Sylius

Release

June 15, 2014

Contents

Sylius is a modern e-commerce solution for PHP, based on **'Symfony2 Framework'_**.

Note: This documentation assumes you have a working knowledge of the Symfony2 Framework. If you haven't, please start by reading the Quick Tour from the Symfony documentation.

CHAPTER 1

The Book

The Developer's guide to leveraging the flexibility of Sylius.

1.1 The Book

1.1.1 Introduction to Sylius

There are several ways you can use Sylius for your project, but there are 2 main use cases:

1. You can use our main application, customize views, configuration and start the project;
2. You can use standalone bundles to build a solution that will fit the most sophisticated needs.

Using Sylius as a Whole

Sylius project provides a **full stack e-commerce solution**. In further parts of The Book, you'll learn how to master it and develop your next project really quickly. Our main application called **Sylius** provides a complete webshop solution and some of its features include:

- Flexible product catalog, with multiple variants per product, options, properties (think attributes) and prototypes.
- Categorization engine, which allows you to categorize the products under various taxonomies, by "Brand", "Category" or whatever you can imagine.
- Inventory tracking system, where you can track every single unit of your inventory or disable tracking at all.
- Powerful shipping with configurable shipping categories, item sizes, weight, shipments management and customizable cost calculators.
- Taxation engine, with support for many different tax categories, rates and zones.
- Orders system allowing you to easily create and manage sales, with super-flexible adjustments which can serve many different purposes, from taxation & shipping to promotions and manual order total changes.
- Customizable checkout process, built from reusable steps.
- ... and more!

If that is what you were looking for, great! But there is other great news for you: All features mentioned above are available as individual bundles too.

Leveraging Sylius Bundles

Even if the main goal of the project is to provide the full stack solution mentioned above, the Sylius full stack solution is built on top of decoupled and independent bundles. Every functionality you like in the main Sylius application can be integrated into your existing project. You can also build a tailored solution from ground-up using those components. (**Bundles** in Symfony2 glossary)

If you have an existing Symfony2 application, for example: A book catalogue, you can use the *SyliusOrderBundle* and add orders management feature.

Here is a full list of all available Sylius bundles:

- *Bundles General Guide*
- *SyliusResourceBundle*
- *SyliusProductBundle*
- *SyliusOrderBundle*
- *SyliusCartBundle*
- *SyliusAddressingBundle*
- *SyliusInventoryBundle*
- *SyliusShippingBundle*
- *SyliusTaxationBundle*
- *SyliusPromotionsBundle*
- *SyliusTaxonomiesBundle*
- *SyliusOmnipayBundle*
- *SyliusSettingsBundle*
- *SyliusFlowBundle*

Final Thoughts

Now, depending on how you want to use Sylius, continue reading The Book, which covers the usage of the full stack solution or browse the Bundles Reference.

1.1.2 Installation

The Sylius main application can serve as end-user app, as well as a foundation for your custom e-commerce application.

This article assumes you're familiar with Composer, a dependency manager for PHP. It also assumes you have **'Composer installed globally'_**.

Note: If you downloaded the Composer phar archive, you should use `php composer.phar` where this guide uses `composer`.

To create a new project using Sylius, run this command:

```
$ composer create-project -s dev sylius/sylius
```

This will create a new sylius project in `sylius`. When all the dependencies are installed, you'll be asked to fill the `parameters.yml` file via interactive script. Please follow the steps. After everything is in place, run the following commands:

```
# move to the newly created sylius directory
$ cd sylius
$ php app/console sylius:install
```

Accessing the Shop

In order to see the shop, access the `web/app_dev.php` file via your web browser.

Tip: If you use PHP 5.4 or higher, you can also use the build-in webserver for Symfony. Run the `php app/console server:run` command and then access `http://localhost:8000`.

1.1.3 Architecture Overview

...

Components

...

Symfony2 Bundles

...

Sylius Application

...

Sylius Standard Edition

...

Final Thoughts

...

Learn more

- ...

1.1.4 Products

...

Variants and The Master Variant

...

Options

...

Properties

...

Prototypes

...

Images

...

Final Thoughts

...

Learn more

- ...

1.1.5 Addresses

...

Countries

...

Provinces

...

Zones

...

Final Thoughts

...

Learn more

- ...

1.1.6 Inventory

...

StockItems

...

StockLocations

...

StockMovements

...

StockTransfers

...

Final Thoughts

...

Learn more

- ...

1.1.7 Orders

...

Customer Reference

...

Billing and Shipping Address

...

Order Contents

...

Taxes and Shipping Fees as Adjustments

...

Shipments

...

Payments

...

The Order State Machine

...

Final Thoughts

...

Learn more

- ...

1.1.8 Shipments

...

The Shipment State Machine

...

Shipping Methods

...

Shipping Cost Calculators

...

Default Calculators

...

Shipping Zones

...

Examples

...

Product and ProductVariant Configuration

...

Final Thoughts

...

Learn more

- ...

1.1.9 Payments

...

Payum

...

The Payment State Machine

...

Payment Methods

...

Gateway and Configurations

...

Payment Processing

...

Supported Gateways

...

Final Thoughts

...

Learn more

- ...

1.1.10 Taxation

...

Tax Categories

...

Tax Rates

...

Tax Zones

...

Default Tax Zone

...

Examples

...

Calculators

...

Final Thoughts

...

Learn more

- ...

1.1.11 Pricing

...

Calculators

...

Product and ProductVariant Configuration

...

Final Thoughts

...

Learn more

- ...

1.1.12 Promotions

...

Promotion Rules

...

Promotion Actions

...

Coupons

...

Examples

...

Exclusive Promotions

Final Thoughts

...

Learn more

- ...

1.1.13 Users and Groups

...

User Management

...

Groups

...

Final Thoughts

...

Learn more

- ...

1.1.14 Currencies

...

Currency Management

...

Currency Context

...

Available Currency Resolver

...

Final Thoughts

...

Learn more

- ...

1.1.15 Locales

...

Locale Management

...

Available Locales Resolver

...

Final Thoughts

...

Learn more

- ...

1.1.16 Content

...

SymfonyCMF and PHPCR

...

Static Content

...

Pages

...

Blocks

...

Final Thoughts

...

Learn more

- ...

1.1.17 Settings

...

General Settings

...

Taxtion Settings

...

Final Thoughts

...

Learn more

- ...
- *Introduction to Sylius*
- *Installation*
- *Architecture Overview*
- *Products*
- *Addresses*

- *Inventory*
- *Orders*
- *Shipments*
- *Payments*
- *Taxation*
- *Pricing*
- *Promotions*
- *Users and Groups*
- *Currencies*
- *Locales*
- *Content*
- *Settings*
- *Introduction to Sylius*
- *Installation*
- *Architecture Overview*
- *Products*
- *Addresses*
- *Inventory*
- *Orders*
- *Shipments*
- *Payments*
- *Taxation*
- *Pricing*
- *Promotions*
- *Users and Groups*
- *Currencies*
- *Locales*
- *Content*
- *Settings*

CHAPTER 2

The User Guide

The User's guide around the Sylius interface and configuration.

2.1 The Guide

2.1.1 Installation

There are several ways to install Sylius main app for the standard usage.

Using Git

...

Using Composer

We assume you're familiar with Composer, a dependency manager for PHP.

If you have Composer installed globally.

```
$ composer create-project -s dev sylius/sylius-standard
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar create-project -s dev sylius/sylius-standard
```

When all the dependencies are installed, you'll be asked to fill the `parameters.yml` file via interactive script. Please follow the guide and when everything is in place, finally run the following commands.

```
$ cd sylius-standard
$ app/console sylius:install
```

- *Installation*
- *Installation*

CHAPTER 3

Cookbook

Specific solutions for specific needs.

3.1 Cookbook

3.1.1 Extending the menu

You can add entries to the menu via events easily. You get passed a **:class:'Sylius\\Bundle\\WebBundle\\Event\\MenuBuilderEvent'** with `FactoryInterface` and `ItemInterface` of KnpMenu. So you can manipulate the whole menu.

Available only for the backend at the moment.

Example Usage

```
// src/Acme/ReportsBundle/EventListener/MenuBuilderListener.php
namespace Acme\ReportsBundle\EventListener;

use Sylius\Bundle\WebBundle\Event\MenuBuilderEvent;

class MenuBuilderListener
{
    public function addBackendMenuItems(MenuBuilderEvent $event)
    {
        $menu = $event->getMenu();

        $menu['sales']->addChild('reports', array(
            'route' => 'acme_reports_index',
            'labelAttributes' => array('icon' => 'glyphicon glyphicon-stats'),
        ))->setLabel('Daily and monthly reports');
    }
}
```

- *YAML*

```
services:
    acme_reports.menu_builder:
        class: Acme\ReportsBundle\EventListener\MenuBuilderListener
        tags:
```

```
            - { name: kernel.event_listener, event: sylius.menu_builder.backend.main, method: ad
            - { name: kernel.event_listener, event: sylius.menu_builder.backend.sidebar, method:
```

- *XML*

```
<!-- src/Acme/ReportsBundle/Resources/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<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_reports.menu_builder" class="Acme\ReportsBundle\EventListener\MenuBuil
            <tag name="kernel.event_listener" event="sylius.menu_builder.backend.main" method="a
            <tag name="kernel.event_listener" event="sylius.menu_builder.backend.sidebar" method
        </service>
    </services>
</container>
```

- *PHP*

```
// src/Acme/ReportsBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

$definition = new Definition('Acme\ReportsBundle\EventListener\MenuBuilderListener');
$definition->->addTag('kernel.event_listener', array('event' => 'sylius.menu_builder.backend.mai
$definition->->addTag('kernel.event_listener', array('event' => 'sylius.menu_builder.backend.sid

$container->setDefinition('acme_reports.menu_builder', $definition);
```

- *Extending the menu*
- *Extending the menu*

CHAPTER 4

Bundles

Documentation of all Sylius bundles.

4.1 Symfony2 Ecommerce Bundles

4.1.1 Bundles General Guide

All Sylius bundles share the same architecture and this guide will introduce you these conventions.

You'll learn how to perform basic CRUD operations on the data and how to override models, controllers, repositories, validation mapping and forms.

Performing basic CRUD operations

Sylius is using the Doctrine Common persistence interfaces. This means that every model within Sylius bundles has its own repository and object manager.

Some interfaces extend the Timestampable and SoftDeletable interfaces. Those interfaces are defined in the SyliusResourceBundle to not create a dependency on Doctrine ORM. They are however compatible with the GedmoDoctrineExtensions when using Doctrine ORM.

Retrieving resources

Retrieving any resource from database always happens via the repository, which implements `Sylius\Bundle\ResourceBundle\Model\RepositoryInterface`. If you have been using Doctrine, you should already be familiar with this concept, as it extends the default Doctrine `ObjectRepository` interface.

Let's assume you want to load a product from database. Your product repository is always accessible via the `sylius.repository.product` service.

```
<?php

public function myAction()
{
    $repository = $this->container->get('sylius.repository.product');
}
```

Retrieving many resources is as simple as calling the proper methods on the repository.

```php
<?php

public function myAction()
{
    $repository = $this->container->get('sylius.repository.product');

    $product = $repository->find(4); // Get product with id 4, returns null if not found.
    $product = $repository->findOneBy(array('slug' => 'my-super-product')); // Get one product by de

    $products = $repository->findAll(); // Load all the products!
    $products = $repository->findBy(array('special' => true)); // Find products matching some custom
}
```

Every Sylius repository supports paginating products. To create a Pagerfanta instance use the `createPaginator` method.

```php
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');

    $products = $repository->createPaginator();
    $products->setMaxPerPage(3);
    $products->setCurrentPage($request->query->get('page', 1));

    // Now you can returns products to template and iterate over it to get products from current page
}
```

Paginator can be created for a specific criteria and with desired sorting.

```php
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');

    $products = $repository->createPaginator(array('foo' => true), array('createdAt' => 'desc'));
    $products->setMaxPerPage(3);
    $products->setCurrentPage($request->query->get('page', 1));
}
```

Creating a new resource object

To create a new resource instance, you can simply call the `createNew()` method on the repository.

Now let's try something else than product, we'll create a new TaxRate model.

```php
<?php

public function myAction()
{
    $repository = $this->container->get('sylius.repository.tax_rate');
    $taxRate = $repository->createNew();
}
```

Note: Creating resources via this factory method makes the code more testable, and allows you to change the model class easily.

Saving and removing resources

To save or remove a resource, you can use any `ObjectManager` which is capable of managing the class. Every model has its own manager alias, for example the `sylius.manager.address` is an alias to the ORM EntityManager.

Of course, it is also perfectly fine if you use the `doctrine.orm.entity_manager` service name or any other appropriate manager service.

```
<?php

public function myAction()
{
    $repository = $this->container->get('sylius.repository.address');
    $manager = $this->container->get('sylius.manager.address'); // Alias to the appropriate doctrine

    $address = $repository->createNew();

    $address
        ->setFirstname('John')
        ->setLastname('Doe')
    ;

    $manager->persist($address);
    $manager->flush(); // Save changes in database.
}
```

To remove a resource, you also use the manager.

```
<?php

public function myAction()
{
    $repository = $this->container->get('sylius.repository.shipping_method');
    $manager = $this->container->get('sylius.manager.shipping_method');

    $shippingMethod = $repository->findOneBy(array('name' => 'DHL Express'));

    $manager->remove($shippingMethod);
    $manager->flush(); // Save changes in database.
}
```

Overriding Models

...

Extending base Models

All Sylius models live in `Sylius\Bundle\XyzBundle\Model` namespace together with the interfaces. As an example, for **SyliusTaxationBundle** it's *TaxCategory* and *TaxRate*.

Let's assume you want to add "zone" field to the Sylius tax rates.

Firstly, you need to create your own `TaxRate` class, which will extend the base model.

```php
namespace Acme\Bundle\ShopBundle\Entity;

use Sylius\Bundle\AddressingBundle\Model\ZoneInterface;
use Sylius\Bundle\TaxationBundle\Model\TaxRate as BaseTaxRate;

class TaxRate extends BaseTaxRate
{
    private $zone;

    public function getZone()
    {
        return $this->zone;
    }

    public function setZone(ZoneInterface $zone)
    {
        $this->zone = $zone;

        return $this;
    }
}
```

Secondly, define the entity mapping inside `Resources/config/doctrine/TaxRate.orm.xml` of your bundle.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<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\ShopBundle\Entity\TaxRate" table="sylius_tax_rate">
        <many-to-one field="zone" target-entity="Sylius\Bundle\AddressingBundle\Model\ZoneInterface">
            <join-column name="zone_id" referenced-column-name="id" nullable="false" />
        </many-to-one>
    </entity>

</doctrine-mapping>
```

Finally, you configure your class in `app/config/config.yml` file.

```yaml
sylius_taxation:
    driver: doctrine/orm
    classes:
        tax_rate:
            model: Acme\ShopBundle\Entity\TaxRate # Your tax rate entity.
```

Done! Sylius will now use your **TaxRate** model!

What has happened?

- Parameter `sylius.model.tax_rate.class` contains `Acme\\Bundle\\ShopBundle\\Entity\\TaxRate`.
- `sylius.repository.tax_rate` represents Doctrine repository for your new class.
- `sylius.manager.tax_rate` represents Doctrine object manager for your new class.
- `sylius.controller.tax_rate` represents the controller for your new class.

- All Doctrine relations to `Sylius\\Bundle\\TaxationBundle\\Model\\TaxRateInterface` are using your new class as *target-entity*, you do not need to update any mappings.
- `TaxRateType` form type is using your model as `data_class`.
- `Sylius\\Bundle\\TaxationBundle\\Model\\TaxRate` is automatically turned into Doctrine Mapped Superclass.

Overriding Controllers

All Sylius bundles are using *SyliusResourceBundle* as a foundation for database storage.

Extending base Controller

If you want to modify the controller or add your custom actions, you can do so by defining a new controller class. By extending resource controller, you also get access to several handy methods. Let's add to our custom controller a new method to recommend a product:

```
<?php

// src/Acme/ShopBundle/Controller/ProductController.php

namespace Acme\ShopBundle\Controller;

use Sylius\Bundle\ResourceBundle\Controller\ResourceController;
use Symfony\Component\HttpFoundation\Request;

class ProductController extends ResourceController
{
    public function recommendAction(Request $request, $id)
    {
        $product = $this->findOr404(array('id' => $id)); // Find product with given id or return 404
        $product->incrementRecommendations(); // Add +1!

        $this->persistAndFlush($product); // Save product.

        return $this->redirect($this->generateUrl('acme_shop_homepage'));
    }
}
```

You also need to configure your controller class in `app/config/config.yml`.

```
# app/config/config.yml

sylius_product:
    driver: doctrine/orm
    classes:
        product:
            controller: Acme\ShopBundle\Controller\ProductController
```

That's it! Now `sylius.controller.product:recommendAction` is available. You can use it by defining a new route.

```
# app/config/routing.yml

acme_shop_product_recommend:
    pattern: /products/{id}/recommend
```

```
    defaults:
        _controller: sylius.controller.product:recommendAction
```

What has happened?

- Parameter `sylius.controller.product.class` contains `Acme\\Bundle\\ShopBundle\\Controller\\Prod`
- Controller service `sylius.controller.product` is using your new controller class.

Overriding Repositories

Overriding a Sylius model repository involves extending the base class and configuring it inside the bundle configuration.

Extending base Repository

Sylius is using both custom and default Doctrine repositories and often you'll need to add your own methods. Let's assume you want to find all orders for a given customer.

Firstly, you need to create your own repository class

```
<?php

// src/Acme/ShopBundle/Repository/OrderRepository.php

namespace Acme\ShopBundle\Repository;

use Sylius\Bundle\ResourceBundle\Doctrine\ORM\EntityRepository;

class OrderRepository extends EntityRepository
{
    public function findByCustomer(Customer $customer)
    {
        return $this
            ->createQueryBuilder('o')
            ->join('o.billingAddress', 'billingAddress')
            ->join('o.shippingAddress', 'shippingAddress')
            ->join('o.customer', 'customer')
            ->where('o.customer = :customer')
            ->setParameter('customer', $customer)
            ->getQuery()
            ->getResult()
        ;
    }
}
```

Secondly, need to configure your repository class in `app/config/config.yml`.

```
# app/config/config.yml

sylius_order:
    driver: doctrine/orm
    classes:
        order:
            repository: Acme\ShopBundle\Repository\OrderRepository
```

That's it! Now `sylius.repository.order` is using your new class.

```php
<?php

public function ordersAction()
{
    $customer = // Obtain customer instance.
    $repository = $this->container->get('sylius.repository.order');

    $orders = $repository->findByCustomer($customer);
}
```

What has happened?

- Parameter `sylius.repository.order.class` contains `Acme\\ShopBundle\\Repository\\OrderRepositor`
- Repository service `sylius.repository.order` is using your new class.

Overriding Validation

All Sylius validation mappings and forms are using `sylius` as the default group.

Changing the validation group

You can configure your own validation for Sylius models. If the defaults do not fit your needs, create `validation.xml` inside your bundle.

```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/schema/dic/services/constraint-mapping-1.

    <class name="Sylius\Bundle\TaxationBundle\Model\TaxCategory">
        <property name="name">
            <constraint name="NotBlank">
                <option name="message">Fill me in!</option>
                <option name="groups">acme</option>
            </constraint>
            <constraint name="Length">
                <option name="min">5</option>
                <option name="max">255</option>
                <option name="minMessage">Looonger!</option>
                <option name="maxMessage">Shooorter!</option>
                <option name="groups">acme</option>
            </constraint>
        </property>
    </class>

</constraint-mapping>
```

You also need to configure the new validation group in `app/config/config.yml`.

```yaml
sylius_taxation:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.
    validation_groups:
        tax_category: [acme]
```

Done! Now all Sylius forms will use `acme` validation group on all forms of tax category.

Overriding Forms

Every form type in Sylius holds its class name in a specific parameter. This allows you to easily add or remove fields by extending the base form class.

Extending base Forms

All Sylius form types live in `Sylius\Bundle\XyzBundle\Form\Type` namespace.

Let's assume you want to add "phone" and remove "company" fields to the Sylius address form.

You have to create your own `AddressType` class, which will extend the base form type.

```
namespace Acme\Bundle\ShopBundle\Form\Type;

use Sylius\Bundle\AddressingBundle\Form\Type\AddressType as BaseAddressType;
use Symfony\Component\Form\FormBuilderInterface;

class AddressType extends BaseAddressType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        parent::buildForm($builder, $options); // Add default fields.

        $builder->remove('company');
        $builder->add('phone', 'text', array('required' => false));
    }
}
```

Now, define the new form class in the `app/config/config.yml`.

```
# app/config/config.yml

sylius_addressing:
    driver: doctrine/orm
    classes:
        address:
            form: Acme\ShopBundle\Form\Type\AddressType
```

Done! Sylius will now use your custom address form everywhere!

What has happened?

- Parameter `sylius.form.type.address.class` contains `Acme\\Bundle\\ShopBundle\\Form\\Type\\Addre`
- `sylius.form.type.address` form type service uses your custom class.
- `sylius_address` form type uses your new form everywhere.

Mapping Relations

All Sylius bundles use the Doctrine RTEL functionality, which allows to map the relations using interfaces, not implementations.

Using interfaces

When defining relation to Sylius model, use the interface name instead of concrete class. It will be automatically replaced with the configured model, enabling much greater flexibility.

```
<?xml version="1.0" encoding="UTF-8"?>

<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\ShopBundle\Entity\Product" table="sylius_product">
        <many-to-one field="taxCategory" target-entity="Sylius\Bundle\TaxationBundle\Model\TaxCategor
           <join-column name="tax_category_id" referenced-column-name="id" nullable="true" />
        </many-to-one>
    </entity>

</doctrine-mapping>
```

Thanks to this approach, if the **TaxCategory** model has changed, you do not need to worry about remapping other entities.

Events

All Sylius bundles are using *SyliusResourceBundle*, which has some built-in events.

Events reference

Event	Description
sylius.<resource>.pre_create	Before persist
sylius.<resource>.create	After persist
sylius.<resource>.post_create	After flush
sylius.<resource>.pre_update	Before persist
sylius.<resource>.update	After persist
sylius.<resource>.post_update	After flush
sylius.<resource>.pre_delete	Before remove
sylius.<resource>.delete	After remove
sylius.<resource>.post_delete	After flush

CRUD events example

Let's take the **Product** model as an example. As you should already know, the product controller is represented by the `sylius.controller.product` service. Several useful events are dispatched during execution of every default action of this controller. When creating a new resource via the `createAction` method, 3 events occur.

First, before the `persist()` is called on the product, the `sylius.product.pre_create` event is dispatched.

Secondly, just before the database flush is performed, Sylius dispatches the `sylius.product.create` event.

Finally, after the data storage is updated, `sylius.product.post_create` is triggered.

The same set of events is available for the `update` and `delete` operations. All the dispatches are using the `GenericEvent` class and return the **Product** object by the `getSubject` method.

Registering a listener

We'll stay with the **Product** model and create a listener which updates Solr (search engine) document every time a product is updated.

```
namespace Acme\ShopBundle\EventListener;

use Symfony\Component\EventDispatcher\GenericEvent;

class SolrListener
{
    // ... Constructor with indexer injection code.

    public function onProductUpdate(GenericEvent $event)
    {
        $this->indexer->updateProductDocument($event->getSubject());
    }
}
```

Now you need to register the listener in services configuration.

