\Class\Nam
    <tag name="kernel.event_subscriber" />
</service>
```

- *PHP*

```
$container
    ->register('kernel.subscriber.your_subscriber_name', 'Fully\Qualified\Subscriber\Class\Name'
    ->addTag('kernel.event_subscriber')
;
```

Note: Your service must implement the `SymfonyComponentEventDispatcherEventSubscriberInterface` interface.

Note: If your service is created by a factory, you **MUST** correctly set the `class` parameter for this tag to work correctly.

Enabling Custom Template Engines

To enable a custom template engine, add it as a regular service in one of your configuration, tag it with `templating.engine`:

- *YAML*

```
services:
    templating.engine.your_engine_name:
        class: Fully\Qualified\Engine\Class\Name
        tags:
            - { name: templating.engine }
```

- *XML*

```
<service id="templating.engine.your_engine_name" class="Fully\Qualified\Engine\Class\Name">
    <tag name="templating.engine" />
</service>
```

- *PHP*

```
$container
    ->register('templating.engine.your_engine_name', 'Fully\Qualified\Engine\Class\Name')
    ->addTag('templating.engine')
;
```

Enabling Custom Routing Loaders

To enable a custom routing loader, add it as a regular service in one of your configuration, and tag it with `routing.loader`:

- *YAML*

```
services:
    routing.loader.your_loader_name:
        class: Fully\Qualified\Loader\Class\Name
        tags:
            - { name: routing.loader }
```

- *XML*

```
<service id="routing.loader.your_loader_name" class="Fully\Qualified\Loader\Class\Name">
    <tag name="routing.loader" />
</service>
```

- *PHP*

```
$container
    ->register('routing.loader.your_loader_name', 'Fully\Qualified\Loader\Class\Name')
```

```
        ->addTag('routing.loader')
    ;
```

Using a custom logging channel with Monolog

Monolog allows you to share its handlers between several logging channels. The logger service uses the channel `app` but you can change the channel when injecting the logger in a service.

- *YAML*

```
    services:
        my_service:
            class: Fully\Qualified\Loader\Class\Name
            arguments: [@logger]
            tags:
                - { name: monolog.logger, channel: acme }
```

- *XML*

```
    <service id="my_service" class="Fully\Qualified\Loader\Class\Name">
        <argument type="service" id="logger" />
        <tag name="monolog.logger" channel="acme" />
    </service>
```

- *PHP*

```
    $definition = new Definition('Fully\Qualified\Loader\Class\Name', array(new Reference('logger'))
    $definition->addTag('monolog.logger', array('channel' => 'acme'));
    $container->register('my_service', $definition);;
```

Note: This works only when the logger service is a constructor argument, not when it is injected through a setter.

Adding a processor for Monolog

Monolog allows you to add processors in the logger or in the handlers to add extra data in the records. A processor receives the record as an argument and must return it after adding some extra data in the `extra` attribute of the record.

Let's see how you can use the built-in `IntrospectionProcessor` to add the file, the line, the class and the method where the logger was triggered.

You can add a processor globally:

- *YAML*

```
    services:
        my_service:
            class: Monolog\Processor\IntrospectionProcessor
            tags:
                - { name: monolog.processor }
```

- *XML*

```
    <service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
        <tag name="monolog.processor" />
    </service>
```

- *PHP*

```
$definition = new Definition('Monolog\Processor\IntrospectionProcessor');
$definition->addTag('monolog.processor');
$container->register('my_service', $definition);
```

Tip: If your service is not a callable (using `__invoke`) you can add the `method` attribute in the tag to use a specific method.

You can add also a processor for a specific handler by using the `handler` attribute:

- *YAML*

```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor, handler: firephp }
```

- *XML*

```
<service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
    <tag name="monolog.processor" handler="firephp" />
</service>
```

- *PHP*

```
$definition = new Definition('Monolog\Processor\IntrospectionProcessor');
$definition->addTag('monolog.processor', array('handler' => 'firephp');
$container->register('my_service', $definition);
```

You can also add a processor for a specific logging channel by using the `channel` attribute. This will register the processor only for the `security` logging channel used in the Security component:

- *YAML*

```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor, channel: security }
```

- *XML*

