# The Silex Book

*generated on July 3, 2012*

# Contents at a Glance

Chapter 1

# Introduction

Silex is a PHP microframework for PHP 5.3. It is built on the shoulders of Symfony2 and Pimple and also inspired by sinatra.

A microframework provides the guts for building simple single-file apps. Silex aims to be:

- *Concise*: Silex exposes an intuitive and concise API that is fun to use.
- *Extensible*: Silex has an extension system based around the Pimple micro service-container that makes it even easier to tie in third party libraries.
- *Testable*: Silex uses Symfony2's HttpKernel which abstracts request and response. This makes it very easy to test apps and the framework itself. It also respects the HTTP specification and encourages its proper use.

In a nutshell, you define controllers and map them to routes, all in one step.

**Let's go!**:

Listing 1-1

```
// web/index.php

require_once __DIR__.'/../vendor/autoload.php';

$app = new Silex\Application();

$app->get('/hello/{name}', function ($name) use ($app) {
    return 'Hello '.$app->escape($name);
});

$app->run();
```

All that is needed to get access to the Framework is to include the autoloader.

Next we define a route to `/hello/{name}` that matches for `GET` requests. When the route matches, the function is executed and the return value is sent back to the client.

Finally, the app is run. Visit `/hello/world` to see the result. It's really that easy!

Installing Silex is as easy as it can get. *Download*[1] the archive file, extract it, and you're done!

1. `http://silex.sensiolabs.org/download`

# Chapter 2
# Usage

This chapter describes how to use Silex.

## Installation

If you want to get started fast, *download*[1] Silex as an archive and extract it, you should have the following directory structure:

Listing 2-1

```
├── composer.json
├── composer.lock
├── vendor
│   └── ...
└── web
    └── index.php
```

If you want more flexibility, use Composer instead. Create a `composer.json`:

Listing 2-2

```
{
    "require": {
        "silex/silex": "1.0.*"
    }
}
```

And run Composer to install Silex and all its dependencies:

Listing 2-3

```
$ curl -s http://getcomposer.org/installer | php
$ php composer.phar install
```

## Upgrading

Upgrading Silex to the latest version is as easy as running the `update` command:

1. `http://silex.sensiolabs.org/download`

Listing 2-4

```
$ php composer.phar update
```

## Bootstrap

To bootstrap Silex, all you need to do is require the `vendor/autoload.php` file and create an instance of `Silex\Application`. After your controller definitions, call the `run` method on your application:

Listing 2-5

```
// web/index.php

require_once __DIR__.'/../vendor/autoload.php';

$app = new Silex\Application();

// definitions

$app->run();
```

Then, you have to configure your web server (read the dedicated chapter for more information).

When developing a website, you might want to turn on the debug mode to ease debugging:

Listing 2-6

```
$app['debug'] = true;
```

If your application is hosted behind a reverse proxy and you want Silex to trust the `X-Forwarded-For*` headers, you will need to run your application like this:

Listing 2-7

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

Request::trustProxyData();
$app->run();
```

## Routing

In Silex you define a route and the controller that is called when that route is matched.

A route pattern consists of:

- *Pattern*: The route pattern defines a path that points to a resource. The pattern can include variable parts and you are able to set RegExp requirements for them.
- *Method*: One of the following HTTP methods: `GET`, `POST`, `PUT` `DELETE`. This describes the interaction with the resource. Commonly only `GET` and `POST` are used, but it is possible to use the others as well.

The controller is defined using a closure like this:

Listing 2-8

```
function () {
    // do something
}
```

Closures are anonymous functions that may import state from outside of their definition. This is different from globals, because the outer state does not have to be global. For instance, you could define a closure in a function and import local variables of that function.

Closures that do not import scope are referred to as lambdas. Because in PHP all anonymous functions are instances of the `Closure` class, we will not make a distinction here.

The return value of the closure becomes the content of the page.

There is also an alternate way for defining controllers using a class method. The syntax for that is `ClassName::methodName`. Static methods are also possible.

## Example GET route

Here is an example definition of a `GET` route:

*Listing 2-9*

```
$blogPosts = array(
    1 => array(
        'date'      => '2011-03-29',
        'author'    => 'igorw',
        'title'     => 'Using Silex',
        'body'      => '...',
    ),
);