```
<?xml version="1.0" encoding="UTF-8"?>

<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/services-1.0.xsd">

    <services>
        <service id="acme.listener.solr" class="Acme\ShopBundle\EventListener\SolrListener">
            <tag name="kernel.event_listener" event="sylius.product.post_update" method="onProductUpd
        </service>
    </services>

</container>
```

Done! Every time a product is edited and the database updated, this listener will also use the indexer to update Solr index accordingly.

4.1.2 SyliusResourceBundle

Easy CRUD and persistence for Symfony2 apps.

During our work on Sylius, we noticed a lot of duplicated code across all controllers. We started looking for a good solution to the problem. We're not big fans of administration generators (they're cool, but not for our use case!) - we wanted something simpler and more flexible.

Another idea was to not limit ourselves to one persistence backend. Initial implementation included custom manager classes, which was quite of an overhead, so we decided to simply stick with Doctrine Common Persistence interfaces. If you are using Doctrine ORM or any of the ODM's, you're already familiar with those concepts. Resource bundle relies mainly on the *ObjectManager* and *ObjectRepository* interfaces.

The last annoying problem this bundle tries to solve, is having separate "backend" and "frontend" controllers, or any other duplication for displaying the same resource, with different presentation (view). We also wanted an easy way to filter some resources from list, sort them or display them by id, slug or any other criteria - without having to define another super simple action for that purpose.

If these are issues you're struggling with, this bundle may be helpful!

Please note that this bundle **is not an admin generator**. It won't create forms, filters and grids for you. It only provides format agnostic controllers as foundation to build on, with some basic sorting and filter mechanisms.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your `composer.json` and download the package.

If you have Composer installed globally.

```
$ composer require "sylius/resource-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/resource-bundle"
```

Adding required bundles to the kernel

You just need to enable proper bundles inside the kernel.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),
    );
}
```

To benefit from bundle services, you have to first register your bundle class as a *resource*.

You also need to enable HTTP method parameter override, by calling the following method on the **Request** object.

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

Request::enableHttpMethodParameterOverride();
```

Configuration

There are two ways to configure the resources used by this bundle. You can manage your configuration for all yours bundles (explained in Basic Configuration) or into yours bundles (explained in Advanced configuration).

Basic configuration

In your *app/config.yml* (or in an imported configuration file), you need to define what resources you want to use :

```
sylius_resource:
    resources:
        app.user:
            driver: doctrine/orm
            templates: App:User
            classes:
                model: App\Entity\User
                interface: App\Entity\UserInterface
                controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
                repository: Sylius\Bundle\ResourceBundle\Doctrine\ORM\EntityRepository
```

Advanced configuration

You need to expose a semantic configuration for your bundle. The following example show you a basic *Configuration* that the resource bundle needs to work.

```
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('bundle_name');

        $rootNode
            // Driver used by the resource bundle
            ->children()
                ->scalarNode('driver')->isRequired()->cannotBeEmpty()->end()
            ->end()

            // Validation groups used by the form component
            ->children()
                ->arrayNode('validation_groups')
                    ->addDefaultsIfNotSet()
                    ->children()
                        ->arrayNode('MyEntity')
                            ->prototype('scalar')->end()
                            ->defaultValue(array('your_group'))
                        ->end()
                    ->end()
                ->end()
            ->end()

            // The resources
            ->children()
                ->arrayNode('classes')
                    ->addDefaultsIfNotSet()
                    ->children()
                        ->arrayNode('MyEntity')
                            ->addDefaultsIfNotSet()
                            ->children()
                                ->scalarNode('model')->defaultValue('Sylius\Bundle\PromotionsBundle\M
                                ->scalarNode('controller')->defaultValue('Sylius\Bundle\ResourceBundl
                                ->scalarNode('repository')->end()
                                ->scalarNode('form')->defaultValue('Sylius\Bundle\PromotionsBundle\Fo
                            ->end()
                        ->end()
                    ->end()
```

```
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

The resource bundle provide you *AbstractResourceExtension*, your bundle extension have to extends it.

```
use Sylius\Bundle\ResourceBundle\DependencyInjection\AbstractResourceExtension;

class MyBundleExtension extends AbstractResourceExtension
{
    // You can choose your application name, it will use to prefix the configuration keys in the cont
    protected $applicationName = 'sylius';

    // You can define where yours service definitions are
    protected $configDirectory = '/../Resources/config';

    // You can define what service definitions you want to load
    protected $configFiles = array(
        'services',
        'forms',
    );

    public function load(array $config, ContainerBuilder $container)
    {
        $this->configure(
            $config,
            new Configuration(),
            $container,
            self::CONFIGURE_LOADER | self::CONFIGURE_DATABASE | self::CONFIGURE_PARAMETERS | self::CO
        );
    }
}
```

The last parameter of the *AbstractResourceExtension::configure()* allows you to define what functionalities you want to use :

- CONFIGURE_LOADER : load yours service definitions located in *$applicationName*
- CONFIGURE_PARAMETERS : set to the container the configured resource classes using the pattern *appName.serviceType.resourceName.class* For example : *sylius.controller.product.class*. For a form, it is a bit different : 'sylius.form.type.product.class'
- CONFIGURE_VALIDATORS : set to the container the configured validation groups using the pattern *appName.validation_group.modelName* For example *sylius.validation_group.product*
- CONFIGURE_DATABASE : Load the database driver, available drivers are *doctrine/orm*, *doctrine/mongodb-odm* and *doctrine/phpcr-odm*

And now, your bundle is configurable like that :

```
sylius_product:
    driver: doctrine/orm
    validation_groups:
        product: [sylius]
    classes:
        product:
```

```
model: Sylius\Bundle\CoreBundle\Model\Product
controller: Sylius\Bundle\CoreBundle\Controller\ProductController
repository: Sylius\Bundle\CoreBundle\Repository\ProductRepository
form: Sylius\Bundle\CoreBundle\Form\Type\ProductType
```

And... we're done!

This configuration registers for you several services and service aliases.

First of all, it gives you **app.manager.user**, which is simple alias to a proper **ObjectManager** service. For *doctrine/orm* it will be your default entity manager, and unless you want to stay completely storage agnostic, you can use the entity (or document) manager the "usual way".

Secondly, you get an **app.repository.user**. It represents repository. This service by default has a custom class, which implements `Sylius\\Bundle\\ResourceBundle\\Model\\RepositoryInterface` (which extends the Doctrine **ObjectRepository**).

The last and most important service is **app.controller.user**, you'll learn about it in next section.

Getting a single resource

Note: ResourceController class is built using FOSRestBundle, thus it's format agnostic, and can serve resources in many formats, like html, json or xml.

Your newly created controller service has a few basic crud actions and is configurable via routing, which allows you to do some really tedious tasks - easily.

The most basic action is **showAction**. It is used to display a single resource. To use it, the only thing you need to do is register a proper route.

```
# routing.yml

app_user_show:
    pattern: /users/{id}
    methods: [GET]
    defaults:
        _controller: app.controller.user:showAction
```

Done! Now when you go to `/users/3`, ResourceController will use the repository (`app.repository.user`) to find user with given id. If the requested user resource does not exist, it will throw a 404 exception.

When a user is found, the default template will be rendered - `App:User:show.html.twig` (like you configured it in *config.yml*) with the User result as the `user` variable. That's the most basic usage of the simple `showAction` action.

Using a custom template

Okay, but what if you want now to display same User resource, but with a different representation?

```
# routing.yml

app_backend_user_show:
    pattern: /backend/users/{id}
    methods: [GET]
    defaults:
        _controller: app.controller.user:showAction
```

```
        _sylius:
            template: App:Backend/User:show.html.twig
```

Nothing more to do here, when you go to `/backend/users/3`, the controller will try to find the user and render it using the custom template you specified under the route configuration. Simple, isn't it?

Overriding default criteria

Displaying the user by id can be boring... and let's say we do not want to allow viewing disabled users? There is a solution for that!

```
# routing.yml

app_user_show:
    pattern: /users/{username}
    methods: [GET]
    defaults:
        _controller: app.controller.user:showAction
        _sylius:
            criteria:
                username: $username
                enabled:  true
```

With this configuration, the controller will look for a user with the given username and exclude disabled users. Internally, it simply uses the `$repository->findOneBy(array $criteria)` method to look for the resource.

Using custom repository methods

By default, resource repository uses **findOneBy(array $criteria)**, but in some cases it's not enough - for example - you want to do proper joins or use a custom query. Creating yet another action to change the method called could be a solution but there is a better way. The configuration below will use a custom repository method to get the resource.

```
# routing.yml

app_user_show:
    pattern: /users/{username}
    methods: [GET]
    defaults:
        _controller: app.controller.user:showAction
        _sylius:
            method: findOneWithFriends
            arguments: [$username]
```

Internally, it simply uses the `$repository->findOneWithFriends($username)` method, where `username` is taken from the current request.

Getting a paginated (or flat) list of resources

To get a paginated list of users, we will use **indexAction** of our controller! In the default scenario, it will return an instance of paginator, with a list of Users.

```
# routing.yml

app_user_index:
    pattern: /users
```

```
    methods: [GET]
    defaults:
        _controller: app.controller.user:indexAction
```

When you go to `/users`, ResourceController will use the repository (`app.repository.user`) to create a paginator. The default template will be rendered - `App:User:index.html.twig` with the paginator as the `users` variable.

Overriding the template and criteria

Just like for the **showAction**, you can override the default template and criteria.

```
# routing.yml

app_user_index_inactive:
    pattern: /users/inactive
    methods: [GET]
    defaults:
        _controller: app.controller.user:indexAction
        _sylius:
            criteria:
                enabled: false
            template: App:User:inactive.html.twig
```

This action will render a custom template with a paginator only for disabled users.

Sorting collection or paginator

Except filtering, you can also sort users.

```
# routing.yml

app_user_index_top:
    pattern: /users/top
    methods: [GET]
    defaults:
        _controller: app.controller.user:indexAction
        _sylius:
            sorting:
                score: desc
            template: App:User:top.html.twig
```

Under that route, you can paginate over the users by their score.

Changing the "max per page" option of paginator

You can also control the "max per page" for paginator, using `paginate` parameter.

```
# routing.yml

app_user_index_top:
    pattern: /users/top
    methods: [GET]
    defaults:
        _controller: app.controller.user:indexAction
```

```
        _sylius:
            paginate: 5
            sorting:
                score: desc
            template: App:User:top.html.twig
```

This will paginate users by 5 per page, where 10 is the default.

Disabling pagination - getting flat list

Pagination is handy, but you do not always want to do it, you can disable pagination and simply request a collection of resources.

```
# routing.yml

app_user_index_top3:
    pattern: /users/top
    methods: [GET]
    defaults:
        _controller: app.controller.user:indexAction
        _sylius:
            paginate: false
            limit: 3
            sorting:
                score: desc
            template: App:User:top3.html.twig
```

That action will return the top 3 users by score, as the `users` variable.

Twig Extensions

sylius_resource_sort

Parameters :

- **property (string) :** [Mandatory] Name of the property (defined in your resource)
- **label (string) :** Label of the column on your grid (default : property name)
- **order (string) :** Default order, it can be asc or desc (default : asc)
- **options (array) : Additional options, the extension can use a custom template or generate a custom route**
  - **template (string) :** Path to the template
  - **route (string) :** Key of the new route
  - **route_params (array) :** Additional route parameters

This extension renders the following template : SyliusResourceBundle:Twig:paginate.html.twig

Example :

```
<div>
    {{ sylius_resource_sort('productId', 'product.id'|trans) }}
</div>
```

sylius_resource_paginate

Parameters :

- **paginator (object) :** [Mandatory] An instance of PagerFanta
- **limits (array) :** [Mandatory] An array of paginate value
- **options (array) : Additional options, the extension can use a custom template or generate a custom route**
  - **template (string) :** Path to the template
  - **route (string) :** Key of the new route
  - **route_params (array) :** Additional route parameters

This extension renders the following template : SyliusResourceBundle:Twig:sorting.html.twig

Example :

```
<div>
    {{ sylius_resource_paginate(paginator, [10, 20, 30]) }}
</div>
```

Creating new resource

To display a form, handle submit or create a new resource via API, you should use **createAction** of your **app.controller.user** service.

```
# routing.yml

app_user_create:
    pattern: /users/new
    methods: [GET, POST]
    defaults:
        _controller: app.controller.user:createAction
```

Done! Now when you go to `/users/new`, ResourceController will use the repository (`app.repository.user`) to create new user instance. Then it will try to create an `app_user` form, and set the newly created user as data.

Note: Currently, this bundle does not generate a form for you, the right form type has to be created and registered in the container manually.

As a response, it will render the `App:User:create.html.twig` template with form view as the `form` variable.

Submitting the form

You can use exactly the same route to handle the submit of the form and create the user.

```
<form method="post" action="{{ path('app_user_create') }}">
```

On submit, the create action with method POST, will bind the request on the form, and if it is valid it will use the right manager to persist the resource. Then, by default it redirects to `app_user_show` to display the created user, but you can easily change that behavior - you'll see this in later sections.

When validation fails, it will render the form just like previously with the errors to display.

Changing the template

Just like for the **show** and **index** actions, you can customize the template per route.

```
# routing.yml

app_user_create:
    pattern: /users/new
    methods: [GET, POST]
    defaults:
        _controller: app.controller.user:createAction
        _sylius:
            template: App:Backend/User:create.html.twig
```

Using different form

You can also use custom form type on per route basis. By default it generates the form type name following the simple convention `bundle prefix + _ + resource logical name`. Below you can see the usage for specifying a custom form.

```
# routing.yml

app_user_create:
    pattern: /users/new
    methods: [GET, POST]
    defaults:
        _controller: app.controller.user:createAction
        _sylius:
            template: App:Backend/User:create.html.twig
            form: app_user_custom
```

Using custom factory method

By default, `ResourceController` will use the `createNew` method with no arguments to create a new instance of your object. However, this behavior can be modified. To use different method of your repository, you can simply configure the `factory` option.

```
# routing.yml

app_user_create:
    pattern: /users/new
    methods: [GET, POST]
    defaults:
        _controller: app.controller.user:createAction
        _sylius:
            factory: createNewWithGroups
```

Additionally, if you want to provide your custom method with arguments from the request, you can do so by adding more parameters.

```
# routing.yml

app_user_create:
    pattern: /users/{groupId}/new
    methods: [GET, POST]
```

```
    defaults:
        _controller: app.controller.user:createAction
        _sylius:
            factory:
                method: createNewWithGroups
                arguments: [$groupId]
```

With this configuration, `$repository->createNewWithGroups($request->get('groupId'))` will be called to create new resource within `createAction`.

Custom redirect after success

By default the controller will try to get the id of the newly created resource and redirect to the "show" route. You can easily change that. For example, to redirect user to list after successfully creating a new resource - you can use the following configuration.

```
# routing.yml

app_user_create:
    pattern: /users/new
    methods: [GET, POST]
    defaults:
        _controller: app.controller.user:createAction
        _sylius:
            redirect: app_user_index
```

You can also perform more complex redirects, with parameters. For example...

```
# routing.yml

app_user_create:
    pattern: /competition/{competitionId}/users/new
    methods: [GET, POST]
    defaults:
        _controller: app.controller.user:createAction
        _sylius:
            redirect:
                route: app_competition_show
                parameters: { id: $competitionId }
```

In addition to the request parameters, you can access some of the newly created objects properties, using the `resource.` prefix.

```
# routing.yml

app_user_create:
    pattern: /users/new
    methods: [GET, POST]
    defaults:
        _controller: app.controller.user:createAction
        _sylius:
            redirect:
                route: app_user_show
                parameters: { email: resource.email }
```

With this configuration, the `email` parameter for route `app_user_show` will be obtained from your newly created user.

Updating existing resource

To display an edit form of a particular resource, change it or update it via API, you should use the **updateAction** action of your **app.controller.user** service.

```
# routing.yml

app_user_update:
    pattern: /users/{id}/edit
    methods: [GET, PUT]
    defaults:
        _controller: app.controller.user:updateAction
```

Done! Now when you go to `/users/5/edit`, ResourceController will use the repository (`app.repository.user`) to find the user with ID === **5**. If found it will create the `app_user` form, and set the existing user as data.

Note: Currently, this bundle does not generate a form for you, the right form type has to be updated and registered in the container manually.

As a response, it will render the `App:User:update.html.twig` template with form view as the `form` variable and the existing User as the `user` variable.

Submitting the form

You can use exactly the same route to handle the submit of the form and update the user.

```
<form method="post" action="{{ path('app_user_update', {'id': user.id}) }}">
    <input type="hidden" name="_method" value="PUT" />
```

On submit, the update action with method PUT, will bind the request on the form, and if it is valid it will use the right manager to persist the resource. Then, by default it redirects to `app_user_show` to display the updated user, but like for creation of the resource - it's customizable.

When validation fails, it will simply render the form again.

Changing the template

Just like for other actions, you can customize the template.

```
# routing.yml

app_user_update:
    pattern: /users/{id}/edit
    methods: [GET, PUT]
    defaults:
        _controller: app.controller.user:updateAction
        _sylius:
            template: App:Backend/User:update.html.twig
```

Using different form

Same way like for **createAction** you can override the default form.

```yaml
# routing.yml

app_user_update:
    pattern: /users/{id}/edit
    methods: [GET, PUT]
    defaults:
        _controller: app.controller.user:updateAction
        _sylius:
            template: App:Backend/User:update.html.twig
            form: app_user_custom
```

Overriding the criteria

By default, the **updateAction** will look for the resource by id. You can easily change that criteria.

```yaml
# routing.yml

app_user_update:
    pattern: /users/{username}/edit
    methods: [GET, PUT]
    defaults:
        _controller: app.controller.user:updateAction
        _sylius:
            criteria: { username: $username }
```

Custom redirect after success

By default the controller will try to get the id of resource and redirect to the "show" route. To change that, use the following configuration.

```yaml
# routing.yml

app_user_update:
    pattern: /users/{id}/edit
    methods: [GET, PUT]
    defaults:
        _controller: app.controller.user:updateAction
        _sylius:
            redirect: app_user_index
```

You can also perform more complex redirects, with parameters. For example...

```yaml
# routing.yml

app_user_update:
    pattern: /competition/{competitionId}/users/{id}/edit
    methods: [GET, PUT]
    defaults:
        _controller: app.controller.user:updateAction
        _sylius:
            redirect:
                route: app_competition_show
                parameters: { id: $competitionId }
```

Deleting resource

Deleting a resource is simple.

```
# routing.yml

app_user_delete:
    pattern: /users/{id}
    methods: [DELETE]
    defaults:
        _controller: app.controller.user:deleteAction
```

Calling the action with method DELETE

Currently browsers do not support the "DELETE" http method. Fortunately, Symfony has a very useful feature. You can make a POST call with override parameter, which will force the framework to treat the request as specified method.

```
<form method="post" action="{{ path('app_user_delete', {'id': user.id}) }}">
    <input type="hidden" name="_method" value="DELETE" />
```

On submit, the delete action with the method DELETE, will remove and flush the resource. Then, by default it redirects to `app_user_index` to display the users index, but like for other actions - it's customizable.

Overriding the criteria

By default, the **deleteAction** will look for the resource by id. However, you can easily change that. For example, you want to delete the user who belongs to particular company, not only by his id.

```
# routing.yml

app_user_delete:
    pattern: /companies/{companyId}/users/{id}
    methods: [DELETE]
    defaults:
        _controller: app.controller.user:deleteAction
        _sylius:
            criteria:
                id:      $id
                company: $companyId
```

There are no magic hacks behind that, it simply takes parameters from request and builds the criteria array for the `findOneBy` repository method.

Custom redirect after success

By default the controller will try to get the id of the resource and redirect to the "index" route. To change that, use the following configuration.

```
# routing.yml

app_user_delete:
    pattern: /competition/{competitionId}/users/{id}
    methods: [DELETE]
    defaults:
        _controller: app.controller.user:deleteAction
```

```
        _sylius:
            redirect:
                route: app_competition_show
                parameters: { id: $competitionId }
```

Custom actions

Note: To be written.

Summary

Configuration reference

```
sylius_resource:
    resources:
        app.user:
            driver: doctrine/orm # Also supported - doctrine/mongodb-odm.
            templates: AppBundle:User
            classes:
                model: App\Entity\User
                controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
                repository: Sylius\Bundle\ResourceBundle\Doctrine\ORM\EntityRepository
```

phpspec2 examples

```
$ composer install --dev --prefer-dist
$ bin/phpspec run -fpretty --verbose
```

Bug tracking

This bundle uses GitHub issues. If you have found bug, please create an issue.

4.1.3 SyliusProductBundle

The Sylius product catalog is made available as set of 2 standalone bundles. This component contains the most basic product model with properties (attributes) support. It also includes the product prototyping system, which serves as a template for creating similar products.

If you need product options support, please see `SyliusVariableProductBundle` which allows to manage product variations with a very flexible architecture.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download the package.

If you have Composer installed globally.

```
$ composer require "sylius/product-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/product-bundle"
```

Adding required bundles to the kernel

You need to enable the bundle inside the kernel.

If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies to kernel. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),
        new Sylius\Bundle\ProductBundle\SyliusProductBundle(),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    );
}
```

Note: Please register the bundle before *DoctrineBundle*. This is important as we use listeners which have to be processed first.

Container configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_product:
    driver: doctrine/orm # Configure the doctrine orm driver used in the documentation.
```

And configure doctrine extensions which are used by the bundle.

```
stof_doctrine_extensions:
    orm:
        default:
            sluggable: true
            timestampable: true
```

Routing configuration

Add the following to your `app/config/routing.yml`.

```
sylius_product:
    resource: @SyliusProductBundle/Resources/config/routing.yml
```

Updating database schema

Run the following command.

```
$ php app/console doctrine:schema:update --force
```

> **Warning:** This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

Congratulations! The bundle is now installed and ready to use.

The Product

Product is the main model in SyliusProductBundle. This simple class represents every unique product in the catalog. The default interface contains the following attributes with appropriate setters and getters.

Attribute	Description
id	Unique id of the product
name	Name of the product
slug	SEO slug, by default created from the name
description	Description of your great product
availableOn	Date when product becomes available in catalog
metaDescription	Description for search engines
metaKeywords	Comma separated list of keywords for product (SEO)
createdAt	Date when product was created
updatedAt	Date of last product update
deletedAt	Date of deletion from catalog

Retrieving products

Retrieving a product from the database should always happen via repository, which always implements `Sylius\Bundle\ResourceBundle\Model\RepositoryInterface`. If you are using Doctrine, you're already familiar with this concept, as it extends the native Doctrine `ObjectRepository` interface.