```
<service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
    <tag name="monolog.processor" channel="security" />
</service>
```

- *PHP*

```
$definition = new Definition('Monolog\Processor\IntrospectionProcessor');
$definition->addTag('monolog.processor', array('channel' => 'security');
$container->register('my_service', $definition);
```

Note: You cannot use both the `handler` and `channel` attributes for the same tag as handlers are shared between all channels.

5.1.13 Requirements for running Symfony2

To run Symfony2, your system needs to adhere to a list of requirements. You can easily see if your system passes all requirements by running the `web/config.php` in your Symfony distribution. Since the CLI often uses a different `php.ini` configuration file, it's also a good idea to check your requirements from the command line via:

```
php app/check.php
```

Below is the list of required and optional requirements.

Required

- PHP needs to be a minimum version of PHP 5.3.2
- JSON needs to be enabled
- ctype needs to be enabled
- Your PHP.ini needs to have the date.timezone setting

Optional

- You need to have the PHP-XML module installed
- You need to have at least version 2.6.21 of libxml
- PHP tokenizer needs to be enabled
- mbstring functions need to be enabled
- iconv needs to be enabled
- POSIX needs to be enabled (only on *nix)
- Intl needs to be installed with ICU 4+
- APC 3.0.17+ (or another opcode cache needs to be installed)
- PHP.ini recommended settings
  - `short_open_tag = Off`
  - `magic_quotes_gpc = Off`
  - `register_globals = Off`
  - `session.autostart = Off`

Doctrine

If you want to use Doctrine, you will need to have PDO installed. Additionally, you need to have the PDO driver installed for the database server you want to use.

- **Configuration Options**:

  Ever wondered what configuration options you have available to you in files such as `app/config/config.yml`? In this section, all the available configuration is broken down by the key (e.g. `framework`) that defines each possible section of your Symfony2 configuration.

  - framework
  - doctrine

  - security
  - assetic
  - swiftmailer
  - twig
  - monolog
  - web_profiler
- **Forms and Validation**
  - Form Field Type Reference
  - Validation Constraints Reference
  - Twig Template Function Reference
- **Other Areas**
  - The Dependency Injection Tags
  - Requirements for running Symfony2
- **Configuration Options**:

  Ever wondered what configuration options you have available to you in files such as `app/config/config.yml`? In this section, all the available configuration is broken down by the key (e.g. `framework`) that defines each possible section of your Symfony2 configuration.
  - framework
  - doctrine
  - security
  - assetic
  - swiftmailer
  - twig
  - monolog
  - web_profiler
- **Forms and Validation**
  - Form Field Type Reference
  - Validation Constraints Reference
  - Twig Template Function Reference
- **Other Areas**
  - The Dependency Injection Tags
  - Requirements for running Symfony2

CHAPTER 6

Bundles

The Symfony Standard Edition comes with some bundles. Learn more about them:

6.1 Symfony SE Bundles

- `SensioFrameworkExtraBundle`
- `SensioGeneratorBundle`
- `JMSSecurityExtraBundle`
- `DoctrineFixturesBundle`
- `DoctrineMigrationsBundle`
- `DoctrineMongoDBBundle`
- `SensioFrameworkExtraBundle`
- `SensioGeneratorBundle`
- `JMSSecurityExtraBundle`
- `DoctrineFixturesBundle`
- `DoctrineMigrationsBundle`
- `DoctrineMongoDBBundle`

CHAPTER 7

Contributing

Contribute to Symfony2:

7.1 Contributing

7.1.1 Contributing Code

Reporting a Bug

Whenever you find a bug in Symfony2, we kindly ask you to report it. It helps us make a better Symfony2.

> **Caution:** If you think you've found a security issue, please use the special procedure instead.

Before submitting a bug:

- Double-check the official documentation to see if you're not misusing the framework;
- Ask for assistance on the users mailing-list, the forum, or on the #symfony IRC channel if you're not sure if your issue is really a bug.

If your problem definitely looks like a bug, report it using the official bug tracker and follow some basic rules:

- Use the title field to clearly describe the issue;
- Describe the steps needed to reproduce the bug with short code examples (providing a unit test that illustrates the bug is best);
- Give as much details as possible about your environment (OS, PHP version, Symfony version, enabled extensions, ...);
- *(optional)* Attach a patch.

Submitting a Patch

Patches are the best way to provide a bug fix or to propose enhancements to Symfony2.

Check List

The purpose of the check list is to ensure that contributions may be reviewed without needless feedback loops to ensure that your contributions can be included into Symfony2 as quickly as possible.

The pull request title should be prefixed with the component name or bundle it relates to.