$app->get('/blog', function () use ($blogPosts) {
    $output = '';
    foreach ($blogPosts as $post) {
        $output .= $post['title'];
        $output .= '<br />';
    }

    return $output;
});
```

Visiting `/blog` will return a list of blog post titles. The `use` statement means something different in this context. It tells the closure to import the $blogPosts variable from the outer scope. This allows you to use it from within the closure.

## Dynamic routing

Now, you can create another controller for viewing individual blog posts:

*Listing 2-10*

```
$app->get('/blog/show/{id}', function (Silex\Application $app, $id) use ($blogPosts) {
    if (!isset($blogPosts[$id])) {
        $app->abort(404, "Post $id does not exist.");
    }

    $post = $blogPosts[$id];

    return  "<h1>{$post['title']}</h1>".
            "<p>{$post['body']}</p>";
});
```

This route definition has a variable `{id}` part which is passed to the closure.

When the post does not exist, we are using `abort()` to stop the request early. It actually throws an exception, which we will see how to handle later on.

## Example POST route

POST routes signify the creation of a resource. An example for this is a feedback form. We will use the `mail` function to send an e-mail:

Listing 2-11
```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$app->post('/feedback', function (Request $request) {
    $message = $request->get('message');
    mail('feedback@yoursite.com', '[YourSite] Feedback', $message);

    return new Response('Thank you for your feedback!', 201);
});
```

It is pretty straightforward.

There is a *SwiftmailerServiceProvider* included that you can use instead of `mail()`.

The current `request` is automatically injected by Silex to the Closure thanks to the type hinting. It is an instance of *Request*[2], so you can fetch variables using the request `get` method.

Instead of returning a string we are returning an instance of *Response*[3]. This allows setting an HTTP status code, in this case it is set to `201 Created`.

Silex always uses a `Response` internally, it converts strings to responses with status code `200 Ok`.

## Other methods

You can create controllers for most HTTP methods. Just call one of these methods on your application: `get`, `post`, `put`, `delete`. You can also call `match`, which will match all methods:

Listing 2-12
```
$app->match('/blog', function () {
    ...
});
```

You can then restrict the allowed methods via the `method` method:

Listing 2-13
```
$app->match('/blog', function () {
    ...
})
->method('PATCH');
```

You can match multiple methods with one controller using regex syntax:

Listing 2-14
```
$app->match('/blog', function () {
    ...
})
->method('PUT|POST');
```

2. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/Request.html`
3. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html`

The order in which the routes are defined is significant. The first matching route will be used, so place more generic routes at the bottom.

## Route variables

As it has been shown before you can define variable parts in a route like this:

Listing 2-15

```
$app->get('/blog/show/{id}', function ($id) {
    ...
});
```

It is also possible to have more than one variable part, just make sure the closure arguments match the names of the variable parts:

Listing 2-16

```
$app->get('/blog/show/{postId}/{commentId}', function ($postId, $commentId) {
    ...
});
```

While it's not suggested, you could also do this (note the switched arguments):

Listing 2-17

```
$app->get('/blog/show/{postId}/{commentId}', function ($commentId, $postId) {
    ...
});
```

You can also ask for the current Request and Application objects:

Listing 2-18

```
$app->get('/blog/show/{id}', function (Application $app, Request $request, $id) {
    ...
});
```

Note for the Application and Request objects, Silex does the injection based on the type hinting and not on the variable name:

Listing 2-19

```
$app->get('/blog/show/{id}', function (Application $foo, Request $bar, $id) {
    ...
});
```

## Route variables converters

Before injecting the route variables into the controller, you can apply some converters:

Listing 2-20

```
$app->get('/user/{id}', function ($id) {
    // ...
})->convert('id', function ($id) { return (int) $id; });
```

This is useful when you want to convert route variables to objects as it allows to reuse the conversion code across different controllers:

Listing 2-21

```
$userProvider = function ($id) {
    return new User($id);
};

$app->get('/user/{user}', function (User $user) {
    // ...
})->convert('user', $userProvider);

$app->get('/user/{user}/edit', function (User $user) {
    // ...
})->convert('user', $userProvider);
```

The converter callback also receives the `Request` as its second argument:

Listing 2-22

```
$callback = function ($post, Request $request) {
    return new Post($request->attributes->get('slug'));
};