Your product repository is always accessible via the `sylius.repository.product` service.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');
}
```

Retrieving products is simple as calling proper methods on the repository.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');
```

```
    $product = $repository->find(4); // Get product with id 4, returns null if not found.
    $product = $repository->findOneBy(array('slug' => 'my-super-product')); // Get one product by de

    $products = $repository->findAll(); // Load all the products!
    $products = $repository->findBy(array('special' => true)); // Find products matching some custom
}
```

Product repository also supports paginating products. To create a Pagerfanta instance use the `createPaginator` method.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');

    $products = $repository->createPaginator();
    $products->setMaxPerPage(3);
    $products->setCurrentPage($request->query->get('page', 1));

   // Now you can return products to the template and iterate over it to get products from the curre
}
```

The paginator also can be created for specific criteria and with desired sorting.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');

    $products = $repository->createPaginator(array('foo' => true), array('createdAt' => 'desc'));
    $products->setMaxPerPage(3);
    $products->setCurrentPage($request->query->get('page', 1));
}
```

Creating new product object

To create new product instance, you can simply call `createNew()` method on the repository.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');
    $product = $repository->createNew();
}
```

Note: Creating a product via this factory method makes the code more testable, and allows you to change the product class easily.

Saving & removing product

To save or remove a product, you can use any `ObjectManager` which manages Product. You can always access it via alias `sylius.manager.product`. But it's also perfectly fine if you use `doctrine.orm.entity_manager`

or other appropriate manager service.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');
    $manager = $this->container->get('sylius.manager.product'); // Alias for appropriate doctrine ma

    $product = $repository->createNew();

    $product
        ->setName('Foo')
        ->setDescription('Nice product')
    ;

    $manager->persist($product);
    $manager->flush(); // Save changes in database.
}
```

To remove a product, you also use the manager.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.product');
    $manager = $this->container->get('sylius.manager.product');

    $product = $repository->find(1);

    $manager->remove($product);
    $manager->flush(); // Save changes in database.
}
```

Properties

A product can also have a set of defined Properties (think Attributes), you'll learn about them in next chapter of this documentation.

Product Properties

Except products, you can also define Properties (think Attributes) and define their values on each product. Default property model has following structure.

Attribute	Description
id	Unique id of the property
name	Name of the property ("T-Shirt Material")
presentation	Pretty name visible for user ("Material")
type	Property type
createdAt	Date when property was created
updatedAt	Date of last property update

Currently there are several different property types are available, a proper form widget (Symfony Form type) will be rendered on product form for entering the value.

Type
text
number
percentage
checkbox
choice

Managing Properties

Managing properties happens exactly the same way like products, you have `sylius.repository.property` and `sylius.manager.property` at your disposal.

Assigning properties to product

Value of specific Property for one of Products, happens through ProductProperty model, which holds the references to Product, Property pair and the value. If you want to programatically set a property value on product, use the following code.

```
<?php

public function myAction(Request $request)
{
    $propertyRepository = $this->container->get('sylius.repository.property');
    $productPropertyRepository = $this->container->get('sylius.repository.product_property');

    $property = $propertyRepository->findOneBy(array('name' => 'T-Shirt Collection'));
    $productProperty = $productPropertyRepository->createNew();

    $productProperty
        ->setProperty($property)
        ->setValue('Summer 2013')
    ;

    $product->addProperty($productProperty);

    $manager = $this->container->get('sylius.manager.product');

    $manager->persist($product);
    $manager->flush(); // Save changes in database.
}
```

This looks a bit tedious, doesn't it? There is a **ProductBuilder** service which simplifies the creation of products dramatically, you can learn about it in appropriate chapter.

Forms

The bundle ships with a set of useful form types for all models. You can use the defaults or `override them` with your own forms.

Product form

The product form type is named `sylius_product` and you can create it whenever you need, using the form factory.

```php
<?php

// src/Acme/ShopBundle/Controller/ProductController.php

namespace Acme\ShopBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class DemoController extends Controller
{
    public function fooAction(Request $request)
    {
        $form = $this->get('form.factory')->create('sylius_product');
    }
}
```

The default product form consists of following fields.

Field	Type
name	text
description	textarea
availableOn	datetime
metaDescription	text
metaKeywords	text

You can render each of these using the usual Symfony way `{{ form_row(form.description) }}`.

Property form

Default form for the Property model has name `sylius_property` and contains several basic fields.

Field	Type
name	text
presentation	text
type	choice

Prototype form

The default form for the Prototype model has name `sylius_prototype` and is built from the following fields.

Field	Type
name	text
properties	sylius_property_choice

Miscellaneous fields

There are a few more form types, which can become useful when integrating the bundle into your app.

`sylius_product_property` is a form which is used to set the product properties (and their values). It has 2 fields, the property choice field and a value input.

`sylius_property_choice` is a ready-to-use select field, with a list of all Properties from database.

`sylius_product_to_identifier` can be used to render a text field, which will transform the value into a product.

If you need to customize existing fields or add your own, please read the `overriding forms chapter`.

Prototypes

...

Prototype Builder

Used to build product based on given prototype.

Here is an example:

```
<?php

$prototype = $this->findOr404(array('id' => $prototypeId));
$product = $this->get('sylius.repository.product')->createNew();

$this
    ->get('sylius.builder.prototype')
    ->build($prototype, $product)
;
```

It will add appropriate options and variants to given product based on prototype.

Product Builder

This service provides a fluent interface for easy product creation.

The example shown below is self explanatory:

```
<?php

$product = $this->get('sylius.builder.product')
    ->create('Github mug')
    ->setDescription("Coffee. Let's face it -- humans need to drink liquids!")
    ->addProperty('collection', 2013)
    ->addProperty('color', 'Red')
    ->addProperty('material', 'Stone')
    ->save()
;
```

Configuration reference

```
sylius_product:
    driver: ~ # The driver used for persistence layer.
    engine: twig # Templating engine to use by default.
    classes:
        product:
            model: ~ # The product model class.
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AssortmentBundle\Form\Type\ProductType
        property:
            model: Sylius\Bundle\ProductBundle\Model\Property
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
```

```
            repository: ~
            form: Sylius\Bundle\AssortmentBundle\Form\Type\PropertyType
        prototype:
            model: Sylius\Bundle\ProductBundle\Model\Prototype
            controller: Sylius\Bundle\ProductBundle\Controller\PrototypeController
            repository: ~
            form: Sylius\Bundle\AssortmentBundle\Form\Type\PrototypeType
```

4.1.4 SyliusCartBundle

A generic solution for a cart system inside a Symfony2 application.

It doesn't matter if you are starting a new project or you need to implement this feature for an existing system - this bundle should be helpful. Currently only the Doctrine ORM driver is implemented, so we'll use it here as an example.

There are two main models inside the bundle, *Cart* and *CartItem*.

There are also 2 main services, **Provider** and **ItemResolver**. You'll get familiar with them in further parts of the documentation.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require "sylius/cart-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/cart-bundle"
```

Adding required bundles to the kernel

First, you need to enable the bundle inside the kernel. If you're not using any other Sylius bundles, you will also need to add the following bundles and their dependencies to the kernel:

- *SyliusResourceBundle*
- *SyliusMoneyBundle*
- *SyliusOrderBundle*

Don't worry, everything was automatically installed via Composer.

Note: Please register the bundle **before** *DoctrineBundle*. This is important as we use listeners which have to be processed first. It is generally a good idea to place all of the Sylius bundles at the beginning of the bundles list, as it is done in the *Sylius-Standard* project.

```
<?php

// app/AppKernel.php

public function registerBundles()
```

```
{
    $bundles = array(
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),
        new Sylius\Bundle\MoneyBundle\SyliusMoneyBundle(),
        new Sylius\Bundle\OrderBundle\SyliusOrderBundle(),
        new Sylius\Bundle\CartBundle\SyliusCartBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
    );
}
```

Creating your entities

This is no longer a required step in the latest version of the *SyliusCartBundle*, and if you are happy with the default implementation (which is `Sylius\Bundle\CartBundle\Model\CartItem`), you can just skip to the next section.

You can create your **CartItem** entity, living inside your application code. We think that **keeping the application-specific and simple bundle structure** is a good practice, so let's assume you have your `AppBundle` registered under `App\AppBundle` namespace.

```
<?php

// src/App/AppBundle/Entity/CartItem.php
namespace App\AppBundle\Entity;

use Sylius\Bundle\CartBundle\Model\CartItem as BaseCartItem;

class CartItem extends BaseCartItem
{
}
```

Now we need to define a simple mapping for this entity to map its fields. You should create a mapping file in your `AppBundle`, put it inside the doctrine mapping directory `src/App/AppBundle/Resources/config/doctrine/CartItem.orm.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<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

    <entity name="App\AppBundle\Entity\CartItem" table="app_cart_item">
    </entity>

</doctrine-mapping>
```

You do **not** have to map the *ID* field because it is already mapped in the `Sylius\Bundle\CartBundle\Model\CartItem` class, together with the relation between **Cart** and **CartItem**.

Let's assume you have a *Product* entity, which represents your main merchandise within your webshop.

Note: Please remember that you can use anything else, *Product* here is just an obvious example, but it will work in a similar way with other entities.

We need to modify the *CartItem* entity and its mapping a bit, so it allows us to put a product inside the cart item.

```
<?php

// src/App/AppBundle/Entity/CartItem.php
namespace App\AppBundle\Entity;

use Sylius\Bundle\CartBundle\Model\CartItem as BaseCartItem;

class CartItem extends BaseCartItem
{
    private $product;

    public function getProduct()
    {
        return $this->product;
    }

    public function setProduct(Product $product)
    {
        $this->product = $product;
    }
}
```

We added a "product" property, and a simple getter and setter. We have to also map the *Product* to *CartItem*, let's create this relation in mapping files.

```
<?xml version="1.0" encoding="UTF-8"?>

<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

    <entity name="App\AppBundle\Entity\CartItem" table="app_cart_item">
        <many-to-one field="product" target-entity="App\AppBundle\Entity\Product">
            <join-column name="product_id" referenced-column-name="id" />
        </many-to-one>
    </entity>

</doctrine-mapping>
```

Similarly, you can create a custom entity for orders. The class that you need to extend is `Sylius\Bundle\CartBundle\Model\Cart`. Carts and Orders in Sylius are in fact the same thing. Do not forget to create the mapping file. But, again, do not put a mapping for the *ID* field — it is already mapped in the parent class.

And that would be all about entities. Now we need to create a really simple service.

Creating ItemResolver service

The **ItemResolver** will be used by the controller to resolve the new cart item - based on a user request information. Its only requirement is to implement `Sylius\Bundle\CartBundle\Resolver\ItemResolverInterface`.

```php
<?php

// src/App/AppBundle/Cart/ItemResolver.php
namespace App\AppBundle\Cart;

use Sylius\Bundle\CartBundle\Model\CartItemInterface;
use Sylius\Bundle\CartBundle\Resolver\ItemResolverInterface;
use Symfony\Component\HttpFoundation\Request;

class ItemResolver implements ItemResolverInterface
{
    public function resolve(CartItemInterface $item, Request $request)
    {
    }
}
```

The class is in place, well done.

We need to do some more coding, so the service is actually doing its job. In our example we want to put *Product* in our cart, so we should inject the entity manager into our resolver service.

```php
<?php

// src/App/AppBundle/Cart/ItemResolver.php
namespace App\AppBundle\Cart;

use Sylius\Bundle\CartBundle\Model\CartItemInterface;
use Sylius\Bundle\CartBundle\Resolver\ItemResolverInterface;
use Symfony\Component\HttpFoundation\Request;
use Doctrine\ORM\EntityManager;

class ItemResolver implements ItemResolverInterface
{
    private $entityManager;

    public function __construct(EntityManager $entityManager)
    {
        $this->entityManager = $entityManager;
    }

    public function resolve(CartItemInterface $item, Request $request)
    {
    }

    private function getProductRepository()
    {
        return $this->entityManager->getRepository('AppBundle:Product');
    }
}
```

We also added a simple method `getProductRepository()` to keep the resolving code cleaner.

We must use this repository to find a product with *id*, given by the user via the request. This can be done in various ways, but to keep the example simple - we'll use a query parameter.

```php
<?php

// src/App/AppBundle/Cart/ItemResolver.php
namespace App\AppBundle\Cart;
```

```php
use Sylius\Bundle\CartBundle\Model\CartItemInterface;
use Sylius\Bundle\CartBundle\Resolver\ItemResolverInterface;
use Sylius\Bundle\CartBundle\Resolver\ItemResolvingException;
use Symfony\Component\HttpFoundation\Request;
use Doctrine\ORM\EntityManager;

class ItemResolver implements ItemResolverInterface
{
    private $entityManager;

    public function __construct(EntityManager $entityManager)
    {
        $this->entityManager = $entityManager;
    }

    public function resolve(CartItemInterface $item, Request $request)
    {
        $productId = $request->query->get('productId');

        // If no product id given, or product not found, we throw exception with nice message.
        if (!$productId || !$product = $this->getProductRepository()->find($productId)) {
            throw new ItemResolvingException('Requested product was not found');
        }

        // Assign the product to the item and define the unit price.
        $item->setProduct($product);
        $item->setUnitPrice($product->getPrice());

        // Everything went fine, return the item.
        return $item;
    }

    private function getProductRepository()
    {
        return $this->entityManager->getRepository('AppBundle:Product');
    }
}
```

Note: Please remember that **item accepts only integers as price and quantity**.

Register our brand new service in the container. We'll use XML as an example, but you are free to pick any other format.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<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/services-1.0.xsd">

    <services>
        <service id="app.cart_item_resolver" class="App\AppBundle\Cart\ItemResolver">
            <argument type="service" id="doctrine.orm.entity_manager" />
        </service>
    </services>
</container>
```

The bundle requires also a simple configuration...

Container configuration

Put this minimal configuration inside your `app/config/config.yml`.

```
sylius_cart:
    resolver: app.cart_item_resolver # The id of our newly created service.
    classes: ~ # This key can be empty but it must be present in the configuration.

sylius_order:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.

sylius_money:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.
```

Or, if you have created any custom entities, use this:

```
sylius_cart:
    resolver: app.cart_item_resolver # The id of our newly created service.
    classes: ~ # This key can be empty but it must be present in the configuration.

sylius_order:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.
    classes:
        order
            model: App\AppBundle\Entity\Cart # If you have created a custom Cart entity.
        order_item:
            model: App\AppBundle\Entity\CartItem # If you have created a custom CartItem entity.

sylius_money:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.
```

Importing routing configuration

Import the default routing from your `app/config/routing.yml`.

```
sylius_cart:
    resource: @SyliusCartBundle/Resources/config/routing.yml
    prefix: /cart
```

Updating database schema

Remember to update your database schema.

For "**doctrine/orm**" driver run the following command.

```
$ php app/console doctrine:schema:update --force
```

> **Warning:** This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

Templates

We think that providing a sensible default template is really difficult, especially when a cart summary is not the simplest page. This is the reason why we do not currently include any, but if you have an idea for a good starter template, let us know!

The bundle requires only the `show.html` template for cart summary page. The easiest way to override the view is by placing it here `app/Resources/SyliusCartBundle/views/Cart/show.html.twig`.

Note: You can use the templates from our Sylius app as inspiration.

The Cart and CartItem

Here is a quick reference of what the default models can do for you.

Cart

You can access the cart total value using the `->getTotal()` method. The denormalized number of cart items is available via `->getTotalItems()` method. Recalculation of totals can happen by calling `->calculateTotal()` method, using the simplest possible math. It will also update the item totals. The carts have their expiration time - `->getExpiresAt()` returns that time and `->incrementExpiresAt()` sets it to +3 hours from now by default. The collection of items (Implementing the `Doctrine\Common\Collections\Collection` interface) can be obtained using the `->getItems()`.

CartItem

Just like for the cart, the total is available via the same method (`->getTotal()`), but the unit price is accessible using the `->getUnitPrice()` Each item also can calculate its total, using the quantity (`->getQuantity()`) and the unit price. It also has a very important method called `->equals(CartItemInterface $item)`, which decides whether the items are "same" or not. If they are, it should return *true*, *false* otherwise. This is taken into account when adding an item to the cart. **If the added item is equal to an existing one, their quantities are summed, but no new item is added to the cart**. By default, it compares the ids, but for our example we would prefer to check the products. We can easily modify our *CartItem* entity to do that correctly.

```
<?php

// src/App/AppBundle/Entity/CartItem.php
namespace App/AppBundle/Entity;

use Sylius\Bundle\CartBundle\Model\CartItem as BaseCartItem;
use Sylius\Bundle\CartBundle\Model\CartItemInterface;

class CartItem extends BaseCartItem
{
    private $product;

    public function getProduct()
    {
        return $this->product;
    }

    public function setProduct(Product $product)
    {
```

```
        $this->product = $product;
    }

    public function equals(CartItemInterface $item)
    {
        return $this->product === $item->getProduct();
    }
}
```

If the user tries to add the same product twice or more, it will just sum the quantities, instead of adding duplicates to the cart.

Routing and default actions

This bundle provides a quite simple default routing with several handy and common actions. You can see the usage guide below.

Cart summary page

To point user to the cart summary page, you can use the `sylius_cart_summary` route. It will render the page with the *cart* and *form* variables by default.

The *cart* is the current cart and *form* is the view of the cart form.

Adding cart item

In our simple example, we only need to add the following link in the places where we need the "add to cart button".

```
<a href="{{ path('sylius_cart_item_add', {'productId': product.id}) }}">Add product to cart</a>
```

Clicking this link will add the selected product to the cart.

Removing item

On the cart summary page you have access to all the cart items, so another simple link will allow a user to remove items from the cart.

```
<a href="{{ path('sylius_cart_item_remove', {'id': item.id}) }}">Remove from cart</a>
```

Where *item* variable represents one of the *cart.items* collection items.

Clearing the whole cart

Clearing the cart is simple as clicking the following link.

```
<a href="{{ path('sylius_cart_clear')}}">Clear cart</a>
```

Basic cart update

On the cart summary page, you have access to the cart form, if you want to save it, simply submit the form with the following action.

```
<form action="{{ path('sylius_cart_save') }}" method="post">Save cart</a>
```

You cart will be validated and saved if everything is alright.

Using the services

When using the bundle, you have access to several handy services. You can use them to manipulate and manage the cart.

Managers and Repositories

Note: Sylius uses `Doctrine\Common\Persistence` interfaces.

You have access to following services which are used to manage and retrieve resources.

This set of default services is shared across almost all Sylius bundles, but this is just a convention. You're interacting with them like you usually do with own entities in your project.

```
<?php

// ...
public function saveAction(Request $request)
{
    // ObjectManager which is capable of managing the Cart resource.
    // For *doctrine/orm* driver it will be EntityManager.
    $this->get('sylius.manager.cart');

    // ObjectRepository for the Cart resource, it extends the base EntityRepository.
    // You can use it like usual entity repository in project.
    $this->get('sylius.repository.cart');

    // Same pair for CartItem resource.
    $this->get('sylius.manager.cart_item');
    $this->get('sylius.repository.cart_item');

    // Those repositories have some handy default methods, for example...
    $item = $this->get('sylius.repository.cart')->createNew();
}
```

Provider and Resolver

There are also 3 more services for you.

You use the provider to obtain the current user cart, if there is none, a new one is created and saved. The `->setCart()` method also allows you to replace the current cart. `->abandonCart()` is resetting the current cart, a new one will be created on the next `->getCart()` call. This is useful, for example, when after completing an order you want to start with a brand new and clean cart.

```
<?php

// ...
public function saveAction(Request $request)
{
    $provider = $this->get('sylius.cart_provider'); // Implements the CartProviderInterface.
```

```
    $currentCart = $provider->getCart();
    $provider->setCart($customCart);
    $provider->abandonCart();
}
```

The resolver is used to create a new item based on the user request.

```
<?php

// ...
public function addItemAction(Request $request)
{
    $resolver = $this->get('sylius.cart_resolver');
    $item = $resolver->resolve($this->createNew(), $request);
}
```

Note: A more advanced example of a resolver implementation is available in Sylius Sandbox application on GitHub.

In templates

When using Twig as your template engine, you have access to 2 handy functions.

The `sylius_cart_get` function uses the provider to get the current cart.

```
{% set cart = sylius_cart_get() %}

Current cart totals: {{ cart.total }} for {{ cart.totalItems }} items!
```

The `sylius_cart_form` returns the form view for the CartItem form. It allows you to easily build more complex actions for adding items to cart. In this simple example we allow to provide the quantity of item. You'll need to process this form in your resolver.

```
{% set form = sylius_cart_form({'product': product}) %} {# You can pass options as an argument. #}

<form action="{{ path('sylius_cart_item_add', {'productId': product.id}) }}" method="post">
    {{ form_row(form.quantity)}}
    <input type="submit" value="Add to cart">
</form>
```

Note: An example with multiple variants of this form can be found in Sylius Sandbox app. It allows for selecting a variation/options/quantity of the product. It also adapts to the product type.

Summary

Configuration reference

```
sylius_cart:
    driver: ~ # The driver used for persistence layer.
    engine: twig # Templating engine to use by default.
    resolver: ~ # Service id of cart item resolver.
    provider: sylius.cart_provider.default # Cart provider service id.
    storage: sylius.cart_storage.session # The id of cart storage for default provider.
    classes:
        cart:
```

```
            model: ~ # The cart model class.
            controller: Sylius\Bundle\CartBundle\Controller\CartController
            repository: ~ # You can override the repository class here.
            form: Sylius\Bundle\CartBundle\Form\Type\CartType # The form type name to use.
        item:
            model: ~ # The cart item model class.
            controller: Sylius\Bundle\CartBundle\Controller\CartItemController
            repository: ~ # You can override the repository class here.
            form: Sylius\Bundle\CartBundle\Form\Type\CartItemType # The form type class name to use.
    validation_groups:
        cart: [sylius] # Cart validation groups.
        item: [sylius] # Cart item validation groups.
```

phpspec2 examples

```
$ composer install --dev --prefer-dist
$ bin/phpspec run -f pretty
```

Working examples

If you want to see working implementation, try out the Sylius sandbox application.

Bug tracking

This bundle uses GitHub issues. If you have found bug, please create an issue.

4.1.5 SyliusOrderBundle

This bundle is a foundation for sales order handling for Symfony2 projects. It allows you to use any model as the merchandise.

It also includes a super flexible adjustments feature, which serves as a basis for any taxation, shipping charges or discounts system.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/order-bundle:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/order-bundle:*
```

Adding required bundles to the kernel

First, you need to enable the bundle inside the kernel.

If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies to the kernel. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),

        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),
        new Sylius\Bundle\MoneyBundle\SyliusMoneyBundle(),
        new Sylius\Bundle\OrderBundle\SyliusOrderBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    );
}
```

Note: Please register the bundle before *DoctrineBundle*. This is important as we use listeners which have to be processed first.

Creating your entities

You have to create your own **Order** entity, living inside your application code. We think that **keeping the app-specific bundle structure simple** is a good practice, so let's assume you have your `AppBundle` registered under `App\Bundle\AppBundle` namespace.

```
<?php

// src/App/AppBundle/Entity/Order.php
namespace App\AppBundle\Entity;

use Sylius\Component\Order\Model\Order as BaseOrder;

class Order extends BaseOrder
{
}
```

Now we need to define simple mapping for this entity, because it only extends the Doctrine mapped super-class. You should create a mapping file in your `AppBundle`, put it inside the doctrine mapping directory `src/App/AppBundle/Resources/config/doctrine/Order.orm.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<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

    <entity name="App\AppBundle\Entity\Order" table="app_order">
        <id name="id" column="id" type="integer">
            <generator strategy="AUTO" />
        </id>
        <one-to-many field="items" target-entity="Sylius\Component\Order\Model\OrderItemInterface" ma
            <cascade>
                <cascade-all/>
            </cascade>
        </one-to-many>

        <one-to-many field="adjustments" target-entity="Sylius\Component\Order\Model\AdjustmentInter
            <cascade>
                <cascade-all/>
            </cascade>
        </one-to-many>
    </entity>

</doctrine-mapping>
```

Note: You might wonder why are we putting interface inside mapping, you can read about this Doctrine feature here.

Now let's assume you have a *Product* entity, which represents your main merchandise in your webshop.

Note: Please remember that you can use anything else, *Product* here is just an obvious example, but it will work in similar way with other entities.

All you need to do is making your *Product* entity to implement `ProductInterface` and configure it inside Symfony settings.