```
[Component] Short title description here.
```

An example title might look like this:

```
[Form] Add selectbox field type.
```

Tip: Please use the title with "[WIP]" if the submission is not yet completed or the tests are incomplete or not yet passing.

All pull requests should include the following template in the request description:

```
Bug fix: [yes|no]
Feature addition: [yes|no]
Backwards compatibility break: [yes|no]
Symfony2 tests pass: [yes|no]
Fixes the following tickets: [comma separated list of tickets fixed by the PR]
Todo: [list of todos pending]
```

An example submission could now look as follows:

```
Bug fix: no
Feature addition: yes
Backwards compatibility break: no
Symfony2 tests pass: yes
Fixes the following tickets: -
Todo: -
```

Thank you for including the filled out template in your submission!

Tip: All feature addition's should be sent to the "master" branch, while all bug fixes should be sent to the oldest still active branch. Furthermore submissions should, as a rule of thumb, not break backwards compatibility.

Tip: To automatically get your feature branch tested, you can add your fork to travis-ci.org. Just login using your github.com account and then simply flip a single switch to enable automated testing. In your pull request, instead of specifying "*Symfony2 tests pass: [yes|no]*", you can link to the travis-ci.org status icon. For more details, see the travis-ci.org Getting Started Guide.

Initial Setup

Before working on Symfony2, setup a friendly environment with the following software:

- Git;
- PHP version 5.3.2 or above;
- PHPUnit 3.6.4 or above.

Set up your user information with your real name and a working email address:

```
$ git config --global user.name "Your Name"
$ git config --global user.email you@example.com
```

Tip: If you are new to Git, we highly recommend you to read the excellent and free ProGit book.

Tip: Windows users: when installing Git, the installer will ask what to do with line endings and suggests to replace all Lf by CRLF. This is the wrong setting if you wish to contribute to Symfony! Selecting the as-is method is your best choice, as git will convert your line feeds to the ones in the repository. If you have already installed Git, you can check the value of this setting by typing:

```
$ git config core.autocrlf
```

This will return either "false", "input" or "true", "true" and "false" being the wrong values. Set it to another value by typing:

```
$ git config --global core.autocrlf input
```

Replace –global by –local if you want to set it only for the active repository

Get the Symfony2 source code:

- Create a GitHub account and sign in;
- Fork the Symfony2 repository (click on the "Fork" button);
- After the "hardcore forking action" has completed, clone your fork locally (this will create a *symfony* directory):

```
$ git clone git@github.com:USERNAME/symfony.git
```

- Add the upstream repository as `remote`:

```
$ cd symfony
$ git remote add upstream git://github.com/symfony/symfony.git
```

Now that Symfony2 is installed, check that all unit tests pass for your environment as explained in the dedicated document.

Working on a Patch

Each time you want to work on a patch for a bug or on an enhancement, you need to create a topic branch.

The branch should be based on the *master* branch if you want to add a new feature. But if you want to fix a bug, use the oldest but still maintained version of Symfony where the bug happens (like *2.0*).

Create the topic branch with the following command:

```
$ git checkout -b BRANCH_NAME master
```

Or, if you want to provide a bugfix for the 2.0 branch, you need to first track the remote *2.0* branch locally:

```
$ git checkout -t origin/2.0
```

Then you can create a new branch off the 2.0 branch to work on the bugfix:

```
$ git checkout -b BRANCH_NAME 2.0
```

Tip: Use a descriptive name for your branch (*ticket_XXX* where *XXX* is the ticket number is a good convention for bug fixes).

The above checkout commands automatically switch the code to the newly created branch (check the branch you are working on with *git branch*).

Work on the code as much as you want and commit as much as you want; but keep in mind the following:

- Follow the coding standards (use *git diff –check* to check for trailing spaces);
- Add unit tests to prove that the bug is fixed or that the new feature actually works;
- Do atomic and logically separate commits (use the power of *git rebase* to have a clean and logical history);
- Write good commit messages.

Tip: A good commit message is composed of a summary (the first line), optionally followed by a blank line and a more detailed description. The summary should start with the Component you are working on in square brackets (`[DependencyInjection]`, `[FrameworkBundle]`, ...). Use a verb (`fixed ...`, `added ...`, ...) to start the summary and don't add a period at the end.

Submitting a Patch

Before submitting your patch, update your branch (needed if it takes you a while to finish your changes):

```
$ git checkout master
$ git fetch upstream
$ git merge upstream/master
$ git checkout BRANCH_NAME
$ git rebase master
```

Tip: Replace *master* with *2.0* if you are working on a bugfix

When doing the `rebase` command, you might have to fix merge conflicts. `git status` will show you the *unmerged* files. Resolve all the conflicts, then continue the rebase:

```
$ git add ... # add resolved files
$ git rebase --continue
```

Check that all tests still pass and push your branch remotely:

```
$ git push origin BRANCH_NAME
```

You can now discuss your patch on the dev mailing-list or make a pull request (they must be done on the `symfony/symfony` repository). To ease the core team work, always include the modified components in your pull request message, like in:

```
[Yaml] foo bar
[Form] [Validator] [FrameworkBundle] foo bar
```

Tip: Take care to point your pull request towards `symfony:2.0` if you want the core team to pull a bugfix based on the 2.0 branch.

If you are going to send an email to the mailing-list, don't forget to reference you branch URL (`https://github.com/USERNAME/symfony.git BRANCH_NAME`) or the pull request URL.

Based on the feedback from the mailing-list or via the pull request on GitHub, you might need to rework your patch. Before re-submitting the patch, rebase with upstream/master or upstream/2.0, don't merge; and force the push to the origin:

```
$ git rebase -f upstream/master
$ git push -f origin BRANCH_NAME
```

Note: when doing a push -f (or –force), always specify the branch name explicitly to avoid messing other branches in the repo (–force tells git that you really want to mess with things so do it carefully).

Often, moderators will ask you to "squash" your commits. This means you will convert many commits to one commit. To do this, use the rebase command:

```
$ git rebase -i head~3
$ git push -f origin BRANCH_NAME
```

The number 3 here must equal the amount of commits in your branch. After you type this command, an editor will popup showing a list of commits:

```
pick 1a31be6 first commit
pick 7fc64b4 second commit
pick 7d33018 third commit
```

To squash all commits into the first one, remove the word "pick" before the second and the last commits, and replace it by the word "squash" or just "s". When you save, git will start rebasing, and if succesful, will ask you to edit the commit message, which by default is a listing of the commit messages of all the commits. When you finish, execute the push command.

Note: All patches you are going to submit must be released under the MIT license, unless explicitly specified in the code.

All bug fixes merged into maintenance branches are also merged into more recent branches on a regular basis. For instance, if you submit a patch for the *2.0* branch, the patch will also be applied by the core team on the *master* branch.

Reporting a Security Issue

Found a security issue in Symfony2? Don't use the mailing-list or the bug tracker. All security issues must be sent to **security [at] symfony-project.com** instead. Emails sent to this address are forwarded to the Symfony core-team private mailing-list.

For each report, we first try to confirm the vulnerability. When it is confirmed, the core-team works on a solution following these steps:

1. Send an acknowledgement to the reporter;
2. Work on a patch;
3. Write a post describing the vulnerability, the possible exploits, and how to patch/upgrade affected applications;
4. Apply the patch to all maintained versions of Symfony;
5. Publish the post on the official Symfony blog.

Note: While we are working on a patch, please do not reveal the issue publicly.

Running Symfony2 Tests

Before submitting a patch for inclusion, you need to run the Symfony2 test suite to check that you have not broken anything.

PHPUnit

To run the Symfony2 test suite, install PHPUnit 3.6.4 or later first:

```
$ pear channel-discover pear.phpunit.de
$ pear channel-discover components.ez.no
$ pear channel-discover pear.symfony-project.com
$ pear install phpunit/PHPUnit
```

Dependencies (optional)

To run the entire test suite, including tests that depend on external dependencies, Symfony2 needs to be able to autoload them. By default, they are autoloaded from *vendor/* under the main root directory (see *autoload.php.dist*).

The test suite needs the following third-party libraries:

- Doctrine
- Swiftmailer
- Twig
- Monolog

To install them all, run the *vendors* script:

```
$ php vendors.php install
```

Note: Note that the script takes some time to finish.

After installation, you can update the vendors to their latest version with the follow command:

```
$ php vendors.php update
```

Running

First, update the vendors (see above).

Then, run the test suite from the Symfony2 root directory with the following command:

```
$ phpunit
```

The output should display *OK*. If not, you need to figure out what's going on and if the tests are broken because of your modifications.

Tip: Run the test suite before applying your modifications to check that they run fine on your configuration.

Code Coverage

If you add a new feature, you also need to check the code coverage by using the *coverage-html* option:

```
$ phpunit --coverage-html=cov/
```

Check the code coverage by opening the generated *cov/index.html* page in a browser.

Tip: The code coverage only works if you have XDebug enabled and all dependencies installed.

Coding Standards

When contributing code to Symfony2, you must follow its coding standards. To make a long story short, here is the golden rule: **Imitate the existing Symfony2 code**. Most open-source Bundles and libraries used by Symfony2 also follow the same guidelines, and you should too.

Remember that the main advantage of standards is that every piece of code looks and feels familiar, it's not about this or that being more readable.

Since a picture - or some code - is worth a thousand words, here's a short example containing most features described below:

```
<?php

/*
 * This file is part of the Symfony package.
 *
 * (c) Fabien Potencier <fabien@symfony.com>
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Acme;

class Foo
{
    const SOME_CONST = 42;

    private $foo;

    /**
     * @param string $dummy Some argument description
     */
    public function __construct($dummy)
    {
        $this->foo = $this->transform($dummy);
    }

    /**
     * @param string $dummy Some argument description
     * @return string|null Transformed input
     */
    private function transform($dummy)
    {
        if (true === $dummy) {
            return;
        }
        if ('string' === $dummy) {
            $dummy = substr($dummy, 0, 5);
        }

