# Brick Documentation

***Release 1ece963***

**Henrik Bjrnskov**

**Sep 27, 2017**

# Contents

Brick is a collection of service providers for Silex. It can be used all together or just mix and match what you need.

CHAPTER 1

# Features

- Easy configuration files by using Tacker and `TackerServiceProvider`.
- Advanced routing with caching by using `RoutingServiceProvider` and Symfony Routing.
- Custom error pages by using `ExceptionServiceProvider`.
- Access to Pimple from inside your Controllers.
- All of the above without removing or disabling any of the normal shortcuts and conventions Silex gives you.

CHAPTER 2

# Documentation

As previously noted Brick is a collection of service providers for Silex or any other project that uses the `silex/api` package. Some of the service providers have dependencies on internal Silex services which are easy to add if used outside of `silex/silex`.

```
<?php

(new Silex\Application)->register(new TackerServiceProvider);

// or with Pimple
(new TackerServiceProvider)->register($pimple = new Pimple);
```

**Note:** This documentation uses a shortcut form or creating object that was introduced in php 5.5.0. Depending on how you are using Brick there might be a better official method. Silex recommend instantiating the application first and then registering service providers.

## Configuration

Configuration are done with Tacker. Tacker provides caching, normalization and support for `php`, `xml`, `yml` and `json` files. For more information on the internals check out its documentation.

To use the service provider add it to your composer file and register it with your application.

```
{
    "require" : {
        "flint/tacker" : "~1.0"
    }
}
```

```
<?php
```

```php
use Brick\TackerServiceProvider;

(new TackerServiceProvider)->register($pimple = new Pimple);

$app['tacker.configurator']->configure($app, 'config.json');
```

By default it will check to see if a `root_dir` parameter exists and will add that to its search paths. Otherwise it can be configured with `tacker.options`.

```php
<?php

// Shows the default values, if root_dir or debug are not enabled wit
// if $app['root_dir'] is set paths will default to array($app['root_dir'])
// if $app['debug'] is set, debug will default to $app['debug']
$app['tacker.options'] = [
    'paths'     => [],
    'cache_dir' => null,
    'debug'     => true,
];
```

**Note:** When using caching it is important that `cache_dir` option in `tacker.options` have been set before loading a configuration file. Otherwise it will not be cached.

## Routing

In order to squeeze more performance out of our application `RoutingServiceProvider` replaces the normal `url_matcher` with the full Router from Symfony. This is done by keeping backwards compatibility and it is still possible to do anything you did before, such as adding easy endpoints.

```php
<?php

use Brick\Provider\RoutingServiceProvider;

$app = new \Silex\Application;
$app->register(new RoutingServiceProvider);

$app->get('/path', function () { });
```

The service provider have some configuration options that can and should be configured. Just as with `TackerServiceProvider` Brick provides some sensible defaults.

```php
<?php

// Shows the default values, if root_dir or debug are not enabled wit
// if $app['root_dir'] is set paths will default to array($app['root_dir'])
// if $app['debug'] is set, debug will default to $app['debug']
$app['routing.options'] = [
    'resource'  => '/path/to/my/routing.xml',
    'paths'     => array(),
    'cache_dir' => null,
    'debug'     => true,
];
```

Not all options must be configured. Only `cache_dir` and `resource` are recommended to use.

**Note:** Because the service provider overwrites the normal `url_generator` service it is incompatible with `UrlGeneratorServiceProvider` which is okay as the router provides the same functionality.

**Note:** **Tip**: Add Twig and its service provider and get automatically access to `url` and `path` from within your templates as you know and love from Symfony

## Custom Error Pages

`ExceptionServiceProvider` adds support for custom error pages that is rendered with Twig. Because of this Twig is a required dependency and can be added to composer with:

```
{
    "require" : {
        "twig/twig" : "~1.8"
    }
}
```

After a quick `composer update twig/twig` the service provider can be added as any other.

```
<?php

use Brick\ExceptionServiceProvider;
use Silex\Provider\TwigServiceProvider;

(new ExceptionServiceProvider)->register($pimple = new Pimple);
(new TwigServiceProvider)->register($pimple);
```

The service provider works by overriding the default exception listener registered. This is only done if `twig` is present and the application runs with `debug` set to `false`.

When looking for a template to render it looks for a template. The template must be loadable from within your `twig.path` setting.

It will look through theese types of templates and return the first found.

1. `Exception/error{statusCode}.{format}.twig` where `{statusCode}` and `{format}` is taken from the current request.
2. `Exception/error.{format}.twig` where `{format}` is taken from the current request.
3. `Exception/error.html.twig` as a fallback.

This is the same lookup that is done in Symfony.

**Note:** When developing theese error pages it can be useful to view them in the dev environment. With a controller and a simple trick, this can easily be done.

```
<?php

namespace My\Controller;

use Brick\Controller\ExceptionController;
```

```
use Symfony\Component\Debug\Exception\FlattenException;
use Symfony\Component\HttpFoundation\Request;

class ErrorPageController
{
    public function __construct(ExceptionController $controller)
    {
        $this->controller = $controller;
    }

    public function __invoke(Request $request, $statusCode)
    {
        $exception = new FlattenException(new Exception(), $statusCode);

        return $controller($request, $exception);
    }
}
```

Now add the above controller to your application:

```
<?php

// $app is an application
$app->get('_error/{$statusCode}', new ErrorPageController($app['exception_controller
↪']));
```

## Puli

Brick comes with experimental support for Puli, which helps manage resources in composer packages.

```
<?php

use Brick\Provider\PuliServiceProvider;

(new PuliServiceProvider)->register($app);
```

If a `twig` service exists it will register `PuliTemplateLoader` and its extension if `puli/url-generator` package is present.

## All in One

All of the above is pretty cool. And using it all together without registering a lot of service providers would be even cooler. For that exact reason is why Brick ships with a `Application` with all of the above service providers pre-registered.

```
<?php

$app = new Brick\Application(['root_dir' => '/path', 'debug' => false]);
$app->configure('config.json');
```

**Warning:** Remember to register `TwigServiceProvider` if you want to have custom error pages.