```php
<?php

// src/App/AppBundle/Entity/Product.php
namespace App\AppBundle\Entity;

use Sylius\Component\Product\Model\ProductInterface;

class Product implements ProductInterface
{
    // Your code...

    public function getName()
    {
        // Here you just have to return the nice display name of your merchandise.
        return $this->name;
    }
}
```

Now, you do not even have to map your *Product* model to the order items. It is all done automatically. And that would be all about entities.

Container configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_order:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.
    classes:
        order:
            model: App\AppBundle\Entity\Order # The order entity.
        order_item:
            model: App\AppBundle\Entity\OrderItem # Your order item entity, if you need to override

sylius_product:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.
    classes:
        product:
            model: App\AppBundle\Entity\Product
```

Updating database schema

Remember to update your database schema.

For "**doctrine/orm**" driver run the following command.

```
$ php app/console doctrine:schema:update --force
```

> **Warning:** This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

The Order and OrderItem

Here is a quick reference of what the default models can do for you.

Order basics

Each order has 2 main identifiers, an *ID* and a human-friendly *number*. You can access those by calling `->getId()` and `-getNumber()` respectively. The number is mutable, so you can change it by calling `->setNumber('E001')` on the order instance.

```
<?php

$order->getId();
$order->getNumber();

$order->setNumber('E001');
```

Confirmation status

To check whether the order is confirmed or not, you can use the `isConfirmed()` method, which returns a *true/false* value. To change that status, you can use the confirmation setter, `setConfirmed(false)`. All orders are confirmed by default, unless you enabled the e-mail confirmation feature. Order also can contain a confirmation token, accessible by the appropriate getter and setter.

```
<?php

if ($order->isConfirmed()) {
```

```
    echo 'This one is confirmed, great!';
}
```

Order totals

Note: All money amounts in Sylius are represented as "cents" - integers.

An order has 3 basic totals, which are all persisted together with the order.

The first total is the *items total*, it is calculated as the sum of all item totals.

The second total is the *adjustments total*, you can read more about this in next chapter.

```
<?php

echo $order->getItemsTotal(); // 1900.
echo $order->getAdjustmentsTotal(); // -250.

$order->calculateTotal();
echo $order->getTotal(); // 1650.
```

The main order total is a sum of the previously mentioned values. You can access the order total value using the `->getTotal()` method. Recalculation of totals can happen by calling `->calculateTotal()` method, using the simplest possible math. It will also update the item totals.

Items management

The collection of items (Implementing the `Doctrine\Common\Collections\Collection` interface) can be obtained using the `->getItems()`. To add or remove items, you can simply use the `addItem` and `removeItem` methods.

```
<?php

// $item1 and $item2 are instances of OrderItemInterface.
$order
    ->addItem($item)
    ->removeItem($item2)
;
```

OrderItem basics

An order item model has only the id as identifier, also it has the order to which it belongs, accessible via `->getOrder()` method.

The sellable object can be retrieved and set, using the following setter and getter - `->getProduct()` & `->setVariant(ProductVariantInterface $variant)`.

```
<?php

$item->setVariant($book);
```

Note: In most cases you'll use the **OrderBuilder** service to create your orders.

Just like for the order, the total is available via the same method, but the unit price is accessible using the `->getUnitPrice()` Each item also can calculate its total, using the quantity (`->getQuantity()`) and the unit price.

```
<?php

$item = $itemRepository->createNew();
$item
    ->setVariant($book)
    ->setUnitPrice(2000)
    ->setQuantity(4)
    ->calculateTotal()
;

echo $item->getTotal(); // 8000.
```

An OrderItem can also hold adjustments.

The Adjustments

Adjustments are based on simple but powerful idea inspired by Spree adjustments. They serve as foundation for any tax, shipping and discounts systems.

Adjustment model

Note: To be written.

Creating orders with OrderBuilder

Note: To be written.

Using the services

When using the bundle, you have access to several handy services. You can use them to retrieve and persist orders.

Managers and Repositories

Note: Sylius uses `Doctrine\Common\Persistence` interfaces.

You have access to following services which are used to manage and retrieve resources.

This set of default services is shared across almost all Sylius bundles, but this is just a convention. You're interacting with them like you usually do with own entities in your project.

```
<?php

// ObjectManager which is capable of managing the resources.
// For *doctrine/orm* driver it will be EntityManager.
$this->get('sylius.manager.order');
$this->get('sylius.manager.order_item');
```

```php
$this->get('sylius.manager.adjustment');

// ObjectRepository for the Order resource, it extends the base EntityRepository.
// You can use it like usual entity repository in project.
$this->get('sylius.repository.order');
$this->get('sylius.repository.order_item');
$this->get('sylius.repository.adjustment');

// Those repositories have some handy default methods, for example...
$item = $itemRepository->createNew();
$orderRepository->find(4);
$paginator = $orderRepository->createPaginator(array('confirmed' => false)); // Get Pagerfanta instan
```

Summary

Note: To be written.

Configuration reference

```yaml
sylius_order:
    driver: ~ # The driver used for persistence layer.
    classes:
        sellable:
            model: ~ # The class name of the entity you want to put inside orders.
        order:
            model: ~ # The order model class.
            controller: Sylius\Bundle\OrderBundle\Controller\OrderController
            repository: ~ # You can override the repository class here.
            form: Sylius\Bundle\OrderBundle\Form\Type\OrderType # The form type name to use.
        order_item:
            model: ~ # The order item model class.
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\OrderBundle\Form\Type\OrderItemType # The form type class name to use
        adjustment:
            model: ~ # The adjustment model class.
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\OrderBundle\Form\Type\AdjustmentType
    validation_groups:
        order: [sylius] # Order validation groups.
        order_item: [sylius] # Order item validation groups.
```

phpspec2 examples

```
$ composer install --dev --prefer-dist
$ bin/phpspec run -fpretty --verbose
```

Bug tracking

This bundle uses GitHub issues. If you have found bug, please create an issue.

4.1.6 SyliusAddressingBundle

This bundle integrates the *Addressing* into Symfony2 and Doctrine.

With minimal configuration you can introduce addresses, countries, provinces and zones management into your project. It's fully customizable, but the default setup should be optimal for most use cases.

It also contains zone matching mechanisms, which allow you to match customer addresses to appropriate tax/shipping (or any other) zones. There are several models inside the bundle, *Address*, *Country*, *Province*, *Zone* and *ZoneMember*.

There is also a **ZoneMatcher** service. You'll get familiar with it in later parts of this documentation.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/addressing-bundle:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/addressing-bundle:*
```

Adding required bundles to the kernel

You need to enable the bundle inside the kernel.

If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies to kernel. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),

        new Sylius\Bundle\AddressingBundle\SyliusAddressingBundle(),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    );
}
```

Note: Please register the bundle before *DoctrineBundle*. This is important as we use listeners which have to be processed first.

Container configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_addressing:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.
```

And configure doctrine extensions which are used in assortment bundle:

```
stof_doctrine_extensions:
    orm:
        default:
            timestampable: true
```

Routing configuration

Import the routing configuration by adding the following to your `app/config/routing.yml`.

```
sylius_addressing:
    resource: @SyliusAddressingBundle/Resources/config/routing.yml
```

Updating database schema

Run the following command.

```
$ php app/console doctrine:schema:update --force
```

> **Warning:** This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

Templates

This bundle provides some default bootstrap templates.

Note: You can check Sylius application to see how to integrate it in your application.

ZoneMatcher

This bundle exposes the **ZoneMatcher** as `sylius.zone_matcher` service.

```
<?php

$zoneMatcher = $this->get('sylius.zone_matcher');

$zone = $zoneMatcher->match($user->getBillingAddress);
```

Forms

The bundle ships with a set of useful form types for all models. You can use the defaults or *override them* with your own types.

Address form

The address form type is named `sylius_address` and you can create it whenever you need, using the form factory.

```
<?php

// src/Acme/ShopBundle/Controller/AddressController.php

namespace Acme\ShopBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class DemoController extends Controller
{
    public function fooAction(Request $request)
    {
        $form = $this->get('form.factory')->create('sylius_address');
    }
}
```

You can also embed it into another form.

```
<?php

// src/Acme/ShopBundle/Form/Type/OrderType.php

namespace Acme\ShopBundle\Form\Type;

use Sylius\Bundle\OrderBundle\Form\Type\OrderType as BaseOrderType;
use Symfony\Component\Form\FormBuilderInterface;

class OrderType extends BaseOrderType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        parent::buildForm($builder, $options);

        $builder
            ->add('billingAddress', 'sylius_address')
            ->add('shippingAddress', 'sylius_address')
        ;
    }
}
```

Configuration Reference

```
sylius_addressing:
    driver: ~ # The driver used for persistence layer.
    classes:
        address:
            model: Sylius\Bundle\AddressingBundle\Model\Address
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AddressingBundle\Form\Type\AddressType
        country:
            model: Sylius\Bundle\AddressingBundle\Model\Country
```

```
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AddressingBundle\Form\Type\CountryType
        province:
            model: Sylius\Bundle\AddressingBundle\Model\Province
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AddressingBundle\Form\Type\ProvinceType
        zone:
            model: Sylius\Bundle\AddressingBundle\Model\Zone
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AddressingBundle\Form\Type\ZoneType
        zone_member:
            model: Sylius\Bundle\AddressingBundle\Model\ZoneMember
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AddressingBundle\Form\Type\ZoneMemberType
        zone_member_country:
            model: Sylius\Bundle\AddressingBundle\Model\ZoneMemberCountry
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AddressingBundle\Form\Type\ZoneMemberCountryType
        zone_member_province:
            model: Sylius\Bundle\AddressingBundle\Model\ZoneMemberProvince
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AddressingBundle\Form\Type\ZoneMemberProvinceType
        zone_member_zone:
            model: Sylius\Bundle\AddressingBundle\Model\ZoneMemberZone
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\AddressingBundle\Form\Type\ZoneMemberZoneType
    validation_groups:
        address: [sylius]
        country: [sylius]
        province: [sylius]
        zone: [sylius]
        zone_member: [sylius]
```

4.1.7 SyliusInventoryBundle

Flexible inventory management for Symfony2 applications.

With minimal configuration you can implement inventory tracking in your project.

It's fully customizable, but the default setup should be optimal for most use cases.

There is *StockableInterface* and *InventoryUnit* model inside the bundle.

There are services **AvailabilityChecker**, **InventoryOperator** and **InventoryChangeListener**.

You'll get familiar with them in later parts of this documentation.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require "sylius/inventory-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/inventory-bundle"
```

Adding required bundles to the kernel

First, you need to enable the bundle inside the kernel. If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies to the kernel. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new FOS\RestBundle\FOSRestBundle(),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),
        new Sylius\Bundle\InventoryBundle\SyliusInventoryBundle(),
    );
}
```

Creating your entities

Let's assume we want to implement a book store application and track the books inventory.

You have to create a *Book* and an *InventoryUnit* entity, living inside your application code. We think that **keeping the app-specific bundle structure simple** is a good practice, so let's assume you have your `AppBundle` registered under `App\Bundle\AppBundle` namespace.

We will create *Book* entity.

```
<?php

// src/App/AppBundle/Entity/Book.php
namespace App\AppBundle\Entity;

use Sylius\Bundle\InventoryBundle\Model\StockableInterface;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="app_book")
 */
class Book implements StockableInterface
```

```
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string")
     */
    protected $isbn;

    /**
     * @ORM\Column(type="string")
     */
    protected $title;

    /**
     * @ORM\Column(type="integer")
     */
    protected $onHand;

    /**
     * @ORM\Column(type="boolean")
     */
    protected $availableOnDemand;

    public function __construct()
    {
        $this->onHand = 1;
        $this->availableOnDemand = true;
    }

    public function getId()
    {
        return $this->id;
    }

    public function getIsbn()
    {
        return $this->isbn;
    }

    public function setIsbn($isbn)
    {
        $this->isbn = $isbn;
    }

    public function getSku()
    {
        return $this->getIsbn();
    }

    public function getTitle()
    {
        return $this->title;
    }
```

```
    public function setTitle($title)
    {
        $this->title = $title;
    }

    public function getInventoryName()
    {
        return $this->getTitle();
    }

    public function isInStock()
    {
        return 0 < $this->onHand;
    }

    public function isAvailableOnDemand()
    {
        return $this->availableOnDemand;
    }

    public function setAvailableOnDemand($availableOnDemand)
    {
        $this->availableOnDemand = (Boolean) $availableOnDemand;
    }

    public function getOnHand()
    {
        return $this->onHand;
    }

    public function setOnHand($onHand)
    {
        $this->onHand = $onHand;
    }
}
```

Note: This example shows the full power of *StockableInterface*. The bundle also provides an *Stockable* entity which implements *StockableInterface* for you. By extending the *Stockable* entity, the example above can be dramatically simplified.

In order to track the books inventory our *Book* entity must implement *StockableInterface*. Note that we added `->getSku()` method which is alias to `->getIsbn()`, this is the power of the interface, we now have full control over the entity mapping. In the same way `->getInventoryName()` exposes the book title as the displayed name for our stockable entity.

The next step requires the creating of the *InventoryUnit* entity, let's do this now.

```
<?php

// src/App/AppBundle/Entity/InventoryUnit.php
namespace App\AppBundle\Entity;

use Sylius\Bundle\InventoryBundle\Entity\InventoryUnit as BaseInventoryUnit;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
```

```
 * @ORM\Table(name="app_inventory_unit")
 */
class InventoryUnit extends BaseInventoryUnit
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;
}
```

Note that we are using base entity from Sylius bundle, which means inheriting some functionality inventory bundle provides. *InventoryUnit* holds the reference to stockable object, which is *Book* in our case. So, if we use the *InventoryOperator* to create inventory units, they will reference the given book entity.

Container configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_inventory:
    driver: doctrine/orm
    backorders: true
    classes:
        unit:
            model: App\AppBundle\Entity\InventoryUnit
        stockable:
            model: App\AppBundle\Entity\Book
```

Routing configuration

Import the routing configuration by adding the following to your *app/config/routing.yml*‘.

```
sylius_inventory:
    resource: @SyliusInventoryBundle/Resources/config/routing.yml
```

Updating database schema

Remember to update your database schema.

For “**doctrine/orm**” driver run the following command.

```
$ php app/console doctrine:schema:update --force
```

> **Warning:** This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

Templates

The bundle provides some default bootstrap templates.

Note: You can check our Sandbox app to see how to integrate it in your application.

Models

Here is a quick reference for the default models.

InventoryUnit

Each unit holds a reference to a stockable object and its state, which can be **sold** or **backordered**. It also provides some handy shortcut methods like *isSold*, *isBackordered* and *getSku*.

Stockable

In order to be able to track stock levels in your application, you must implement *StockableInterface* or use the *Stockable* model. It uses the SKU to identify stockable, need to provide display name and to check if stockable is available on demand. It can get/set current stock level with *getOnHand* and *setOnHand* methods.

Using the services

When using the bundle, you have access to several handy services.

AvailabilityChecker

The name speaks for itself, this service checks availability for given stockable object. It takes backorders setting into account, so if backorders are enabled, stockable will always be available. Backorders can be enabled per stockable if it is available on demand. If none of this is the case, it means that backorders are not enabled for the given stockable and *AvailabilityChecker* will rely on the current stock level.

There are two methods for checking availability. `->isStockAvailable()` just checks whether stockable object is available in stock and doesn't care about quantity. `->isStockSufficient()` checks if there is enough units in the stock for given quantity.

InventoryOperator

Inventory operator is the heart of this bundle. It can be used to manage stock levels and inventory units. It can also fill backorders for the given stockable, this is a very powerful feature in combination with *InventoryChangeListener*. Creating/destroying inventory units with a given state is also the operators job.

InventoryChangeListener

It simply triggers `InventoryOperatorInterface::fillBackorders()`. This can be extended by implementing *InventoryChangeListenerInterface*. Events can be configured like explained later on the documentation *Summary*.

Twig Extension

There are two handy twig functions bundled in: *sylius_inventory_is_available* and *sylius_inventory_is_sufficient*.

They are simple proxies to the availability checker, and can be used to show if the stockable object is available/sufficient.

Here is a simple example, note that *product* variable has to be an instance of *StockableInterface*.

```
{% if not sylius_inventory_is_available(product) %}
    <span class="label label-important">out of stock</span>
{% endif %}
```

Summary

Configuration reference

```
sylius_inventory:
    driver: ~ # The driver used for persistence layer.
    engine: twig # Templating engine to use by default.
    backorders: true # Enable/disable backorders.
    events: ~ # Array of events for InventoryChangeListener
    classes:
        unit:
            model: ~ # The inventory unit model class.
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~ # You can override the repository class here.
        stockable:
            model: ~ # The stockable model class.
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~ # You can override the repository class here.
    checker: sylius_inventory.checker.default # The availability checker service id.
    operator: sylius_inventory.operator.default # The inventory operator service id.
```

phpspec2 examples

```
$ composer install --dev --prefer-dist
$ bin/phpspec run -f pretty
```

Working examples

If you want to see working implementation, try out the Sylius sandbox application.

Bug tracking

This bundle uses GitHub issues. If you have found bug, please create an issue.

4.1.8 SyliusShippingBundle

SyliusShippingBundle is the shipment management component for Symfony2 e-commerce applications.

If you need to manage shipments, shipping methods and deal with complex cost calculation, this bundle can help you a lot!

Your products or whatever you need to deliver, can be categorized under unlimited set of categories. You can display appropriate shipping methods available to the user, based on object category, weight, dimensions and anything you can imagine.

Flexible shipping cost calculation system allows you to create your own calculator services.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require "sylius/shipping-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/shipping-bundle"
```

Adding required bundles to the kernel

You need to enable the bundle inside the kernel.

If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),
        new Sylius\Bundle\ShippingBundle\SyliusShippingBundle(),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    );
}
```

Note: Please register the bundle before *DoctrineBundle*. This is important as we use listeners which have to be processed first.

Container configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_shipping:
    driver: doctrine/orm # Configure the Doctrine ORM driver used in documentation.
```

Configure doctrine extensions which are used by this bundle.

```
stof_doctrine_extensions:
    orm:
        default:
            timestampable: true
```

Routing configuration

Add the following to your `app/config/routing.yml`.

```
sylius_shipping:
    resource: @SyliusShipping/Resources/config/routing.yml
```

Updating database schema

Run the following command.

```
$ php app/console doctrine:schema:update --force
```

> **Warning:** This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

The ShippableInterface

In order to handle your merchandise through the Sylius shipping engine, your models need to implement **ShippableInterface**.

Implementing the interface

Let's assume that you have a **Book** entity in your application.

First step is to implement the simple interface, which contains few simple methods.

```
namespace Acme\Bundle\ShopBundle\Entity;

use Sylius\Bundle\ShippingBundle\Model\ShippableInterface;
use Sylius\Bundle\ShippingBundle\Model\ShippingCategoryInterface;

class Book
{
    private $shippingCategory;

    public function getShippingCategory()
    {
        return $this->shippingCategory;
    }

    public function setShippingCategory(ShippingCategoryInterface $shippingCategory) // This method
```

```
    {
        $this->shippingCategory = $shippingCategory;

        return $this;
    }

    public function getShippingWeight()
    {
        // return integer representing the object weight.
    }

    public function getShippingWidth()
    {
        // return integer representing the book width.
    }

    public function getShippingHeight()
    {
        // return integer representing the book height.
    }

    public function getShippingDepth()
    {
        // return integer representing the book depth.
    }
}
```

Second and last task is to define the relation inside `Resources/config/doctrine/Book.orm.xml` of your bundle.

```
<?xml version="1.0" encoding="UTF-8"?>

<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\ShopBundle\Entity\Book" table="acme_book">
        <!-- your mappings... -->

        <many-to-one field="shippingCategory" target-entity="Sylius\Bundle\ShippingBundle\Model\Shipp
            <join-column name="shipping_category_id" referenced-column-name="id" nullable="false" />
        </many-to-one>
    </entity>

</doctrine-mapping>
```

Done! Now your **Book** model can be used in Sylius shippingation engine.

Forms

If you want to add a shipping category selection field to your model form, simply use the `sylius_shipping_category_choice` type.

```
namespace Acme\ShopBundle\Form\Type;

use Symfony\Component\Form\FormBuilderInterface;
```

```php
use Symfony\Component\Form\AbstractType;

class BookType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title', 'text')
            ->add('shippingCategory', 'sylius_shipping_category_choice')
        ;
    }
}
```

The ShippingSubjectInterface

The find available shipping methods or calculate shipping cost you need to use object implementing `ShippingSubjectInterface`.

The default **Shipment** model is already implementing `ShippingSubjectInterface`.

Interface methods

- The `getShippingMethod` returns a `ShippingMethodInterface` instance, representing the method.
- The `getShippingItemCount` provides you with the count of items to ship.
- The `getShippingItemTotal` returns the total value of shipment, if applicable. The default **Shipment** model returns 0.
- The `getShippingWeight` returns the total shipment weight.
- The `getShippables` returns a collection of unique `ShippableInterface` instances.

The Shipping Categories

Every shippable object needs to have a shipping category assigned. The **ShippingCategory** model is extremely simple and described below.

Attribute	Description
id	Unique id of the shipping category
name	Name of the shipping category
description	Human friendly description of the classification
createdAt	Date when the category was created
updatedAt	Date of the last shipping category update

Calculating shipping cost

Calculating shipping cost is as simple as using the `sylius.shipping_calculator` service and calling `calculate` method on `ShippingSubjectInterface`.

Let's calculate the cost of existing shipment.

```php
public function myAction()
{
    $calculator = $this->get('sylius.shipping_calculator');
```

```
    $shipment = $this->get('sylius.repository.shipment')->find(5);

    echo $calculator->calculate($shipment); // Returns price in cents. (integer)
}
```

What has happened?

- The delegating calculator gets the **ShippingMethod** from the **ShippingSubjectInterface** (Shipment).
- Appropriate **Calculator** instance is loaded, based on the **ShippingMethod.calculator** parameter.
- The `calculate(ShippingSubjectInterface, array $configuration)` is called, where configuration is taken from **ShippingMethod.configuration** attribute.

Default calculators

Default calculators can be sufficient solution for many use cases.

Flat rate

The `flat_rate` calculator, charges concrete amount per shipment.

Per item rate

The `per_item_rate` calculator, charges concrete amount per shipment item.

Flexible rate

The `flexible_rate` calculator, charges one price for the first item, and another price for every other item.

Weight rate

The `weight_rate` calculator, charges one price for certain weight of shipment. So if the shipment weights 5 kg, and calculator is configured to charge $4 per kg, the final price is $20.

More calculators

Depending on community contributions and Sylius resources, more default calculators can be implemented, for example `weight_range_rate`.

Custom calculators

Sylius ships with several default calculators, but you can easily register your own.

Simple calculators

All shipping cost calculators implement `CalculatorInterface`. In our example we'll create a calculator which calls an external API to obtain the shipping cost.