        return $dummy;
```

```
    }
}
```

Structure

- Never use short tags (*<?*);
- Don't end class files with the usual *?>* closing tag;
- Indentation is done by steps of four spaces (tabs are never allowed);
- Use the linefeed character (*0x0A*) to end lines;
- Add a single space after each comma delimiter;
- Don't put spaces after an opening parenthesis and before a closing one;
- Add a single space around operators (==, *&&*, ...);
- Add a single space before the opening parenthesis of a control keyword (*if*, *else*, *for*, *while*, ...);
- Add a blank line before *return* statements, unless the return is alone inside a statement-group (like an *if* statement);
- Don't add trailing spaces at the end of lines;
- Use braces to indicate control structure body regardless of the number of statements it contains;
- Put braces on their own line for classes, methods, and functions declaration;
- Separate the conditional statements (*if*, *else*, ...) and the opening brace with a single space and no blank line;
- Declare visibility explicitly for class, methods, and properties (usage of *var* is prohibited);
- Use lowercase PHP native typed constants: *false*, *true*, and *null*. The same goes for *array()*;
- Use uppercase strings for constants with words separated with underscores;
- Define one class per file - this does not apply to private helper classes that are not intended to be instantiated from the outside and thus are not concerned by the PSR-0 standard;
- Declare class properties before methods;
- Declare public methods first, then protected ones and finally private ones.

Naming Conventions

- Use camelCase, not underscores, for variable, function and method names;
- Use underscores for option, argument, parameter names;
- Use namespaces for all classes;
- Suffix interfaces with *Interface*;
- Use alphanumeric characters and underscores for file names;
- Don't forget to look at the more verbose Conventions document for more subjective naming considerations.

Documentation

- Add PHPDoc blocks for all classes, methods, and functions;
- Omit the *@return* tag if the method does not return anything;
- The *@package* and *@subpackage* annotations are not used.

License

- Symfony is released under the MIT license, and the license block has to be present at the top of every PHP file, before the namespace.

Conventions

The Coding Standards document describes the coding standards for the Symfony2 projects and the internal and third-party bundles. This document describes coding standards and conventions used in the core framework to make it more consistent and predictable. You are encouraged to follow them in your own code, but you don't need to.

Method Names

When an object has a "main" many relation with related "things" (objects, parameters, ...), the method names are normalized:

- `get()`
- `set()`
- `has()`
- `all()`
- `replace()`
- `remove()`
- `clear()`
- `isEmpty()`
- `add()`
- `register()`
- `count()`
- `keys()`

The usage of these methods are only allowed when it is clear that there is a main relation:

- a `CookieJar` has many `Cookie` objects;
- a Service `Container` has many services and many parameters (as services is the main relation, we use the naming convention for this relation);
- a Console `Input` has many arguments and many options. There is no "main" relation, and so the naming convention does not apply.

For many relations where the convention does not apply, the following methods must be used instead (where `XXX` is the name of the related thing):

Main Relation	Other Relations
`get()`	`getXXX()`
`set()`	`setXXX()`
n/a	`replaceXXX()`
`has()`	`hasXXX()`
`all()`	`getXXXs()`
`replace()`	`setXXXs()`
`remove()`	`removeXXX()`
`clear()`	`clearXXX()`
`isEmpty()`	`isEmptyXXX()`
`add()`	`addXXX()`
`register()`	`registerXXX()`
`count()`	`countXXX()`
`keys()`	n/a

Note: While "setXXX" and "replaceXXX" are very similar, there is one notable difference: "setXXX" may replace, or add new elements to the relation. "replaceXXX" on the other hand is specifically forbidden to add new elements, but most throw an exception in these cases.

Symfony2 License

Symfony2 is released under the MIT license.

According to Wikipedia:

> "It is a permissive license, meaning that it permits reuse within proprietary software on the condition that the license is distributed with that software. The license is also GPL-compatible, meaning that the GPL permits combination and redistribution with software that uses the MIT License."

The License

7.1.2 Contributing Documentation

Contributing to the Documentation

Documentation is as important as code. It follows the exact same principles: DRY, tests, ease of maintenance, extensibility, optimization, and refactoring just to name a few. And of course, documentation has bugs, typos, hard to read tutorials, and more.

Contributing

Before contributing, you need to become familiar with the markup language used by the documentation.

The Symfony2 documentation is hosted on GitHub:

```
https://github.com/symfony/symfony-docs
```

If you want to submit a patch, fork the official repository on GitHub and then clone your fork:

```
$ git clone git://github.com/YOURUSERNAME/symfony-docs.git
```

Unless you're documenting a feature that's new to Symfony 2.1, you changes should be based on the 2.0 branch instead of the master branch. To do this checkout the 2.0 branch before the next step:

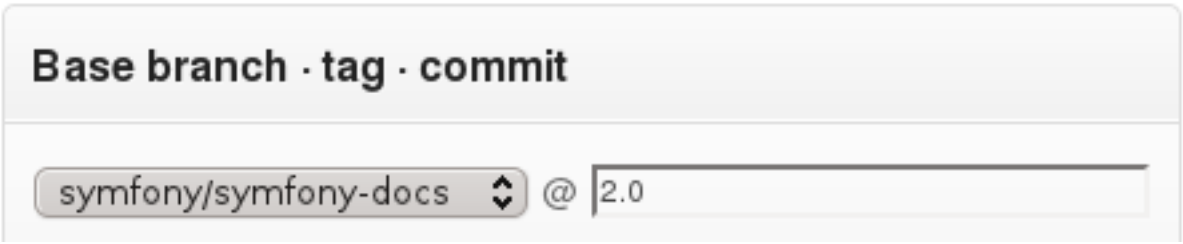
```
$ git checkout 2.0
```

Next, create a dedicated branch for your changes (for organization):