```
<?php

// src/Acme/ShopBundle\Shipping/DHLCalculator.php

namespace Acme\ShopBundle\Shipping;

use Acme\ShopBundle\Shipping\DHLService;
use Sylius\Bundle\ShippingBundle\Calculator\Calculator;
use Sylius\Bundle\ShippingBundle\Model\ShippingSubjectInterface;

class DHLCalculator extends Calculator
{
    private $dhlService;

    public function __construct(DHLService $dhlService)
    {
        $this->dhlService = $dhlService;
    }

    public function calculate(ShippingSubjectInterface $subject, array $configuration)
    {
        return $this->dhlService->getShippingCostForWeight($subject->getShippingWeight());
    }
}
```

Now, you need to register your new service in container and tag it with `sylius.shipping_calculator`.

```
<?xml version="1.0" encoding="UTF-8"?>

<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/services-1.0.xsd">

    <services>
        <service id="acme.shipping_calculator.dhl" class="Acme\ShopBundle\Shipping\DHLCalculator">
            <argument type="service" id="acme.dhl_service" />
            <tag name="sylius.shipping_calculator" calculator="dhl" label="DHL" />
        </service>
    </services>
</container>
```

That would be all. This new option ("DHL") will appear on the **ShippingMethod** creation form, in the "calculator" field.

Configurable calculators

You can also create configurable calculators, meaning that you can have several **ShippingMethod**'s using same type of calculator, with different settings.

Let's modify the **DHLCalculator**, so that it charges 0 if shipping more than X items. First step is to define the configuration options, using the Symfony **OptionsResolver** component.

```php
<?php

// src/Acme/ShopBundle\Shipping/DHLCalculator.php

namespace Acme\ShopBundle\Shipping;

use Acme\ShopBundle\Shipping\DHLService;
use Sylius\Bundle\ShippingBundle\Calculator\Calculator;
use Sylius\Bundle\ShippingBundle\Model\ShippingSubjectInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class DHLCalculator extends Calculator
{
    private $dhlService;

    public function __construct(DHLService $dhlService)
    {
        $this->dhlService = $dhlService;
    }

    public function calculate(ShippingSubjectInterface $subject, array $configuration)
    {
        return $this->dhlService->getShippingCostForWeight($subject->getShippingWeight());
    }

    public function setConfiguration(OptionsResolverInterface $resolver)
    {
        $resolver
            ->setDefaults(array(
                'limit' => 10
            ))
            ->setAllowedTypes(array(
                'limit' => array('integer'),
            ))
        ;
    }
}
```

Done, we've set the default item limit to 10. Now we have to create a form type which will be displayed if our calculator is selected.

```php
<?php

// src/Acme/ShopBundle/Form/Type/Shipping/DHLConfigurationType.php

namespace Acme\ShopBundle\Form\Type\Shipping;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\Type;

class DHLConfigurationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
```

```
            ->add('limit', 'integer', array(
                'label' => 'Free shipping above total items',
                'constraints' => array(
                    new NotBlank(),
                    new Type(array('type' => 'integer')),
                )
            ))
        ;
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver
            ->setDefaults(array(
                'data_class' => null
            ))
        ;
    }

    public function getName()
    {
        return 'acme_shipping_calculator_dhl';
    }
}
```

We also need to register the form type and the calculator in the container.

```
<?xml version="1.0" encoding="UTF-8"?>

<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/services-1.0.xsd">

    <services>
        <service id="acme.shipping_calculator.dhl" class="Acme\ShopBundle\Shipping\DHLCalculator">
            <argument type="service" id="acme.dhl_service" />
            <tag name="sylius.shipping_calculator" calculator="dhl" label="DHL" />
        </service>
        <service id="acme.form.type.shipping_calculator.dhl" class="Acme\ShopBundle\Form\Type\Shippi
            <tag name="form.type" alias="acme_shipping_calculator_dhl" />
        </service>
    </services>
</container>
```

Finally, configure the calculator to use the form, by implementing simple `getConfigurationFormType` method.

```
<?php

// src/Acme/ShopBundle\Shipping/DHLCalculator.php

namespace Acme\ShopBundle\Shipping;

use Acme\ShopBundle\Shipping\DHLService;
use Sylius\Bundle\ShippingBundle\Calculator\Calculator;
use Sylius\Bundle\ShippingBundle\Model\ShippingSubjectInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class DHLCalculator extends Calculator
```

```
{
    private $dhlService;

    public function __construct(DHLService $dhlService)
    {
        $this->dhlService = $dhlService;
    }

    public function calculate(ShippingSubjectInterface $subject, array $configuration)
    {
        return $this->dhlService->getShippingCostForWeight($subject->getShippingWeight());
    }

    public function setConfiguration(OptionsResolverInterface $resolver)
    {
        $resolver
            ->setDefaults(array(
                'limit' => 10
            ))
            ->setAllowedTypes(array(
                'limit' => array('integer'),
            ))
        ;
    }

    public function getConfigurationFormType()
    {
        return 'acme_shipping_calculator_dhl';
    }
}
```

Perfect, now we're able to use the configuration inside the `calculate` method.

```
<?php

// src/Acme/ShopBundle\Shipping/DHLCalculator.php

namespace Acme\ShopBundle\Shipping;

use Acme\ShopBundle\Shipping\DHLService;
use Sylius\Bundle\ShippingBundle\Calculator\Calculator;
use Sylius\Bundle\ShippingBundle\Model\ShippingSubjectInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class DHLCalculator extends Calculator
{
    private $dhlService;

    public function __construct(DHLService $dhlService)
    {
        $this->dhlService = $dhlService;
    }

    public function calculate(ShippingSubjectInterface $subject, array $configuration)
    {
        if ($subject->getShippingItemCount() > $configuration['limit']) {
            return 0;
        }
```

```
        return $this->dhlService->getShippingCostForWeight($subject->getShippingWeight());
    }

    public function setConfiguration(OptionsResolverInterface $resolver)
    {
        $resolver
            ->setDefaults(array(
                'limit' => 10
            ))
            ->setAllowedTypes(array(
                'limit' => array('integer'),
            ))
        ;
    }

    public function getConfigurationFormType()
    {
        return 'acme_shipping_calculator_dhl';
    }
}
```

Your new configurable calculator is ready to use. When you select the "DHL" calculator in **ShippingMethod** form, configuration fields will appear automatically.

Shipping method requirements

Sylius has a very flexible system for displaying only the right shipping methods to the user.

Shipping categories

Every **ShippableInterface** can hold a reference to **ShippingCategory**. The **ShippingSubjectInterface** (or **ShipmentInterface**) returns a collection of shippables.

ShippingMethod has an optional shipping category setting as well as **categoryRequirement** which has 3 options. If this setting is set to null, categories system is ignored.

"Match any" requirement With this requirement, the shipping method will support any shipment (or shipping subject) which contains at least one shippable with the same category.

"Match all" requirement All shippables have to reference the same category as the **ShippingMethod**.

"Match none" requirement None of the shippables can have the same shipping category.

Shipping rules

The categories system is sufficient for basic use cases, for more advanced requirements, Sylius has the shipping rules system.

Every **ShippingMethod** can have a collection of **ShippingRule** instances. Each shipping rule, has a **checker** and configuration assigned. The **ShippingRule.checker** attribute holds the alias name of appropriate service implementing **RuleCheckerInterface**. Just like with cost calculators, Sylius ships with default checkers, but you can easily implement your own.

The **ShippingMethodEligibilityChecker** service can check if given subject, satisfies the category and shipping rules.

Default rule checkers

Sylius ships with several shipping rule checker types, so you can easily decide whether the shipping method is applicable to given shipment.

Item count

The `item_count` checker, accepts the subject only if the items count fits into **min** and **max** range.

You can configure a method, which will be available only if the shipment contains more than 5 items, but less than 20.

Item total

The `item_total` checker, is testing if the shipping subject total value is more than configured minimum, or eventually, less than maximum.

Weight

The `weight` checker, allows to ship the shipment using the particular method, only if the shipment weight falls into the configure **min** and **max** range.

More checkers

Depending on community contributions and Sylius resources, more default checkers can be implemented.

Custom rule checkers

Implementing a custom rule checker is really simple, just like calculators, you can use simple services, or more complex - configurable checkers.

Simple checkers

Note: To be written.

Configurable calculators

Note: To be written.

Configuration reference

```
sylius_shipping:
    driver: ~ # The driver used for persistence layer.
    classes:
        shipment:
            model: Sylius\Bundle\ShippingBundle\Model\Shipment
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\ShippingBundle\Form\Type\ShipmentType
        shipment_item:
            model: Sylius\Bundle\ShippingBundle\Model\ShipmentItem
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\ShippingBundle\Form\Type\ShipmentItemType
        shipping_method:
            model: Sylius\Bundle\ShippingBundle\Model\ShippingMethod
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\ShippingBundle\Form\Type\ShippingMethodType
        shipping_method_rule:
            model: Sylius\Bundle\ShippingBundle\Model\ShippingMethodRule
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\ShippingBundle\Form\Type\ShippingMethodRuleType
    validation_groups:
        shipment: [sylius]
        shipment_item: [sylius]
        shipping_method: [sylius]
        shipping_method_rule: [sylius]
```

4.1.9 SyliusTaxationBundle

Calculating and applying taxes is a common task for most of ecommerce applications. **SyliusTaxationBundle** is a reusable taxation component for Symfony2. You can integrate it into your existing application and enable the tax calculation logic for any model implementing the `TaxableInterface`.

It supports different tax categories and customizable tax calculators - you're able to easily implement your own calculator services. The default implementation handles tax included in and excluded from the price.

As with any Sylius bundle, you can override all the models, controllers, repositories, forms and services.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require "sylius/taxation-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/taxation-bundle"
```

Adding required bundles to the kernel

First, you need to enable the bundle inside the kernel.

If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies to kernel. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),
        new Sylius\Bundle\TaxationBundle\SyliusTaxationBundle(),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    );
}
```

Note: Please register the bundle before *DoctrineBundle*. This is important as we use listeners which have to be processed first.

Container configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_taxation:
    driver: doctrine/orm # Configure the Doctrine ORM driver used in documentation.
```

And configure doctrine extensions which are used by this bundle:

```
stof_doctrine_extensions:
    orm:
        default:
            timestampable: true
```

Routing configuration

Add the following to your `app/config/routing.yml`.

```
sylius_taxation:
    resource: @SyliusTaxationBundle/Resources/config/routing.yml
```

Updating database schema

Run the following command.

```
$ php app/console doctrine:schema:update --force
```

Warning: This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

The TaxableInterface

In order to calculate the taxes for a model in your application, it needs to implement the `TaxableInterface` It is a very simple interface, with only one method - the `getTaxCategory()`, as every taxable has to belong to a specific tax category.

Implementing the interface

Let's assume that you have a **Server** entity in your application. Every server has it's price and other parameters, but you would like to calculate the tax included in price Every server has it's price and other parameters, but a would like to calculate the tax included in price. You could calculate the math in a simple method, but it's not enough when you have to handle multiple tax rates, categories and zones.

First step is to implement the simple interface.

```
namespace Acme\Bundle\ShopBundle\Entity;

use Sylius\Bundle\TaxationBundle\Model\TaxCategoryInterface;
use Sylius\Bundle\TaxationBundle\Model\TaxableInterface;

class Server
{
    private $taxCategory;

    public function getTaxCategory()
    {
        return $this->taxCategory;
    }

    public function setTaxCategory(TaxCategoryInterface $taxCategory) // This method is not required
    {
        $this->taxCategory = $taxCategory;

        return $this;
    }
}
```

Second and last task is to define the relation inside `Resources/config/doctrine/Server.orm.xml` of your bundle.

```
<?xml version="1.0" encoding="UTF-8"?>

<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\ShopBundle\Entity\Server" table="acme_server">
        <!-- your mappings... -->
```

```
        <many-to-one field="taxCategory" target-entity="Sylius\Bundle\TaxationBundle\Model\TaxCategor
            <join-column name="tax_category_id" referenced-column-name="id" nullable="false" />
        </many-to-one>
    </entity>

</doctrine-mapping>
```

Done! Now your **Server** model can be used in Sylius taxation engine.

Forms

If you want to add a tax category selection field to your model form, simply use the `sylius_tax_category_choice` type.

```
namespace Acme\ShopBundle\Form\Type;

use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\AbstractType;

class ServerType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name', 'text')
            ->add('taxCategory', 'sylius_tax_category_choice')
        ;
    }
}
```

The Tax Rates

Tax rate model holds the configuration for particular tax category.

Attribute	Description
id	Unique id of the tax rate
name	Name of the rate
amount	Amount as float (for example 0,23)
includedInPrice	Is the tax included in price?
calculator	Type of calculator
createdAt	Date when the rate was created
updatedAt	Date of the last tax rate update

Configuring taxation

To start calculating taxes, we need to configure the system. In most cases, the configuration process is done via web interface, but you can also do it programatically.

Creating the tax categories

First step is to create a new tax category.

```php
<?php

public function configureAction()
{
    $repository = $this->container->get('sylius.repository.tax_category');
    $manager = $this->container->get('sylius.manager.tax_category');

    $clothing = $repository
        ->createNew()
        ->setName('Clothing')
        ->setDescription('T-Shirts and this kind of stuff.')
    ;
    $food = $repository
        ->createNew()
        ->setName('Food')
        ->setDescription('Yummy!')
    ;

    $manager->persist($clothing);
    $manager->persist($food);

    $manager->flush();
}
```

Categorizing the taxables

Second thing to do is to classify the taxables, in our example we'll get two products and assign the proper categories to them.

```php
<?php

public function configureAction()
{
    $tshirt = // ...
    $banana = // ... Some logic behind loading entities.

    $repository = $this->container->get('sylius.repository.tax_category');

    $clothing = $repository->findOneBy(array('name' => 'Clothing'));
    $food = $repository->findOneBy(array('name' => 'Food'));

    $tshirt->setTaxCategory($clothing);
    $food->setTaxCategory($food);

    // Save the product entities.
}
```

Configuring the tax rates

Finally, you have to create appropriate tax rates for each of categories.

```php
<?php

public function configureAction()
{
```

```
    $taxCategoryRepository = $this->container->get('sylius.repository.tax_category');

    $clothing = $taxCategoryRepository->findOneBy(array('name' => 'Clothing'));
    $food = $taxCategoryRepository->findOneBy(array('name' => 'Food'));

    $repository = $this->container->get('sylius.repository.tax_rate');
    $manager = $this->container->get('sylius.repository.tax_rate');

    $clothingTax = $repository
        ->createNew()
        ->setName('Clothing Tax')
        ->setAmount(0,08)
    ;
    $foodTax = $repository
        ->createNew()
        ->setName('Food')
        ->setAmount(0,12)
    ;

    $manager->persist($clothingTax);
    $manager->persist($foodTax);

    $manager->flush();
}
```

Done! See the *"Calculating Taxes" chapter* to see how to apply these rates.

Calculating taxes

Warning: When using the CoreBundle (i.e: full stack Sylius framework), the taxes are already calculated at each cart change. It is implemented by the `TaxationProcessor` class, which is called by the *OrderTaxationListener*'.

In order to calculate tax amount for given taxable, we need to find out the applicable tax rate. The tax rate resolver service is available under `sylius.tax_rate_resolver` id, while the delegating tax calculator is accessible via `sylius.tax_calculator` name.

Resolving rate and using calculator

```
<?php

namespace Acme\ShopBundle\Taxation

use Acme\ShopBundle\Entity\Order;
use Sylius\Bundle\TaxationBundle\Calculator\CalculatorInterface;
use Sylius\Bundle\TaxationBundle\Resolver\TaxRateResolverInterface;

class TaxApplicator
{
    private $calculator;
    private $taxRateResolver;

    public function __construct(
        CalculatorInterface $calculator,
```

```
        TaxRateResolverInterface $taxRateResolver,
    )
    {
        $this->calculator = $calculator;
        $this->taxRateResolver = $taxRateResolver;

    }

    public function applyTaxes(Order $order)
    {
        $tax = 0;

        foreach ($order->getItems() as $item) {
            $taxable = $item->getProduct();
            $rate = $this->taxRateResolver->resolve($taxable);

            if (null === $rate) {
                continue; // Skip this item, there is no matching tax rate.
            }

            $tax += $this->calculator->calculate($item->getTotal(), $rate);
        }

        $order->setTaxTotal($tax); // Set the calculated taxes.
    }
}
```

Using custom tax calculators

Every **TaxRate** model holds a *calculator* variable with the name of the tax calculation service, used to compute the tax amount. While the default calculator should fit for most common use cases, you're free to define your own implementation.

Creating the calculator

All calculators implement the **TaxCalculatorInterface**. First, you need to create a new class.

```
namespace Acme\Bundle\ShopBundle\TaxCalculator;

use Sylius\Bundle\TaxationBundle\Calculator\TaxCalculatorInterface;
use Sylius\Bundle\TaxationBundle\Model\TaxRateInterface;

class FeeCalculator implements TaxCalculatorInterface
{
    public function calculate($amount, TaxRate $rate)
    {
        return $amount * ($rate->getAmount() + 0,15 * 0,30);
    }
}
```

Configuration reference

```
sylius_taxation:
    driver: ~ # The driver used for persistence layer.
```

```
    classes:
        tax_category:
            model: Sylius\Bundle\TaxationBundle\Model\TaxCategory
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\TaxationBundle\Form\Type\TaxCategoryType
        tax_rate:
            model: Sylius\Bundle\TaxationBundle\Model\TaxRate
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\TaxationBundle\Form\Type\TaxRateType
    validation_groups:
        tax_category: [sylius]
        tax_rate: [sylius]
```

4.1.10 SyliusPromotionsBundle

Promotions system for Symfony2 applications.

With minimal configuration you can introduce promotions and coupons into your project. The following types of promotions are available and **totally mixable**:

- percentage discounts
- fixed amount discounts
- promotions limited by time
- promotions limited by a maximum number of usages
- promotions based on coupons

This means you can for instance create the following promotions :

- 20$ discount for New Year orders having more than 3 items
- 8% discount for Christmas orders over 100 EUR
- first 3 orders have 100% discount
- 5% discount this week with the coupon code *WEEK5*
- 40€ discount with the code you have received by mail

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your `composer.json` and download the package.

If you have Composer installed globally.

```
$ composer require "sylius/promotions-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/promotions-bundle"
```

Adding required bundles to the kernel

You need to enable the bundle inside the kernel.

If you're not using any other Sylius bundles, you will also need to add `SyliusResourceBundle` and its dependencies to kernel. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),
        new Sylius\Bundle\PromotionsBundle\SyliusPromotionsBundle(),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    );
}
```

Note: Please register the bundle before *DoctrineBundle*. This is important as we use listeners which have to be processed first.

Container configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_promotions:
    driver: doctrine/orm # Configure the doctrine orm driver used in the documentation.
```

And configure doctrine extensions which are used by the bundle.

```
stof_doctrine_extensions:
    orm:
        default:
            timestampable: true
```

Routing configuration

Add the following to your `app/config/routing.yml`.

```
sylius_promotion:
    resource: @SyliusPromotionsBundle/Resources/config/routing.yml
```

Updating database schema

Run the following command.

```
$ php app/console doctrine:schema:update --force
```

Warning: This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

Congratulations! The bundle is now installed and ready to use.

Models

All the models of this bundle are defined in `Sylius\Bundle\PromotionsBundle\Model`.

Rule

A `Rule` is used to check if your order is eligible to the promotion. A promotion can have none, one or several rules. `SyliusPromotionsBundle` comes with 2 types of rules :

- item count rule : the number of items of the order is checked
- item total rule : the amount of the order is checked

A rule is configured via the `configuration` attribute which is an array serialized into database. For item count rules, you have to configure the `count` key, whereas the `amount` key is used for item total rules. For both types of rules, you can configure the `equal` key to `false` or `true` depending is you want the rule to be strict or not. For instance, for an item count rule configured with `equal` to `false` and `count` to 4, orders with only **more** than 4 items will be eligible.

Action

An `Action` defines the the nature of the discount. Common actions are :

- percentage discount
- fixed amount discount

An action is configured via the `configuration` attribute which is an array serialized into database. For percentage discount actions, you have to configure the `percentage` key, whereas the `amount` key is used for fixed discount rules.

Coupon

A `Coupon` is a ticket having a `code` that can be exchanged for a financial discount. A promotion can have none, one or several coupons.

A coupon is considered as valid if the method `isValid()` returns `true`. This method checks the number of times this coupon can be used (attribute `usageLimit`), the number of times this has already been used (attribute `used`) and the coupon expiration date (attribute `expiresAt`). If `usageLimit` is not set, the coupon will be usable an unlimited times.

PromotionSubjectInterface

A `PromotionSubjectInterface` is the object you want to apply the promotion on. For instance, in Sylius Standard, a `Sylius\Bundle\CoreBundle\Model\Order` can be subject to promotions.

By implementing `PromotionSubjectInterface`, your object will have to define the following methods : - `getPromotionSubjectItemTotal()` should return the amount of your order - `getPromotionSubjectItemCount()` should return the number of items of your order - `getPromotionCoupon()` should return the coupon linked to your order. If you do not want to use coupon, simply return `null`.

Promotion

The `Promotion` is the main model of this bundle. A promotion has a `name`, a `description` and :

- can have none, one or several rules
- should have at least one action to be effective
- can be based on coupons
- can have a limited number of usages by using the attributes `usageLimit` and `used`. When `used` reaches `usageLimit` the promotion is no longer valid. If `usageLimit` is not set, the promotion will be usable an unlimited times.
- can be limited by time by using the attributes `startsAt` and `endsAt`

How rules are checked ?

Everything related to this subject is located in `Sylius\Bundle\PromotionsBundle\Checker`.

Rule checkers

New rules can be created by implementing `RuleCheckerInterface`. This interface provides the method `isEligible` which aims to determine if the promotion subject respects the current rule or not.

I told you before that `SyliusPromotionsBundle` ships with 2 types of rules : item count rule and item total rule.

Item count rule is defined via the service `sylius.promotion_rule_checker.item_count` which uses the class `ItemCountRuleChecker`. The method `isEligible` checks here if the promotion subject has the minimum number of items (method `getPromotionSubjectItemCount()` of `PromotionSubjectInterface`) required by the rule.

Item total rule is defined via the service `sylius.promotion_rule_checker.item_total` which uses the class `ItemTotalRuleChecker`. The method `isEligible` checks here if the promotion subject has the minimum amount (method `getPromotionSubjectItemTotal()` of `PromotionSubjectInterface`) required by the rule.

The promotion eligibility checker service

To be eligible to a promotion, a subject must :

1. respect all the rules related to the promotion
2. respect promotion dates if promotion is limited by time
3. respect promotions usages count if promotion has a limited number of usages
4. if a coupon is provided with this order, it must be valid and belong to this promotion

The service `sylius.promotion_eligibility_checker` checks all these constraints for you with the method `isEligible()` which returns `true` or `false`. This service uses the class `PromotionEligibilityChecker`.

How actions are applied ?

Everything related to this subject is located in `Sylius\Bundle\PromotionsBundle\Action`.

Actions

Actions can be created by implementing `PromotionActionInterface`. This interface provides the method `execute` which aim is to apply a promotion to its subject. It also provides the method `getConfigurationFormType` which has to return the form name related to this action.

Actions have to be defined as services and have to use the tag named `sylius.promotion_action` with the attributes `type` and `label`.

As `SyliusPromotionsBundle` is totally independent, it does not provide some actions out of the box. Great examples of actions are provided by `Sylius/Standard-Edition`.

Note: `Sylius\Bundle\CoreBundle\Promotion\Action\FixedDiscountAction` from `Sylius/Standard-Edition` is an example of action for a fixed amount discount. The related service is called `sylius.promotion_action.fixed_discount`.

Note: `Sylius\Bundle\CoreBundle\Promotion\Action\PercentageDiscountAction` from `Sylius/Standard-Edition` is an example of action for a discount based on percentage. The related service is called `sylius.promotion_action.percentage_discount`.

All actions that you have defined as services will be automatically registered thanks to `Sylius\Bundle\PromotionsBundle\Action\Registry\PromotionActionRegistry`.

Applying actions to promotions

We have seen above how actions can be created. Now let's see how they are applied to their subject.

The `PromotionApplicator` is responsible of this via its method `apply`. This method will `execute` all the registered actions of a promotion on a subject.

How promotions are applied ?

By using the `promotion eligibility checker` and the `promotion applicator checker` services, the promotion processor applies all the possible promotions on a subject.

The promotion processor is defined via the service `sylius.promotion_processor` which uses the class `Sylius\Bundle\PromotionsBundle\Processor\PromotionProcessor`. Basically, it calls the method `apply` of the promotion applicator for all the active promotions that are eligible to the given subject.

Coupon based promotions

Coupon based promotions require special needs that are covered by this documentation.

Coupon generator

SyliusPromotionsBundle provides a way of generating coupons for a promotion : the coupon generator. Provided as a service sylius.generator.promotion_coupon via the class Sylius\Bundle\PromotionsBundle\Generator\CouponGenerator, its goal is to generate unique coupon codes.

Coupon to code transformer

SyliusPromotionsBundle provides a way to transform a simple string code to a real Coupon object (and vice versa). This is done via the Sylius\Bundle\PromotionsBundle\Form\DataTransformer\CouponToCodeTransformer class.

This data transformer is used by default with the Sylius\Bundle\PromotionsBundle\Form\Type\CouponToCodeType form, provided as the service sylius.form.type.promotion_coupon_to_code.

Note: An example of integration of this form can be found in the Sylius\Bundle\CoreBundle\Form\Type\CartType class of Sylius/Standard-Edition.

Coupon controller

The Sylius\Bundle\PromotionsBundle\Controller\CouponController provides an interface for easily generating new coupons.

Configuration reference

```
sylius_promotions:
    driver: ~ # The driver used for persistence layer.
    classes:
        promotion:
            model: Sylius\Bundle\PromotionsBundle\Model\Promotion
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\PromotionsBundle\Form\Type\PromotionType
        promotion_rule:
            model: Sylius\Bundle\PromotionsBundle\Model\Rule
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\PromotionsBundle\Form\Type\RuleType
        promotion_action:
            model: Sylius\Bundle\PromotionsBundle\Model\Action
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\PromotionsBundle\Form\Type\ActionType
        promotion_coupon:
            model: Sylius\Bundle\PromotionsBundle\Model\Coupon
            controller: Sylius\Bundle\PromotionsBundle\Controller\CouponController
            repository: ~
            form: Sylius\Bundle\PromotionsBundle\Form\Type\CouponType
        promotion_subject:
            model: ~
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
```

```
    validation_groups:
        promotion: [sylius]
        promotion_rule: [sylius]
        promotion_coupon: [sylius]
        promotion_action: [sylius]
        promotion_rule_item_total_configuration: [sylius]
        promotion_rule_item_count_configuration: [sylius]
        promotion_rule_user_loyality_configuration: [sylius]
        promotion_rule_shipping_country_configuration: [sylius]
        promotion_rule_taxonomy_configuration: [sylius]
        promotion_rule_nth_order_configuration: [sylius]
        promotion_action_fixed_discount_configuration: [sylius]
        promotion_action_percentage_discount_configuration: [sylius]
        promotion_action_add_product_configuration: [sylius]
        promotion_coupon_generate_instruction: [sylius]
        promotion_action_shipping_discount_configuration: [sylius]
```

4.1.11 SyliusOmnipayBundle

Symfony2 integration for Omnipay library, PHP abstraction for payment gateways.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require "sylius/omnipay-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/omnipay-bundle"
```

Adding required bundles to the kernel

First, you need to enable the bundle inside the kernel.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),

        new Sylius\Bundle\OmnipayBundle\SyliusOmnipayBundle()
    );
}
```

Configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_omnipay:
    gateways:
        AuthorizeNet_AIM:            # gateway name, use anyone
            type: AuthorizeNet_AIM   # predefined list of types, read father for explanation
            label: Authorize.Net AIM # how is gateway will be displayed in a form, etc.
        AuthorizeNet_SIM:
            type: AuthorizeNet_SIM
            label: Authorize.Net SIM
        Stripe:
            type: Stripe
            label: Stripe
            mode: true               # optional, default: false, activate test mode
            active: false            # optional, default: true, does this gateway is active
            options:                 # optional, predefine list of options to get work with gateway
                apikey: secretapikey
        PayPal_Express:
            type: PayPal_Express
            label: PayPal Express
        PayPal_Pro:
            type: PayPal_Pro
            label: PayPal Pro
```

Implemented gateways

The following gateways are already implemented:

- 2Checkout
- Authorize.Net AIM
- Authorize.Net SIM
- CardSave
- Dummy
- GoCardless
- Manual
- Netaxept (BBS)
- PayFast
- Payflow Pro
- PaymentExpress (DPS) PxPay
- PaymentExpress (DPS) PxPost
- PayPal Express Checkout
- PayPal Payments Pro
- Pin Payments
- Sage Pay Direct
- Sage Pay Server

- Stripe
- WorldPay

The list above is always growing. The full list of supported gateways can be found at the Omnipay github repository.

How to manage gateways

The bundle provide handy services, which help to manage the activated gateways and options.

Services

Here are a couple of services which are available to use out of the box.

- **sylius.omnipay.gateway_factory** - implement *Omnipay\Common\GatewayFactory* to manipulate with gateway and options
- **sylius.form.type.omnipay.gateway_choice** - quick proxy service to list defined in configuration gateways

Controller

To be written.

4.1.12 SyliusTaxonomiesBundle

Flexible categorization system for Symfony2 applications.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download the package.

If you have Composer installed globally.

```
$ composer require "sylius/taxonomies-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/taxonomies-bundle"
```

Note: This version is compatible only with Symfony 2.3 or newer. Please see the CHANGELOG file in the repository, to find version to use with older vendors.

Adding required bundles to the kernel

First, you need to enable the bundle inside the kernel. If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies to the kernel. Don't worry, everything was automatically installed via Composer.

```php
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),
        new Sylius\Bundle\TaxonomiesBundle\SyliusTaxonomiesBundle(),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    );
}
```

Note: Please register the bundle before *DoctrineBundle*. This is important as we use listeners which have to be processed first.

Container configuration

Put this configuration inside your `app/config/config.yml`.

```yaml
sylius_taxonomies:
    driver: doctrine/orm # Configure the doctrine orm driver used in documentation.
```

And configure doctrine extensions which are used in the taxonomies bundle:

```yaml
stof_doctrine_extensions:
    orm:
        default:
            tree: true
            sluggable: true
```

Routing configuration

Add the following lines to your `app/config/routing.yml`.

```yaml
sylius_taxonomies:
    resource: @SyliusTaxonomiesBundle/Resources/config/routing.yml
    prefix: /taxonomies
```

Updating database schema

Run the following command.

```
$ php app/console doctrine:schema:update --force
```

Warning: This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

Taxonomy and Taxons

Taxonomy is a list constructed from individual Taxons. Every taxonomy has one special taxon, which serves as a root of the tree. All taxons can have many child taxons, you can define as many of them as you need.

A good examples of taxonomies are "Categories" and "Brands". Below you can see an example tree.

```
| Categories
|--  T-Shirts
|    |-- Men
|    `-- Women
|--  Stickers
|--  Mugs
`--  Books

| Brands
|-- SuperTees
|-- Stickypicky
|-- Mugland
`-- Bookmania
```

Here is the full list of attributes related to Taxonomy and Taxon models.

Taxonomy	
Attribute	Description
id	Unique id of the taxonomy
name	Name of the taxonomy
root	First, "root" Taxon
createdAt	Date when taxonomy was created
updatedAt	Date of last update

Taxon	
Attribute	Description
id	Unique id of the taxon
name	Name of the taxon
slug	Urlized name
permalink	Full permalink for given taxon
description	Description of taxon
taxonomy	Taxonomy
parent	Parent taxon
children	Sub taxons
left	Location within taxonomy
right	Location within taxonomy
level	How deep it is in the tree
createdAt	Date when taxon was created
updatedAt	Date of last update

Retrieving taxonomies and taxons

Retrieving taxonomy from database should always happen via repository, which implements `Sylius\Bundle\ResourceBundle\Model\RepositoryInterface`. If you are using Doctrine, you're already familiar with this concept, as it extends the native Doctrine `ObjectRepository` interface.

Your taxonomy repository is always accessible via `sylius.repository.taxonomy` service. Of course, `sylius.repository.taxon` is also available for use, but usually you obtains taxons directly from Taxonomy model. You'll see that in further parts of this document.

```php
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.taxonomy');
}
```

Retrieving taxonomies is simpleas calling proper methods on the repository.

```php
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.taxonomy');

    $taxonomy = $repository->find(2); // Get taxonomy with id 4, returns null if not found.
    $taxonomy = $repository->findOneBy(array('name' => 'Specials')); // Get one taxonomy by defined c

    $taxonomies = $repository->findAll(); // Load all the taxonomies!
    $taxonomies = $repository->findBy(array('hidden' => true)); // Find taxonomies matching some cust
}
```

Taxonomy repository also supports paginating taxonomies. To create a Pagerfanta instance use the `createPaginator` method.

```php
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.taxonomy');

    $taxonomies = $repository->createPaginator();
    $taxonomies->setMaxPerPage(5);
    $taxonomies->setCurrentPage($request->query->get('page', 1));

   // Now you can return taxonomies to template and iterate over it to get taxonomies from current pa
}
```

Paginator also can be created for specific criteria and with desired sorting.

```php
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.taxonomy');

    $taxonomies = $repository->createPaginator(array('foo' => true), array('createdAt' => 'desc'));
    $taxonomies->setMaxPerPage(3);
    $taxonomies->setCurrentPage($request->query->get('page', 1));
}
```

Creating new taxonomy object

To create new taxonomy instance, you can simply call `createNew()` method on the repository.

```php
<?php

public function myAction(Request $request)
```

```
{
    $repository = $this->container->get('sylius.repository.taxonomy');
    $taxonomy = $repository->createNew();
}
```

Note: Creating taxonomy via this factory method makes the code more testable, and allows you to change taxonomy class easily.

Saving & removing taxonomy

To save or remove a taxonomy, you can use any `ObjectManager` which manages Taxonomy. You can always access it via alias `sylius.manager.taxonomy`. But it's also perfectly fine if you use `doctrine.orm.entity_manager` or other appropriate manager service.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.taxonomy');
    $manager = $this->container->get('sylius.manager.taxonomy'); // Alias for appropriate doctrine ma

    $taxonomy = $repository->createNew();

    $taxonomy
        ->setName('Foo')
    ;

    $manager->persist($taxonomy);
    $manager->flush(); // Save changes in database.
}
```

To remove a taxonomy, you also use the manager.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.taxonomy');
    $manager = $this->container->get('sylius.manager.taxonomy');

    $taxonomy = $repository->find(5);

    $manager->remove($taxonomy);
    $manager->flush(); // Save changes in database.
}
```

Taxons

Taxons can be handled exactly the same way, but with usage of `sylius.repository.taxon`.

Taxonomy contains methods which allow you to retrieve the child taxons. Let's look again at our example tree.

```
| Categories
|--  T-Shirts
|    |-- Men
```

```
|     `-- Women
|--  Stickers
|--  Mugs
`--  Books
```

To get a flat list of taxons under taxonomy, use the `getTaxonsAsList` method.

```
<?php

public function myAction(Request $request)
{
    // Find the taxonomy
    $taxonomyRepository = $this->container->get('sylius.repository.taxonomy');
    $taxonomy = $taxonomyRepository->findOneByName('Categories');

    // Get the taxons as a list
    $taxonRepository = $this->container->get('sylius.repository.taxon');
    $taxons = $taxonRepository->getTaxonsAsList($taxonomy);
}
```

`$taxons` variable will now contain flat list (ArrayCollection) of taxons in following order: T-Shirts, Men, Women, Stickers, Mugs, Books.

If, for example, you want to render them as tree.

```
<?php

public function myAction(Request $request)
{
    $repository = $this->container->get('sylius.repository.taxonomy');
    $taxonomy = $repository->findOneByName('Categories');

    $taxons = $taxonomy->getTaxons();
}
```

Now `$taxons` contains only the 4 main items, and you can access their children by calling `$taxon->getChildren()`.

Configuration reference

```
sylius_taxonomies:
    driver: ~ # The driver used for persistence layer.
    classes:
        taxonomy:
            model: Sylius\Bundle\TaxonomiesBundle\Model\Taxonomy
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~ # Taxonomy repository class.
            form: Sylius\Bundle\TaxonomiesBundle\Form\Type\TaxonomyType # Taxonomy form type class na
        taxon:
            model: Sylius\Bundle\TaxonomiesBundle\Model\Taxon
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~ # Taxon repository class.
            form: Sylius\Bundle\TaxonomiesBundle\Form\Type\TaxonType # Taxon form type class name.
    validation_groups:
        taxonomy: [sylius] # Taxonomy validation groups.
        taxon: [sylius] # Taxon validation groups.
```

4.1.13 SyliusSettingsBundle

Settings system with web editing interface for Symfony2 applications.

Allowing application users to edit some global configuration and storing it in database is a common need in a variety of projects. This bundle provides exactly this feature - you only need to define the configuration.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download the package.

If you have Composer installed globally.

```
$ composer require "sylius/settings-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/settings-bundle"
```

Adding required bundles to the kernel

First, you need to enable the bundle inside the kernel. If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies to the kernel. This bundle also uses *LiipDoctrineCacheBundle*. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
        new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        new Liip\DoctrineCacheBundle\LiipDoctrineCacheBundle(),
        new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),
        new Sylius\Bundle\SettingsBundle\SyliusSettingsBundle(),
        new Sylius\Bundle\ResourceBundle\SyliusResourceBundle(),

        // Other bundles...
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    );
}

Please register the bundle before *DoctrineBundle*. This is important as we use listeners which have
```

Container configuration

Put this configuration inside your `app/config/config.yml`.

```
sylius_settings:
    driver: doctrine/orm

liip_doctrine_cache:
    namespaces:
        sylius_settings:
            type: file_system
```

Importing routing configuration

Import default routing from your `app/config/routing.yml`.

```
sylius_settings:
    resource: @SyliusSettingsBundle/Resources/config/routing.yml
    prefix: /settings
```

Note: We used `default` namespace in this example. If you want to use other namespaces for saving your settings, routing config should be updated as it contains the namespace parameter.

Updating database schema

Run the following command.

```
$ php app/console doctrine:schema:update --force
```

> **Warning:** This should be done only in **dev** environment! We recommend using Doctrine migrations, to safely update your schema.

Creating your settings schema

You have to create a new class implementing **SchemaInterface**, which will represent the structure of your configuration. For purpose of this tutorial, let's define the page metadata settings.

```
<?php

// src/Acme/ShopBundle/Settings/MetaSettingsSchema.php

namespace Acme\ShopBundle\Settings;

use Sylius\Bundle\SettingsBundle\Schema\SchemaInterface;
use Sylius\Bundle\SettingsBundle\Schema\SettingsBuilderInterface;
use Symfony\Component\Form\FormBuilderInterface;

class MetaSettingsSchema implements SchemaInterface
{
    public function buildSettings(SettingsBuilderInterface $builder)
    {
        $builder
            ->setDefaults(array(
                'title'            => 'Sylius - Modern ecommerce for Symfony2',
                'meta_keywords'    => 'symfony, sylius, ecommerce, webshop, shopping cart',
                'meta_description' => 'Sylius is modern ecommerce solution for PHP. Based on the Sym
```

```
            ))
            ->setAllowedTypes(array(
                'title'            => array('string'),
                'meta_keywords'    => array('string'),
                'meta_description' => array('string'),
            ))
        ;
    }

    public function buildForm(FormBuilderInterface $builder)
    {
        $builder
            ->add('title')
            ->add('meta_keywords')
            ->add('meta_description', 'textarea')
        ;
    }
}
```

Note: **SettingsBuilderInterface** is extended version of Symfony's OptionsResolver component.

As you can see there are two methods in our schema and both are very simple. First one, the `->buildSettings()` defines default values and allowed data types. Second, `->buildForm()` creates the form to be used in the web interface to update the settings.

Now, lets register our **MetaSettingsSchema** service. Remember that we are tagging it as *sylius.settings_schema*:

```
<service id="acme.settings_schema.meta" class="Acme\ShopBundle\Settings\MetaSettingsSchema">
    <tag name="sylius.settings_schema" namespace="meta" />
</service>
```

Your new settings schema is available for use.

Editing the settings

To edit the settings via the web interface, simply point users to the `sylius_settings_update` route with proper parameters.

In order to update our meta settings, generate the following link.

```
<a href="{{ path('sylius_settings_update', {'namespace': 'meta'}) }}">Edit SEO</a>
```

A proper form will be generated, with a submit action, which updates the settings in database.

Using in templates

Bundle provides handy **SyliusSettingsExtension** which you can use in your templates.

In our example, it can be something like:

```
{% set meta = sylius_settings_all('meta') %}

<head>
    <title>{{ meta.title }}</title>
    <meta name="keywords" content="{{ meta.meta_keywords }}">
    <meta name="description" content="{{ meta.meta_description }}">
</head>
```

There is also *sylius_settings_get()* to get particular setting directly.

```
{% set title = sylius_settings_get('meta.title') %}

<head>
    <title>{{ title }}</title>
</head>
```

Using the settings in services

You can also load and save the settings in any service. Simply use the **SettingsManager** service, available under the `sylius.settings.manager` id.

Loading the settings

```
<?php

// src/Acme/ShopBundle/Taxation/TaxApplicator.php

namespace Acme\ShopBundle\Taxation;

use Sylius\Bundle\SettingsBundle\Manager\SettingsManagerInterface;

class TaxApplicator
{
    private $settingsManager;

    public function __construct(SettingsManagerInterface $settingsManager)
    {
        $this->settingsManager = $settingsManager;
    }

    public function applyTaxes(Order $order)
    {
        $taxationSettings = $this->settingsManager->loadSettings('taxation');
        $itemsTotal = $order->getItemsTotal();

        $order->setTaxTotal($taxationSettings->get('rate') * $itemsTotal);
    }
}
```

Injecting the settings manager is as simple as using any other service.

```
<service id="acme.tax_applicator" class="Acme\ShopBundle\Taxation\TaxApplicator">
    <argument type="service" id="sylius.settings.manager" />
</service>
```

Saving the settings

Note: To be written.

Configuration reference

```
sylius_settings:
    driver: ~ # The driver used for persistence layer.
    classes:
        parameter:
            model: Sylius\Bundle\SettingsBundle\Model\Parameter
            controller: Sylius\Bundle\ResourceBundle\Controller\ResourceController
            repository: ~
            form: Sylius\Bundle\SettingsBundle\Form\Type\ParameterType
```

4.1.14 SyliusFlowBundle

Wizards with reusable steps for Symfony2 applications. Suitable for building checkouts or installations.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use the following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require "sylius/flow-bundle"
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require "sylius/flow-bundle"
```

Adding required bundles to the kernel

First, you need to enable the bundle inside the kernel. If you're not using any other Sylius bundles, you will also need to add *SyliusResourceBundle* and its dependencies to the kernel. Don't worry, everything was automatically installed via Composer.

```
<?php

// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        new Sylius\Bundle\FlowBundle\SyliusFlowBundle(),

        // Other bundles...
    );
}
```

Creating your steps

We will create a very simple wizard now, without forms, storage, to keep things simple and get started fast.

Lets create a few simple steps:

```php
<?php

namespace Acme\DemoBundle\Process\Step;

use Sylius\Bundle\FlowBundle\Process\Context\ProcessContextInterface;
use Sylius\Bundle\FlowBundle\Process\Step\ControllerStep;

class FirstStep extends ControllerStep
{
    public function displayAction(ProcessContextInterface $context)
    {
        return $this->render('AcmeDemoBundle:Process/Step:first.html.twig');
    }

    public function forwardAction(ProcessContextInterface $context)
    {
        return $this->complete();
    }
}
```

```php
<?php

namespace Acme\DemoBundle\Process\Step;

use Sylius\Bundle\FlowBundle\Process\Context\ProcessContextInterface;
use Sylius\Bundle\FlowBundle\Process\Step\ControllerStep;

class SecondStep extends ControllerStep
{
    public function displayAction(ProcessContextInterface $context)
    {
        return $this->render('AcmeDemoBundle:Process/Step:second.html.twig');
    }

    public function forwardAction(ProcessContextInterface $context)
    {
        return $this->complete();
    }
}
```

And so on, one class for each step in the wizard.

As you can see, there are two actions in each step, display and forward. Usually, there is a form in a forward action where you can pick some data. When you do `return $this->complete()` the wizard will take you to the next step.

Creating scenario

To group steps into the wizard, we will implement *ProcessScenarioInterface*:

```php
<?php

namespace Acme\DemoBundle\Process;

use Sylius\Bundle\FlowBundle\Process\Builder\ProcessBuilderInterface;
use Sylius\Bundle\FlowBundle\Process\Scenario\ProcessScenarioInterface;
use Symfony\Component\DependencyInjection\ContainerAware;
```

```
use Acme\DemoBundle\Process\Step;

class SyliusScenario extends ContainerAware implements ProcessScenarioInterface
{
    public function build(ProcessBuilderInterface $builder)
    {
        $builder
            ->add('first', new Step\FirstStep())
            ->add('second', new Step\SecondStep())
            // ...
        ;
    }
}
```

As you can see, we just add each step to process builder with a desired name. The name will be used in the routes to navigate to particular step.

Registering scenario

In order for this to work, we need to register *SyliusScenario* and tag it as `sylius.process.scenario`:

```
<service id="sylius.scenario.flow" class="Acme\DemoBundle\Process\SyliusScenario">
    <call method="setContainer">
        <argument type="service" id="service_container" />
    </call>
    <tag name="sylius.process.scenario" alias="sylius_flow" />
</service>
```

The configured alias will be used later in the route parameters to identify the scenario as you can have more then one.

Routing configuration

Import routing configuration:

```
sylius_flow:
    resource: @SyliusFlowBundle/Resources/config/routing.yml
    prefix: /flow
```

If you take a look into imported routing configuration, you will see that `sylius_flow_start` is a wizard entry point. `sylius_flow_display` displays the step with the given name, `sylius_flow_forward` forwards to the next step from the step with the given name. All routes have an *scenarioAlias* as a required parameter to identify the scenario.

Templates

Step templates are like any other action template, usually due to the nature of multi-step wizards, they have back and forward buttons:

```
<h1>Welcome to second step</h1>
<a href="{{ path('sylius_flow_display', {'scenarioAlias': 'sylius_flow', 'stepName': 'first'}) }}" cl
<a href="{{ path('sylius_flow_forward', {'scenarioAlias': 'sylius_flow', 'stepName': 'second'}) }}" c
```

Using storage

Storing data with default storage

By default, flow bundle will use session for data storage. Here is simple example how to use it in your steps:

```
<?php

namespace Acme\DemoBundle\Process\Step;

use Sylius\Bundle\FlowBundle\Process\Context\ProcessContextInterface;
use Sylius\Bundle\FlowBundle\Process\Step\ControllerStep;

class FirstStep extends ControllerStep
{
    // ...

    public function forwardAction(ProcessContextInterface $context)
    {
        $request = $this->getRequest();
        $form = $this->createForm('my_form');

        if ($request->isMethod('POST') && $form->bind($request)->isValid()) {
            $context->getStorage()->set('my_data', $form->getData());

            return $this->complete();
        }

        return $this->render('AcmeDemoBundle:Process/Step:first.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

You can later get data with `$context->getStorage()->get('my_data')`.