```
$ git checkout -b improving_foo_and_bar
```

You can now make your changes directly to this branch and commit them. When you're done, push this branch to *your* GitHub fork and initiate a pull request. The pull request will be between your `improving_foo_and_bar` branch and the `symfony-docs` `master` branch.

If you have made your changes based on the 2.0 branch then you need to follow the change commit link and change the base branch to be @2.0:



GitHub covers the topic of pull requests in detail.

Note: The Symfony2 documentation is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License.

Reporting an Issue

The most easy contribution you can make is reporting issues: a typo, a grammar mistake, a bug in code example, a missing explanation, and so on.

Steps:

- Submit a bug in the bug tracker;
- *(optional)* Submit a patch.

Translating

Read the dedicated document.

Documentation Format

The Symfony2 documentation uses reStructuredText as its markup language and Sphinx for building the output (HTML, PDF, ...).

reStructuredText

reStructuredText "is an easy-to-read, what-you-see-is-what-you-get plaintext markup syntax and parser system".

You can learn more about its syntax by reading existing Symfony2 documents or by reading the reStructuredText Primer on the Sphinx website.

If you are familiar with Markdown, be careful as things as sometimes very similar but different:

- Lists starts at the beginning of a line (no indentation is allowed);
- Inline code blocks use double-ticks (``like this``).

Sphinx

Sphinx is a build system that adds some nice tools to create documentation from reStructuredText documents. As such, it adds new directives and interpreted text roles to standard reST markup.

Syntax Highlighting All code examples uses PHP as the default highlighted language. You can change it with the `code-block` directive:

```
.. code-block:: yaml

    { foo: bar, bar: { foo: bar, bar: baz } }
```

If your PHP code begins with `<?php`, then you need to use `html+php` as the highlighted pseudo-language:

```
.. code-block:: html+php

    <?php echo $this->foobar(); ?>
```

Note: A list of supported languages is available on the Pygments website.

Configuration Blocks Whenever you show a configuration, you must use the `configuration-block` directive to show the configuration in all supported configuration formats (`PHP`, `YAML`, and `XML`)

```
.. configuration-block::

    .. code-block:: yaml

        # Configuration in YAML

    .. code-block:: xml

        <!-- Configuration in XML //-->

    .. code-block:: php

        // Configuration in PHP
```

The previous reST snippet renders as follow:

- *YAML*

```
# Configuration in YAML
```

- *XML*

```
<!-- Configuration in XML //-->
```

- *PHP*

```
// Configuration in PHP
```

The current list of supported formats are the following:

Markup format	Displayed
html	HTML
xml	XML
php	PHP
yaml	YAML
jinja	Twig
html+jinja	Twig
jinja+html	Twig
php+html	PHP
html+php	PHP
ini	INI
php-annotations	Annotations

Testing Documentation To test documentation before a commit:

- Install Sphinx;
- Run the Sphinx quick setup;
- Install the configuration-block Sphinx extension (see below);
- Run `make html` and view the generated HTML in the `build` directory.

Installing the configuration-block Sphinx extension

- Download the extension from the configuration-block source repository

- Copy the `configurationblock.py` to the `_exts` folder under your source folder (where `conf.py` is located)
- Add the following to the `conf.py` file:

```
# ...
sys.path.append(os.path.abspath('_exts'))