For more details about storage, check **SyliusBundleFlowBundleStorageStorageInterface** class.

Summary

Configuration reference

```
sylius_flow:
    storage: sylius.process_storage.session # The storage used to save flow data.
```

Tests

```
$ composer install --dev --prefer-dist
$ phpunit
```

Working examples

If you want to see working implementation, try out the Sylius application.

There is also an example that shows how to integrate this bundle into Symfony Standard Edition.

Bug tracking

This bundle uses GitHub issues. If you have found bug, please create an issue.

- *Bundles General Guide*
- *SyliusResourceBundle*
- *SyliusProductBundle*
- *SyliusOrderBundle*
- *SyliusCartBundle*
- *SyliusAddressingBundle*
- *SyliusInventoryBundle*
- *SyliusShippingBundle*
- *SyliusTaxationBundle*
- *SyliusPromotionsBundle*
- *SyliusTaxonomiesBundle*
- *SyliusOmnipayBundle*
- *SyliusSettingsBundle*
- *SyliusFlowBundle*
- *Bundles General Guide*
- *SyliusResourceBundle*
- *SyliusProductBundle*
- *SyliusOrderBundle*
- *SyliusCartBundle*
- *SyliusAddressingBundle*
- *SyliusInventoryBundle*
- *SyliusShippingBundle*
- *SyliusTaxationBundle*
- *SyliusPromotionsBundle*
- *SyliusTaxonomiesBundle*
- *SyliusOmnipayBundle*
- *SyliusSettingsBundle*
- *SyliusFlowBundle*

CHAPTER 5

Components

E-Commerce components for PHP.

5.1 PHP Ecommerce Components

5.1.1 Addressing

Sylius Addressing component for PHP E-Commerce applications.

The component provides you with basic Address, Country, Province and Zone models.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/addressing:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/addressing:*
```

The Address

This is a very basic representation of Address model.

Attribute	Description
id	Unique id of the address
firstname	
lastname	
company	
country	Reference to Country model
province	Reference to Province model
street	
city	
postcode	
createdAt	Date when address was created
updatedAt	Date of last address update

Creating and modifying Address object is very simple.

```
<?php

use Sylius\Component\Addressing\Address;

$address = new Address();

$address
    ->setFirstname('John')
    ->setLastname('Doe')
    ->setStreet('Superstreet 14')
    ->setCity('New York')
    ->setPostcode('13111')
;

$user = // Get your user from somewhere or any model which can reference an Address.
$user->setShippingAddress($address);
```

Country

"Country" model represents a geographical area of a country.

Attribute	Description
id	Unique id of the country
name	
isoName	
provinces	Collection of provinces
createdAt	Date when country was created
updatedAt	Date of last country update

```
<?php

use Sylius\Component\Addressing\Address;
use Sylius\Component\Addressing\Country;

$poland = new Country();
$poland
    ->setName('Poland')
    ->setIsoName('PL')
;

$address
    ->setFirstname('Pawel')
```

```
    ->setLastname('Jedrzejewski')
    ->setCountry($poland)
    ->setStreet('Testing 123')
    ->setCity('Lodz')
    ->setPostcode('21-242')
;
```

Province

"Province" is a smaller area inside of a Country. You can use it to manage provinces or states and assign it to an address as well.

Attribute	Description
id	Unique id of the province
name	
country	Reference to a country
createdAt	Date when province was created
updatedAt	Date of last update

```
<?php

use Sylius\Component\Addressing\Address;
use Sylius\Component\Addressing\Country;
use Sylius\Component\Addressing\Province;

$usa = new Country();
$usa
    ->setName('United States of America')
    ->setIsoName('US')
;

$tennessee = new Province();
$tennessee->setName('Tennessee');

$address
    ->setFirstname('John')
    ->setLastname('Deo')
    ->setCountry($usa)
    ->setProvince($tennessee)
    ->setStreet('Testing 111')
    ->setCity('Nashville')
    ->setPostcode('123123')
;
```

Zones and Zone Matching

This library allows you to define **Zones**, which represent a specific geographical area.

Zone Model

Every **Zone** is represented by the following model:

Attribute	Description
id	Unique id of the zone
name	
type	String type of zone
members	Zone members
createdAt	Date when zone was created
updatedAt	Date of last update

It exposes the following API:

```
<?php

$zone->getName(); // Name of the zone, for example "EU".
$zone->getType(); // Type, for example "country".

foreach ($zone->getMembers() as $member) {
    echo $member->getName();
}

$zone->getCreatedAt();
$zone->getUpdatedAt();
```

Three different types of zones are supported out-of-the-box.

- *country* zone, which consists of many countries.
- *province* zone, which is constructed from multiple provinces.
- *zone*, which is a group of other zones.

Each zone type has different **ZoneMember** model, but they all expose the same API:

```
<?php

foreach ($zone->getMembers() as $member) {
    echo $member->getName();

    echo $member->getZone()->getName(); // Name of the zone.
}
```

There are following models and each of them represents a different zone member:

- ZoneMemberCountry
- ZoneMemberProvince
- ZoneMemberZone

Each **ZoneMember** instance holds a reference to the **Zone** object and appropriate area entity, for example a **Country**.

Creating a zone requires adding appropriate members:

```
<?php

use Sylius\Component\Addressing\Country;
use Sylius\Component\Addressing\Zone;
use Sylius\Component\Addressing\ZoneInterface;
use Sylius\Component\Addressing\ZoneMemberCountry;

$eu = new Zone();
$eu
    ->setName('European Union')
    ->setType(ZoneInterface::TYPE_COUNTRY)
```

```
;

$germany = new Country();
$germany
    ->setName('Germany')
    ->setIsoName('DE')
;
$france = new Country();
$france
    ->setName('France')
    ->setIsoName('FR')
;
$poland = new Country();
$poland
    ->setName('Poland')
    ->setIsoName('PL')
;

$germanyMember = new ZoneMemberCountry();
$germanyMember->setCountry($germany)

$franceMember = new ZoneMemberCountry();
$franceMember->setCountry($france)

$polandMember = new ZoneMemberCountry();
$polandMember->setCountry($poland)

$eu
    ->addMember($germany)
    ->addMember($france)
    ->addMember($poland)
;
```

Tip: Default zone types are defined as constants in the `ZoneInterface` interface.

Exactly the same process applies to different types of Zones.

Matching a Zone

Zones are not very useful by themselves, but they can be part of a complex taxation/shipping or any other system. A service implementing the *ZoneMatcherInterface* is responsible for matching the **Address** to a specific **Zone**.

Default implementation uses a collaborator implementing `RepositoryInterface` to obtain all available zones and then compares them with the given **Address**.

```
<?php

use Sylius\Component\Addressing\Matcher\ZoneMatcher;

$zoneRepository = // Get the repository.
$zoneMatcher = new ZoneMatcher($zoneRepository);

$zone = $zoneMatcher->match($user->getAddress());

// Apply appropriate taxes or return shipping methods supported for given zone.
```

Zone matcher can also return all matching zones. (not only the best one)

```php
<?php

$zones = $zoneMatcher->matchAll($user->getAddress());

// Inventory can be take from stock locations in the following zones...
```

There are many other use-cases!

5.1.2 Attribute

Handling of dynamic attributes on PHP models is a common task for most of modern business applications. Sylius component makes it easier to handle different types of attributes and attach them to any object by implementing a simple interface.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/attribute:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/attribute:*
```

Attributes and AttributeValues

Every attribute is represented by the **Attribute** instance and has following properties:

Attribute	Description
id	Unique id of the attribute
name	Name of the attribute ("tshirt_material")
presentation	Pretty name visible for user ("Material")
type	Attribute type
createdAt	Date when attribute was created
updatedAt	Date of last attribute update

The following attribute types are available by default.

Type
text
number
percentage
checkbox
choice

Implementing SubjectInterface

To characterize an object using attribute, it needs to implement the **SubjectInterface** interface:

- `getAttributes()`

- `setAttributes(Collection $attributes)`
- `addAttribute(AttributeValue $attribute)`
- `removeAttribute(AttributeValue $attribute)`
- `hasAttribute(AttributeValue $attribute)`
- `hasAttributeByName($attributeName)`
- `getAttributeByName($attributeName)`

5.1.3 Cart

Common models, services and interface to handle a shopping cart in PHP e-commerce application.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/cart:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/cart:*
```

5.1.4 Currency

Managing different currencies, exchange rates and converting cash amounts for PHP apps.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/currency:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/currency:*
```

5.1.5 Inventory

Inventory management for PHP applications.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/inventory:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/inventory:*
```

5.1.6 Order

E-Commerce PHP library for creating and managing sales orders.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/order:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/order:*
```

The Order and OrderItem

The library comes with 2 basic models representing the **Order** and it **OrderItems**.

Order Basics

Order is constructed with the following attributes:

Attribute	Description
id	Unique id of the order
number	Human-friendly number
state	String represeting the status
items	Collection of OrderItems
itemsTotal	Total value of the items
adjustments	Collection of Adjustments
adjustmentsTotal	Total value of adjustments
total	Order grand total
confirmed	Boolean indicator of order confirmation
confirmationToken	Random string for order confirmation
createdAt	Date when order was created
updatedAt	Date of last change
completedAt	Checkout completion time
deletedAt	Date of deletion

Each order has 2 main identifiers, an *ID* and a human-friendly *number*. You can access those by calling `->getId()` and `-getNumber()` respectively. The number is mutable, so you can change it by calling `->setNumber('E001')` on the order instance.

```
<?php

$order->getId();
$order->getNumber();

$order->setNumber('E001');
```

Confirmation Status

To check whether the order is confirmed or not, you can use the `isConfirmed()` method, which returns a *true/false* value.

To change that status, you can use the confirmation setter, `setConfirmed(false)`. All orders are confirmed by default. Order can contain a confirmation token, accessible by the appropriate getter and setter.

```
<?php

if ($order->isConfirmed()) {
    echo 'This one is confirmed, great!';
}
```

Order Totals

Note: All money amounts in Sylius are represented as "cents" - integers.

An order has 3 basic totals, which are all persisted together with the order.

The first total is the *items total*, it is calculated as the sum of all item totals.

The second total is the *adjustments total*, you can read more about this in next chapter.

```
<?php

echo $order->getItemsTotal(); // 1900.
echo $order->getAdjustmentsTotal(); // -250.
```

```
$order->calculateTotal();
echo $order->getTotal(); // 1650.
```

The main order total is a sum of the previously mentioned values. You can access the order total value using the `->getTotal()` method.

Recalculation of totals can happen by calling `->calculateTotal()` method, using the simplest math. It will also update the item totals.

Items Management

The collection of items (Implementing the `Doctrine\Common\Collections\Collection` interface) can be obtained using the `->getItems()`. To add or remove items, you can simply use the `addItem` and `removeItem` methods.

```
<?php

use Sylius\Component\Order\Model\Order;
use Sylius\Component\Order\Model\OrderItem;

$order = new Order();

$item1 = new OrderItem();
$item1
    ->setName('Super cool product')
    ->setUnitPrice(1999) // 19.99!
    ->setQuantity(2)
;
$item1 = new OrderItem();
$item1
    ->setName('Interesting t-shirt')
    ->setUnitPrice(2549) // 25.49!
;

$order
    ->addItem($item1)
    ->addItem($item2)
    ->removeItem($item1)
;
```

OrderItem Basics

OrderItem model has the attributes listed below:

Attribute	Description
id	Unique id of the item
order	Reference to an Order
unitPrice	The price of a single unit
quantity	Quantity of sold item
adjustments	Collection of Adjustments
adjustmentsTotal	Total value of adjustments
total	Order grand total
createdAt	Date when order was created
updatedAt	Date of last change

An order item model has only the **id** property as identifier and it has the order reference, accessible via `->getOrder()` method.

```
<?php

echo $item->getId(); / Prints e.g. 12.
$item->setName($book);
```

Just like for the order, the total is available via the same method, but the unit price is accessible using the `->getUnitPrice()` Each item also can calculate its total, using the quantity (`->getQuantity()`) and the unit price.

```
<?php

use Sylius\Component\Order\Model\OrderItem;

$item = new OrderItem();
$item
    ->setName('Game of Thrones')
    ->setUnitPrice(2000)
    ->setQuantity(4)
    ->calculateTotal()
;

echo $item->getTotal(); // 8000.
```

An OrderItem can also hold adjustments.

Adjustments

Adjustment object represents an adjustment to the order's or order item's total.

Their amount can be positive (charges - taxes, shipping fees etc.) or negative (discounts etc.).

Adjustment Basics

Adjustments have the following properties:

Attribute	Description
id	Unique id of the adjustment
adjustable	Reference to Order or OrderItem
label	Type of the adjustment (e.g. "tax"")
description	e.g. "Clothing Tax 9%"
amount	Integer amount
neutral	Boolean flag of neutrality
createdAt	Date when adjustment was created
updatedAt	Date of last change

Neutral Adjustments

In some cases, you may want to use **Adjustment** just for displaying purposes. For example, when your order items have the tax already included in the price.

Every **Adjustment** instance has the *neutral* property, which indicates if it should be counted against object total.

```php
<?php

use Sylius\Component\Order\Order;
use Sylius\Component\Order\OrderItem;
use Sylius\Component\Order\Adjustment;

$order = new Order();
$tshirt = new OrderItem();
$tshirt
    ->setName('Awesome T-Shirt')
    ->setUnitPrice(4999)
;

$shippingFees = new Adjustment();
$shippingFees->setAmount(1000);

$tax = new Adjustment();
$tax
    ->setAmount(1150)
    ->setLabel
    ->setNeutral(true)
;

echo $order
    ->addItem($tshirt)
    ->addAdjustment($shippingFees)
    ->addAdjustment($tax)
    ->calculateTotal()
    ->getTotal()
;

// Output will be 5999.
```

Negative Adjustments

Adjustments can also have negative amounts, which means that they will decrease the order total by certain amount. Let's add a 5$ discount to the previous example.

```php
<?php

use Sylius\Component\Order\Order;
use Sylius\Component\Order\OrderItem;
use Sylius\Component\Order\Adjustment;

$order = new Order();
$tshirt = new OrderItem();
$tshirt
    ->setName('Awesome T-Shirt')
    ->setUnitPrice(4999)
;

$shippingFees = new Adjustment();
$shippingFees->setAmount(1000);

$tax = new Adjustment();
$tax
    ->setAmount(1150)
```

```
    ->setLabel
    ->setNeutral(true)
;

$discount = new Adjustment();
$discount->setAmount(500);

echo $order
    ->addItem($tshirt)
    ->addAdjustment($shippingFees)
    ->addAdjustment($tax)
    ->addAdjustment($discount)
    ->calculateTotal()
    ->getTotal()
;

// Output will be 5499.
```

Order States

Sylius itself uses a complex state machine system to manage all states of the business domain. This component has some sensible default states defined in the **OrderInterface**.

Default States

All new **Order** instances have the state `cart` by default, which means they are unconfirmed.

The following states are defined:

State	Description
cart	Unconfirmed order, ready to add/remove items
cart_locked	Cart which should not be removed when expired
pending	Order waiting for confirmation
confirmed	Confirmed and completed order
shipped	Order has been shipped to the customer
abandoned	Been waiting too long for confirmation
cancelled	Cancelled by customer or manager
returned	Order has been returned and refunded

Note: Please keep in mind that these states are just default, you can define and use your own. If you use this component with **SyliusOrderBundle** and Symfony2, you will have full state machine configuration at your disposal.

5.1.7 Payment

PHP library which provides abstraction of payments management. It ships with default Payment, PaymentMethod and CreditCard models.

Note: This bundle does not provide any payment gateway. Integrating it with Payum.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/payment:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/payment:*
```

5.1.8 Pricing

Calculating prices is a common task for most PHP E-Commerce applications. This library provides multiple strategies and flexible calculators system.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/pricing:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/pricing:*
```

5.1.9 Product

Powerful products catalog for PHP applications.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/product:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/product:*
```

5.1.10 Promotion

Super-flexible promotions system with support of complex rules and actions. Coupon codes included!

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/promotion:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/promotion:*
```

5.1.11 Registry

Simple registry component useful for all type of applications.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/registry:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/registry:*
```

5.1.12 Resource

Domain management abstraction for PHP. It provides interface for most common operations on the application resources.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/resource:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/resource:*
```

5.1.13 Sequence

Component for generating number/hash sequences in PHP.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/sequence:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/sequence:*
```

5.1.14 Shipping

Shipments and shipping methods management for PHP E-Commerce apps. It contains flexible calculators system for computing the shipping costs.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/shipping:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/shipping:*
```

5.1.15 Taxation

Tax rates and tax classification for PHP apps. You can define different tax categories and match them to objects.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/taxation:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/taxation:*
```

5.1.16 Taxonomy

Basic taxonomies library for any PHP application.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/taxonomy:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/taxonomy:*
```

5.1.17 Variation

Library for managing object variants and options. This functionality can be attached to any object to create different configurations.

Installation

We assume you're familiar with Composer, a dependency manager for PHP. Use following command to add the bundle to your *composer.json* and download package.

If you have Composer installed globally.

```
$ composer require sylius/variation:*
```

Otherwise you have to download .phar file.

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar require sylius/variation:*
```

- *Addressing*
- *Attribute*
- *Cart*
- *Currency*
- *Inventory*
- *Order*
- *Payment*
- *Pricing*
- *Product*
- *Promotion*
- *Registry*
- *Resource*
- *Sequence*
- *Shipping*
- *Taxation*

- *Taxonomy*
- *Variation*
- *Addressing*
- *Attribute*
- *Cart*
- *Currency*
- *Inventory*
- *Order*
- *Payment*
- *Pricing*
- *Product*
- *Promotion*
- *Registry*
- *Resource*
- *Sequence*
- *Shipping*
- *Taxation*
- *Taxonomy*
- *Variation*

CHAPTER 6

Contributing

A guide to contribute to Sylius.

6.1 Contributing

Note: This section is based on the great Symfony2 documentation.

6.1.1 Contributing Code

Reporting a Bug

Whenever you find a bug in Sylius, we kindly ask you to report it. It helps us make a better e-commerce solution for PHP.

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 stackoverflow.com or on the #sylius IRC channel if you're not sure 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 Behat scenario that illustrates the bug is best);
- Give as much detail as possible about your environment (OS, PHP version, Symfony version, Sylius 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 Sylius.

Step 1: Setup your Environment

Install the Software Stack Before working on Sylius, setup a Symfony2 friendly environment with the following software:

- Git;
- PHP version 5.3.3 or above;
- PHPUnit 3.6.4 or above;
- MySQL.

Configure Git 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, you are highly recommended to read the excellent and free ProGit book.

Tip: If your IDE creates configuration files inside the project's directory, you can use global `.gitignore` file (for all projects) or `.git/info/exclude` file (per project) to ignore them. See Github's documentation.

Tip: Windows users: when installing Git, the installer will ask what to do with line endings, and suggests replacing all LF with CRLF. This is the wrong setting if you wish to contribute to Sylius. 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. Change it to "input" 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 Sylius Source Code Get the Sylius source code:

- Create a GitHub account and sign in;
- Fork the Sylius repository (click on the "Fork" button);
- After the "forking action" has completed, clone your fork locally (this will create a `Sylius` directory):

```
$ git clone git@github.com:USERNAME/Sylius.git
```

- Add the upstream repository as a remote:

```
$ cd sylius
$ git remote add upstream git://github.com/Sylius/Sylius.git
```

Step 2: Work on your Patch

The License Before you start, you must know that all the patches you are going to submit must be released under the *MIT license*, unless explicitly specified in your commits.

Create a Topic Branch Each time you want to work on a patch for a bug or on an enhancement, create a topic branch:

```
$ git checkout -b BRANCH_NAME master
```

Tip: Use a descriptive name for your branch (`issue_XXX` where `XXX` is the GitHub issue 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 your Patch Work on the code as much as you want and commit as much as you want; but keep in mind the following:

- Practice *BDD*, which is the development methodology we use at Sylius;
- Read about the Sylius *conventions* and follow the coding *standards* (use `git diff --check` to check for trailing spaces – also read the tip below);
- Do atomic and logically separate commits (use the power of `git rebase` to have a clean and logical history);
- Squash irrelevant commits that are just about fixing coding standards or fixing typos in your own code;
- Never fix coding standards in some existing code as it makes the code review more difficult (submit CS fixes as a separate patch);
- Write good commit messages (see the tip below).

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 (`[Cart]`, `[Taxation]`, ...). Use a verb (`fixed ...`, `added ...`, ...) to start the summary and **don't add a period at the end**.

Prepare your Patch for Submission When your patch is not about a bug fix (when you add a new feature or change an existing one for instance), it must also include the following:

- An explanation of the changes in the relevant `CHANGELOG` file(s) (the `[BC BREAK]` or the `[DEPRECATION]` prefix must be used when relevant);
- An explanation on how to upgrade an existing application in the relevant `UPGRADE` file(s) if the changes break backward compatibility or if you deprecate something that will ultimately break backward compatibility.

Step 3: Submit your Patch

Whenever you feel that your patch is ready for submission, follow the following steps.

Rebase your 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
```

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
```

Push your branch remotely:

```
$ git push --force origin BRANCH_NAME
```

Make a Pull Request You can now make a pull request on the `Sylius/Sylius` GitHub repository.

To ease the core team work, always include the modified components in your pull request message, like in:

```
[Cart] Fixed something
[Taxation] [Addressing] Added something
```

The pull request description must include the following checklist at the top to ensure that contributions may be reviewed without needless feedback loops and that your contributions can be included into Sylius as quickly as possible:

```
| Q             | A
| ------------- | ---
| Bug fix?      | [yes|no]
| New feature?  | [yes|no]
| BC breaks?    | [yes|no]
| Deprecations? | [yes|no]
| Tests pass?   | [yes|no]
| Fixed tickets | [comma separated list of tickets fixed by the PR]
| License       | MIT
| Doc PR        | [The reference to the documentation PR if any]
```

An example submission could now look as follows:

```
| Q             | A
| ------------- | ---
| Bug fix?      | no
| New feature?  | no
| BC breaks?    | no
| Deprecations? | no
| Fixed tickets | #12, #43
| License       | MIT
| Doc PR        | Sylius/Sylius-Docs#123
```

The whole table must be included (do **not** remove lines that you think are not relevant). For simple typos, minor changes in the PHPDocs, or changes in translation files, use the shorter version of the check-list:

```
| Q             | A
| ------------- | ---
| Fixed tickets | [comma separated list of tickets fixed by the PR]
| License       | MIT
```

Some answers to the questions trigger some more requirements:

- If you answer yes to "Bug fix?", check if the bug is already listed in the Sylius issues and reference it/them in "Fixed tickets";
- If you answer yes to "New feature?", you must submit a pull request to the documentation and reference it under the "Doc PR" section;

- If you answer yes to "BC breaks?", the patch must contain updates to the relevant `CHANGELOG` and `UPGRADE` files;
- If you answer yes to "Deprecations?", the patch must contain updates to the relevant `CHANGELOG` and `UPGRADE` files;

If some of the previous requirements are not met, create a todo-list and add relevant items:

```
- [ ] Fix the specs as they have not been updated yet
- [ ] Submit changes to the documentation
- [ ] Document the BC breaks
```

If the code is not finished yet because you don't have time to finish it or because you want early feedback on your work, add an item to todo-list:

```
- [ ] Finish the feature
- [ ] Gather feedback for my changes
```

As long as you have items in the todo-list, please prefix the pull request title with "[WIP]".

In the pull request description, give as much details as possible about your changes (don't hesitate to give code examples to illustrate your points). If your pull request is about adding a new feature or modifying an existing one, explain the rationale for the changes. The pull request description helps the code review.

In addition to this "code" pull request, you must also send a pull request to the documentation repository to update the documentation when appropriate.

Rework your Patch Based on the feedback on the pull request, you might need to rework your patch. Before re-submitting the patch, rebase with `upstream/master`, don't merge; and force the push to the origin:

```
$ git rebase -f upstream/master
$ git push --force origin BRANCH_NAME
```

Note: When doing a `push --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, Sylius team members 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 upstream/master
$ git push --force origin BRANCH_NAME
```

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 successful, will ask you to edit the commit message, which by default is a listing of the commit messages of all the commits. When you are finished, execute the push command.

Security Issues

This document explains how Sylius issues are handled by the Sylius core team.

Reporting a Security Issue

If you think that you have found a security issue in Sylius, don't use the bug tracker and do not post it publicly. Instead, all security issues must be sent to **security [at] sylius.org**. Emails sent to this address are forwarded to the Sylius core team members.

Resolving Process

For each report, we first try to confirm the vulnerability. When it is confirmed, the team works on a solution following these steps:

1. Send an acknowledgement to the reporter;
2. Work on a patch;
3. Write a security announcement for the official Sylius blog about the vulnerability. This post should contain the following information:
    - a title that always include the "Security release" string;
    - a description of the vulnerability;
    - the affected versions;
    - the possible exploits;
    - how to patch/upgrade/workaround affected applications;
    - credits.
4. Send the patch and the announcement to the reporter for review;
5. Apply the patch to all maintained versions of Sylius;
6. Publish the post on the official Sylius blog;
7. Update the security advisory list (see below).

Note: Releases that include security issues should not be done on Saturday or Sunday, except if the vulnerability has been publicly posted.

Note: While we are working on a patch, please do not reveal the issue publicly.

BDD Methodology

Note: This part of documentation is inspired by the official PHPSpec docs.

Sylius adopted the full-stack BDD methodology for its development processes.

According to Wikipedia:

> "BDD is a software development process based on test-driven development (TDD). Behavior-driven development combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software developers and business analysts with shared tools and a shared process to collaborate on software development, with the aim of delivering software that matters."

Setting up Behat & PHPSpec

To run the entire suite of features and specs, including the ones that depend on external dependencies, Sylius needs to be able to autoload them. By default, they are autoloaded from `vendor/` under the main root directory (see `autoload.php.dist`).

To install them all, use Composer:

Step 1: Get Composer

```
$ curl -s http://getcomposer.org/installer | php
```

Make sure you download `composer.phar` in the same folder where the `composer.json` file is located.

Step 2: Install vendors

```
$ php composer.phar install
```

Note: Note that the script takes some time (several minutes) to finish.

Note: If you don't have `curl` installed, you can also just download the `installer` file manually at http://getcomposer.org/installer. Place this file into your project and then run:

```
$ php installer
$ php composer.phar install
```

Install Selenium2 Download Selenium server 2.38 here.

Create a VirtualHost Add this VirtualHost configuration:

```
<VirtualHost *:80>
    ServerName sylius-test.local

    RewriteEngine On

    DocumentRoot /var/www/sylius/web
    <Directory /var/www/sylius/web>
        Options Indexes FollowSymLinks MultiViews
        AllowOverride None
        Order allow,deny
        allow from all
    </Directory>

    RewriteCond %{DOCUMENT_ROOT}/%{REQUEST_FILENAME} !-f
    RewriteRule ^(.*) %{DOCUMENT_ROOT}/app_test.php [QSA,L]

    ErrorLog ${APACHE_LOG_DIR}/sylius-test-error.log

    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/sylius-test-access.log combined

</VirtualHost>
```

Update your `/etc/hosts` file to include the VirtualHost hostname:

```
127.0.0.1   sylius-test.local
```

Additionally, copy `behat.yml.dist` to `behat.yml`, edit base_url parameter to match your host:

```
default:
    ...
    extensions:
        Behat\MinkExtension\Extension:
            ...
            base_url: http://sylius-test.local/app_test.php/
```

Behat

We use Behat for StoryBDD and you should always write new scenarios when adding a feature, or update existing stories to adapt Sylius to business requirements changes.

Sylius is an open source project, so the **client** is not clearly visible at first look. But they are here - the Sylius users. We have our needs and Behat helps us understand and satisfy these needs.

Note: To be written.

You can launch Selenium by issuing the following command:

```
$ java -jar selenium-server-standalone-2.33.0.jar
```

Configure behat for Selenium:

```
default:
    ...
    extensions:
        Behat\MinkExtension\Extension:
            default_session: selenium2
            browser_name: firefox
            base_url: http://sylius-test.local/app_test.php
            selenium2:
                capabilities: { "browser": "firefox", "version": "28"}
```

Run your scenario using the `behat` console:

```
$ bin/behat
```

PHPSpec

PHPSpec is a PHP toolset to drive emergent design by specification. It is not really a testing tool, but a design instrument, which helps structuring the objects and how they work together.

Sylius approach is to always describe the behavior of the next object you are about to implement.

As an example, we'll write a service, which updates product prices based on an external API. To initialize a new spec, use the `desc` command.

We just need to tell **PHPSpec** we will be working on the *PriceUpdater* class.

```
$ bin/phpspec desc "Sylius\Bundle\CoreBundle\Pricing\PriceUpdater"
Specification for PriceUpdater created in spec.
```

What have we just done? **PHPSpec** has created the spec for us. You can navigate to the spec folder and see the spec there:

```
<?php

namespace spec\Sylius\Bundle\CoreBundle\Pricing;

use PhpSpec\ObjectBehavior;
use Prophecy\Argument;

class PriceUpdaterSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this->shouldHaveType('Sylius\Bundle\CoreBundle\Pricing\PriceUpdater');
    }
}
```

The object behavior is made of examples. Examples are encased in public methods, started with `it_`. or `its_`.

PHPSpec searches for such methods in your specification to run. Why underscores for example names? `just_because_its_much_easier_to_read` than `someLongCamelCasingLikeThat`.

Now, let's write first example which will update the products price:

```
<?php

namespace spec\Sylius\Bundle\CoreBundle\Pricing;

use Acme\ApiClient;
use PhpSpec\ObjectBehavior;
use Prophecy\Argument;
use Sylius\Bundle\CoreBundle\Model\ProductInterface;

class PriceUpdaterSpec extends ObjectBehavior
{
    function let(ApiClient $api)
    {
        $this->beConstructedWith($api);
    }

    function it_updates_product_price_through_api($api, ProductInterface $product)
    {
        $product->getSku()->shouldBeCalled()->willReturn('TES-12-A-1090');
        $api->getCurrentProductPrice('TES-12-A-1090')->shouldBeCalled()->willReturn(1545);
        $product->setPrice(1545)->shouldBeCalled();

        $this->updatePrice($product);
    }
}
```

The example looks clear and simple, the `PriceUpdater` service should obtain the SKU of the product, call the external API and update products price accordingly.

Try running the example by using the following command:

```
$ bin/phpspec run

> spec\Sylius\Bundle\CoreBundle\Pricing\PriceUpdater
```

```
   it updates product price through api
      Class PriceUpdater does not exists.

          Do you want me to create it for you? [Y/n]
```

Once the class is created and you run the command again, PHPSpec will ask if it should create the method as well. Start implementing the very initial version of the price updater.

```php
<?php

namespace Sylius\Bundle\CoreBundle\Pricing;

use Sylius\Bundle\CoreBundle\Model\ProductInterface;
use Acme\ApiClient;

class PriceUpdater
{
    private $api;

    public function __construct(ApiClient $api)
    {
        $this->api = $api;
    }

    public function updatePrice(ProductInterface $product)
    {
        $price = $this->api->getCurrentProductPrice($product->getSku());
        $product->setPrice($price);
    }
}
```

Done! If you run PHPSpec again, you should see the following output:

```
$ bin/phpspec run

> spec\Sylius\Bundle\CoreBundle\Pricing\PriceUpdater

   it updates product price through api

1 examples (1 passed)
223ms
```

This example is greatly simplified, in order to illustrate how we work. There should be few more examples, which cover errors, API exceptions and other edge-cases.

Few tips & rules to follow when working with PHPSpec & Sylius:

- RED is good, add or fix the code to make it green;
- RED-GREEN-REFACTOR is our rule;
- All specs must pass;
- When writing examples, **describe** the behavior of the object in present tense;
- Omit the `public` keyword;
- Use underscores (_) in the examples;
- Use type hinting to mock and stub classes;
- If your specification is getting too complex, the design is wrong, try decoupling a bit more;

- If you cannot describe something easily, probably you should not be doing it that way.

Happy coding!

Coding Standards

When contributing code to Sylius, you must follow its coding standards.

Sylius follows the standards defined in the PSR-0, PSR-1 and PSR-2 documents.

Here is a short example containing most features described below:

```php
<?php

/*
 * This file is part of the Sylius package.
 *
 * (c) Paweł Jędrzejewski
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Acme;

/**
 * Coding standards demonstration.
 */
class FooBar
{
    const SOME_CONST = 42;

    private $fooBar;

    /**
     * @param string $dummy Some argument description
     */
    public function __construct($dummy)
    {
        $this->fooBar = $this->transformText($dummy);
    }

    /**
     * @param string $dummy Some argument description
     * @param array  $options
     *
     * @return string|null Transformed input
     *
     * @throws \RuntimeException
     */
    private function transformText($dummy, array $options = array())
    {
        $mergedOptions = array_merge(
            array(
                'some_default'    => 'values',
                'another_default' => 'more values',
            ),
            $options
        );
```

```
        if (true === $dummy) {
            return;
        }

        if ('string' === $dummy) {
            if ('values' === $mergedOptions['some_default']) {
                return substr($dummy, 0, 5);
            }

            return ucwords($dummy);
        }

        throw new \RuntimeException(sprintf('Unrecognized dummy option "%s"', $dummy));
    }
}
```

Structure

- Add a single space after each comma delimiter;
- Add a single space around operators (==, &&, ...);
- Add a comma after each array item in a multi-line array, even after the last one;
- Add a blank line before `return` statements, unless the return is alone inside a statement-group (like an `if` statement);
- Use braces to indicate control structure body regardless of the number of statements it contains;
- 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;
- Use parentheses when instantiating classes regardless of the number of arguments the constructor has;
- Exception message strings should be concatenated using **:phpfunction:‘sprintf‘**.

Naming Conventions

- Use camelCase, not underscores, for variable, function and method names, arguments;
- Use underscores for option names and parameter names;
- Use namespaces for all classes;
- Prefix abstract classes with `Abstract`.
- Suffix interfaces with `Interface`;
- Suffix traits with `Trait`;
- Suffix exceptions with `Exception`;
- Use alphanumeric characters and underscores for file names;
- Don’t forget to look at the more verbose *Conventions* document for more subjective naming considerations.

Service Naming Conventions

- A service name contains groups, separated by dots;
- All Sylius services use `sylius` as first group;
- Use lowercase letters for service and parameter names;
- A group name uses the underscore notation;
- Each service has a corresponding parameter containing the class name, following the `service_name.class` convention.

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

- Sylius 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

This document describes coding standards and conventions used in the Sylius codebase to make it more consistent and predictable.

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, the naming convention is used 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, cannot add new elements. If an unrecognized key is passed to "replaceXXX" it must throw an exception.

Deprecations

Warning: Sylius is the pre-alpha development stage. We release minor version before every larger change, but be prepared for BC breaks to happen until 1.0.0 release.

From time to time, some classes and/or methods are deprecated in the framework; that happens when a feature implementation cannot be changed because of backward compatibility issues, but we still want to propose a "better" alternative. In that case, the old implementation can simply be **deprecated**.

A feature is marked as deprecated by adding a `@deprecated` phpdoc to relevant classes, methods, properties, ...:

```
/**
 * @deprecated Deprecated since version 1.X, to be removed in 1.Y. Use XXX instead.
 */
```

The deprecation message should indicate the version when the class/method was deprecated, the version when it will be removed, and whenever possible, how the feature was replaced.

A PHP `E_USER_DEPRECATED` error must also be triggered to help people with the migration starting one or two minor versions before the version where the feature will be removed (depending on the criticality of the removal):

```
trigger_error(
    'XXX() is deprecated since version 2.X and will be removed in 2.Y. Use XXX instead.',
    E_USER_DEPRECATED
);
```

Git

This document explains some conventions and specificities in the way we manage the Sylius code with Git.

Pull Requests

Whenever a pull request is merged, all the information contained in the pull request is saved in the repository.

You can easily spot pull request merges as the commit message always follows this pattern:

```
merged branch USER_NAME/BRANCH_NAME (PR #1111)
```

The PR reference allows you to have a look at the original pull request on GitHub: https://github.com/Sylius/Sylius/pull/1111. Often, this can help understand what the changes were about and the reasoning behind the changes.

Sylius License

Sylius 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

6.1.2 Community

Support

We have very friendly community which provides support for all Sylius users seeking help!

IRC Channels

There are 2 channels available on Freenode IRC network, where you can meet other Sylius developers, ask for help or discuss ideas.

Users channel Channel **#sylius** is for Sylius users - a place where you should seek help and advices. Usually you can meet there several people eager to provide your with community support and answer your questions.

We invite everyone interested in using Sylius to this room, we warmly welcome new people in our growing community!

Developers channel **sylius-dev** is the channel where you'll most likely meet Sylius core team and contributors. It is the right place to discuss about Pull Requests, ideas and architecture. It will be perfect starting point if you want to contribute to the project or chat about concept you'd like to introduce at Sylius.

Now the best part! You do not need to be a Symfony2 guru to help! There is plenty of work which can be done by people starting with Symfony, and to us, every contribution is invaluable!

How to connect You should pick a nice username and connect to `irc.freenode.org` via XChat or any other IRC client. You can also use web client.

StackOverflow.com

We encourage asking Sylius related questions on the stackoverflow.com platform. Be sure to tag them with **sylius** tag - it will make it easier to find for people who can answer it.

To view all Sylius related questions - visit this link. You can also search for phrase.

Statistics

You can always check sylius.org/community to see an overview of our community at work!

Below, you can find more useful links:

- Sylius Pull Requests - pull requests
- Most recent patches - commits
- GitHub issues - issues
- Symfony2 bundles index - knpbundles.com

6.1.3 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 Sylius documentation is hosted on GitHub:

```
https://github.com/Sylius/Sylius-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/Sylius-Docs.git
```

The `master` branch holds the documentation for the development branch of the code.

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.

Creating a Pull Request Following the example, the pull request will default to be between your `improving_foo_and_bar` branch and the `Sylius-Docs` `master` branch.

GitHub covers the topic of pull requests in detail.

Note: The Sylius documentation is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported *License*.

You can also prefix the title of your pull request in a few cases:

- `[WIP]` (Work in Progress) is used when you are not yet finished with your pull request, but you would like it to be reviewed. The pull request won't be merged until you say it is ready.
- `[WCM]` (Waiting Code Merge) is used when you're documenting a new feature or change that hasn't been accepted yet into the core code. The pull request will not be merged until it is merged in the core code (or closed if the change is rejected).

Pull Request Format Unless you're fixing some minor typos, the pull request description **must** include the following checklist to ensure that contributions may be reviewed without needless feedback loops and that your contributions can be included into the documentation as quickly as possible:

```
| Q             | A
| ------------- | ---
| Doc fix?      | [yes|no]
| New docs?     | [yes|no] (PR # on Sylius/Sylius if applicable)
| Fixed tickets | [comma separated list of tickets fixed by the PR]
```

An example submission could now look as follows:

```
| Q             | A
| ------------- | ---
| Doc fix?      | yes
| New docs?     | yes (Sylius/Sylius#1250)
| Fixed tickets | #1075
```

Tip: Online documentation is rebuilt on every code-push to github.

Documenting new Features or Behavior Changes

If you're documenting a brand new feature or a change that's been made in Sylius, you should precede your description of the change with a `.. versionadded:: 1.X` tag and a short description:

```
.. versionadded:: 1.1
    The ``getProductDiscount`` method was introduced in Sylius 1.1.
```

Standards

All documentation in the Sylius Documentation should follow *the documentation standards*.

Reporting an Issue

The most easy contribution you can make is reporting issues: a typo, a grammar mistake, a bug in a code example, a missing explanation, and so on.

Steps:

- Submit new issue in the GitHub tracker;
- *(optional)* Submit a patch.

Translating

Read the dedicated `document`.

Documentation Format

The Sylius 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 Sylius documents or by reading the reStructuredText Primer on the Sphinx website.

If you are familiar with Markdown, be careful as things are 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
html+php	PHP
ini	INI
php-annotations	Annotations

Adding Links To add links to other pages in the documents use the following syntax:

```
:doc:'/path/to/page'
```

Using the path and filename of the page without the extension, for example:

```
:doc:'/book/architecture'
```

```
:doc:'/bundles/SyliusAddressingBundle/installation'
```

The link text will be the main heading of the document linked to. You can also specify alternative text for the link:

```
:doc:'Simple CRUD </bundles/SyliusResourceBundle/installation>'
```

You can also add links to the API documentation:

```
:namespace:'Sylius\\Bundle\\CoreBundle'
```

```
:class:'Sylius\\Bundle\\CoreBundle\\Model\\Order'
```

```
:method:'Sylius\\Bundle\\AddressingBundle\\Matcher\\ZoneMatcher::match'
```

and to the PHP documentation:

```
:phpclass:'SimpleXMLElement'
```

```
:phpmethod:'DateTime::createFromFormat'
```

```
:phpfunction:'iterator_to_array'
```

Testing Documentation To test documentation before a commit:

- Install Sphinx;
- Run the Sphinx quick setup;
- Install the Sphinx extensions (see below);
- Run `make html` and view the generated HTML in the `build` directory.

Installing the Sphinx extensions

- Download the extension from the source repository
- Copy the `sensio` directory 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'))

# adding PhpLexer
from sphinx.highlighting import lexers
from pygments.lexers.web import PhpLexer

# ...
# add the extensions to the list of extensions
extensions = [..., 'sensio.sphinx.refinclude', 'sensio.sphinx.configurationblock', 'sensio.sphinx.php

# enable highlighting for PHP code not between ''<?php ... ?>'' by default
lexers['php'] = PhpLexer(startinline=True)
```

```
lexers['php-annotations'] = PhpLexer(startinline=True)
lexers['php-standalone'] = PhpLexer(startinline=True)
lexers['php-symfony'] = PhpLexer(startinline=True)

# use PHP as the primary domain
primary_domain = 'php'

# set URL for API links
api_url = 'http://api.sylius.org/master/%s'
```

Documentation Standards

In order to help the reader as much as possible and to create code examples that look and feel familiar, you should follow these standards.

Sphinx

- The following characters are chosen for different heading levels: level 1 is =, level 2 −, level 3 ~, level 4 `.` and level 5 `"`;
- Each line should break approximately after the first word that crosses the 72nd character (so most lines end up being 72-78 characters);
- The `::` shorthand is *preferred* over `.. code-block:: php` to begin a PHP code block (read the Sphinx documentation to see when you should use the shorthand);
- Inline hyperlinks are **not** used. Separate the link and their target definition, which you add on the bottom of the page;
- Inline markup should be closed on the same line as the open-string;

Example

```
Example
=======

When you are working on the docs, you should follow the
'Sylius Documentation'_ standards.

Level 2
-------

A PHP example would be::

    echo 'Hello World';

Level 3
~~~~~~~

.. code-block:: php

    echo 'You cannot use the :: shortcut here';

.. _'Sylius Documentation': http://docs.sylius.org/en/latest/contributing/documentation/standards.htr
```

Code Examples

- The code follows the *Sylius Coding Standards* as well as the Twig Coding Standards;
- To avoid horizontal scrolling on code blocks, we prefer to break a line correctly if it crosses the 85th character;
- When you fold one or more lines of code, place `...` in a comment at the point of the fold. These comments are: `// ...` (php), `# ...` (yaml/bash), `{# ... #}` (twig), `<!-- ... -->` (xml/html), `; ...` (ini), `...` (text);
- When you fold a part of a line, e.g. a variable value, put `...` (without comment) at the place of the fold;
- Description of the folded code: (optional) If you fold several lines: the description of the fold can be placed after the `...` If you fold only part of a line: the description can be placed before the line;
- If useful to the reader, a PHP code example should start with the namespace declaration;
- When referencing classes, be sure to show the `use` statements at the top of your code block. You don't need to show *all* `use` statements in every example, just show what is actually being used in the code block;
- If useful, a `codeblock` should begin with a comment containing the filename of the file in the code block. Don't place a blank line after this comment, unless the next line is also a comment;
- You should put a `$` in front of every bash line.

Formats Configuration examples should show recommended formats using *configuration blocks*. The recommended formats (and their orders) are:

- **Configuration** (including services and routing): YAML
- **Validation**: XML
- **Doctrine Mapping**: XML

Example

```
// src/Foo/Bar.php
namespace Foo;

use Acme\Demo\Cat;
// ...

class Bar
{
    // ...

    public function foo($bar)
    {
        // set foo with a value of bar
        $foo = ...;

        $cat = new Cat($foo);

        // ... check if $bar has the correct value

        return $cat->baz($bar, ...);
    }
}
```

Caution: In YAML you should put a space after `{` and before `}` (e.g. `{ _controller: ... }`), but this should not be done in Twig (e.g. `{'hello' : 'value'}`).

Language Standards

- For sections, use the following capitalization rules: Capitalization of the first word, and all other words, except for closed-class words:

  The Vitamins are in my Fresh California Raisins

- Do not use Serial (Oxford) Commas;
- You should use a form of *you* instead of *we* (i.e. avoid the first person point of view: use the second instead);
- When referencing a hypothetical person, such as "a user with a session cookie", gender-neutral pronouns (they/their/them) should be used. For example, instead of:
  - he or she, use they
  - him or her, use them
  - his or her, use their
  - his or hers, use theirs
  - himself or herself, use themselves

Sylius Documentation License

This is a human-readable summary of the Legal Code (the full license).

- **Code**
  - *Bugs*
  - *Patches*
  - *Security*
  - *Behavior Driven Development*
  - *Coding Standards*
  - *Code Conventions*
  - *Git*
  - *License*
- **Documentation**
  - *Overview*
  - *Format*
  - *Documentation Standards*
  - *License*
- **Community**
  - *Support*
  - *Statistics*
- **Code**
  - *Bugs*
  - *Patches*
  - *Security*
  - *Behavior Driven Development*
  - *Coding Standards*
  - *Code Conventions*
  - *Git*
  - *License*
- **Documentation**
  - *Overview*
  - *Format*
  - *Documentation Standards*
  - *License*
- **Community**
  - *Support*
  - *Statistics*