# ...
# add configurationblock to the list of extensions
extensions = ['configurationblock']
```

Translations

The Symfony2 documentation is written in English and many people are involved in the translation process.

Contributing

First, become familiar with the markup language used by the documentation.

Then, subscribe to the Symfony docs mailing-list, as collaboration happens there.

Finally, find the *master* repository for the language you want to contribute for. Here is the list of the official *master* repositories:

- *English*: http://github.com/symfony/symfony-docs
- *French*: https://github.com/gscorpio/symfony-docs-fr
- *Italian*: https://github.com/garak/symfony-docs-it
- *Japanese*: https://github.com/symfony-japan/symfony-docs-ja
- *Polish*: http://github.com/ampluso/symfony-docs-pl
- *Romanian*: http://github.com/sebio/symfony-docs-ro
- *Russian*: http://github.com/avalanche123/symfony-docs-ru
- *Spanish*: https://github.com/gitnacho/symfony-docs-es

Note: If you want to contribute translations for a new language, read the *dedicated section*.

Joining the Translation Team

If you want to help translating some documents for your language or fix some bugs, consider joining us; it's a very easy process:

- Introduce yourself on the Symfony docs mailing-list;
- *(optional)* Ask which documents you can work on;
- Fork the *master* repository for your language (click the "Fork" button on the GitHub page);
- Translate some documents;
- Ask for a pull request (click on the "Pull Request" from your page on GitHub);
- The team manager accepts your modifications and merges them into the master repository;

- The documentation website is updated every other night from the master repository.

Adding a new Language

This section gives some guidelines for starting the translation of the Symfony2 documentation for a new language.

As starting a translation is a lot of work, talk about your plan on the Symfony docs mailing-list and try to find motivated people willing to help.

When the team is ready, nominate a team manager; he will be responsible for the *master* repository.

Create the repository and copy the *English* documents.

The team can now start the translation process.

When the team is confident that the repository is in a consistent and stable state (everything is translated, or non-translated documents have been removed from the toctrees – files named `index.rst` and `map.rst.inc`), the team manager can ask that the repository is added to the list of official *master* repositories by sending an email to Fabien (fabien at symfony.com).

Maintenance

Translation does not end when everything is translated. The documentation is a moving target (new documents are added, bugs are fixed, paragraphs are reorganized, ...). The translation team need to closely follow the English repository and apply changes to the translated documents as soon as possible.

> **Caution:** Non maintained languages are removed from the official list of repositories as obsolete documentation is dangerous.

Symfony2 Documentation License

  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

- *Notice* — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

This is a human-readable summary of the Legal Code (the full license).

7.1.3 Community

IRC Meetings

The purpose of this meeting is to discuss topics in real time with many of the Symfony2 devs.

Anyone may propose topics on the symfony-dev mailing-list until 24 hours before the meeting, ideally including well prepared relevant information via some URL. 24 hours before the meeting a link to a doodle will be posted including a list of all proposed topics. Anyone can vote on the topics until the beginning of the meeting to define the order in the agenda. Each topic will be timeboxed to 15mins and the meeting lasts one hour, leaving enough time for at least 4 topics.

> **Caution:** Note that its not the expected goal of them meeting to find final solutions, but more to ensure that there is a common understanding of the issue at hand and move the discussion forward in ways which are hard to achieve with less real time communication tools.

Meetings will happen each Thursday at 17:00 CET (+01:00) on the #symfony-dev channel on the Freenode IRC server.

The IRC logs will later be published on the trac wiki, which will include a short summary for each of the topics. Tickets will be created for any tasks or issues identified during the meeting and referenced in the summary.

Some simple guidelines and pointers for participation:

- It's possible to change votes until the beginning of the meeting by clicking on "Edit an entry";
- The doodle will be closed for voting at the beginning of the meeting;
- Agenda is defined by which topics got the most votes in the doodle, or whichever was proposed first in case of a tie;
- At the beginning of the meeting one person will identify him/herself as the moderator;
- The moderator is essentially responsible for ensuring the 15min timebox and ensuring that tasks are clearly identified;
- Usually the moderator will also handle writing the summary and creating trac tickets unless someone else steps up;
- Anyone can join and is explicitly invited to participate;
- Ideally one should familiarize oneself with the proposed topic before the meeting;
- When starting on a new topic the proposer is invited to start things off with a few words;
- Anyone can then comment as they see fit;
- Depending on how many people participate one should potentially retrain oneself from pushing a specific argument too hard;
- Remember the IRC logs will be published later on, so people have the chance to review comments later on once more;
- People are encouraged to raise their hand to take on tasks defined during the meeting.

Here is an example doodle.

Other Resources

In order to follow what is happening in the community you might find helpful these additional resources:

- List of open pull requests
- List of recent commits
- List of open bugs and enhancements
- List of open source bundles
- **Code**:
  - Bugs |
  - Patches |
  - Security |
  - Tests |
  - Coding Standards |
  - Code Conventions |
  - License
- **Documentation**:
  - Overview |
  - Format |
  - Translations |
  - License
- **Community**:
  - IRC Meetings |
  - Other Resources
- **Code**:
  - Bugs |
  - Patches |
  - Security |
  - Tests |
  - Coding Standards |
  - Code Conventions |
  - License
- **Documentation**:
  - Overview |
  - Format |
  - Translations |

T

V

W