$app->get('/blog/{id}/{slug}', function (Post $post) {
    // ...
})->convert('post', $callback);
```

## Requirements

In some cases you may want to only match certain expressions. You can define requirements using regular expressions by calling `assert` on the `Controller` object, which is returned by the routing methods.

The following will make sure the `id` argument is numeric, since `\d+` matches any amount of digits:

Listing 2-23

```
$app->get('/blog/show/{id}', function ($id) {
    ...
})
->assert('id', '\d+');
```

You can also chain these calls:

Listing 2-24

```
$app->get('/blog/show/{postId}/{commentId}', function ($postId, $commentId) {
    ...
})
->assert('postId', '\d+')
->assert('commentId', '\d+');
```

## Default values

You can define a default value for any route variable by calling `value` on the `Controller` object:

Listing 2-25

```
$app->get('/{pageName}', function ($pageName) {
    ...
})
->value('pageName', 'index');
```

This will allow matching `/`, in which case the `pageName` variable will have the value `index`.

## Named routes

Some providers (such as `UrlGeneratorProvider`) can make use of named routes. By default Silex will generate a route name for you, that cannot really be used. You can give a route a name by calling `bind` on the `Controller` object that is returned by the routing methods:

Listing 2-26

```
$app->get('/', function () {
    ...
})
->bind('homepage');
```

```
$app->get('/blog/show/{id}', function ($id) {
    ...
})
->bind('blog_post');
```

It only makes sense to name routes if you use providers that make use of the `RouteCollection`.

## Before, after and finish filters

Silex allows you to run code before, after every request and even after the response has been sent. This happens through `before`, `after` and `finish` filters. All you need to do is pass a closure:

Listing 2-27

```
$app->before(function () {
    // set up
});

$app->after(function () {
    // tear down
});

$app->finish(function () {
    // after response has been sent
});
```

The before filter has access to the current Request, and can short-circuit the whole rendering by returning a Response:

Listing 2-28

```
$app->before(function (Request $request) {
    // redirect the user to the login screen if access to the Resource is protected
    if (...) {
        return new RedirectResponse('/login');
    }
});
```

The after filter has access to the Request and the Response:

Listing 2-29

```
$app->after(function (Request $request, Response $response) {
    // tweak the Response
});
```

The finish filter has access to the Request and the Response:

Listing 2-30

```
$app->finish(function (Request $request, Response $response) {
    // send e-mails ...
});
```

The filters are only run for the "master" Request.

## Route middlewares

Route middlewares are PHP callables which are triggered when their associated route is matched:

- `before` middlewares are fired just before the route callback, but after the application `before` filters;
- `after` middlewares are fired just after the route callback, but before the application `after` filters.

This can be used for a lot of use cases; for instance, here is a simple "anonymous/logged user" check:

*Listing 2-31*

```
$mustBeAnonymous = function (Request $request) use ($app) {
    if ($app['session']->has('userId')) {
        return $app->redirect('/user/logout');
    }
};

$mustBeLogged = function (Request $request) use ($app) {
    if (!$app['session']->has('userId')) {
        return $app->redirect('/user/login');
    }
};

$app->get('/user/subscribe', function () {
    ...
})
->before($mustBeAnonymous);

$app->get('/user/login', function () {
    ...
})
->before($mustBeAnonymous);

$app->get('/user/my-profile', function () {
    ...
})
->before($mustBeLogged);
```

The `before` and `after` methods can be called several times for a given route, in which case they are triggered in the same order as you added them to the route.

For convenience, the `before` middlewares are called with the current `Request` instance as an argument and the `after` middlewares are called with the current `Request` and `Response` instance as arguments.

If any of the before middlewares returns a Symfony HTTP Response, it will short-circuit the whole rendering: the next middlewares won't be run, neither the route callback. You can also redirect to another page by returning a redirect response, which you can create by calling the Application `redirect` method.

If a before middleware does not return a Symfony HTTP Response or `null`, a `RuntimeException` is thrown.

## Global Configuration

If a controller setting must be applied to all controllers (a converter, a middleware, a requirement, or a default value), you can configure it on `$app['controllers']`, which holds all application controllers:

```php
$app['controllers']
    ->value('id', '1')
    ->assert('id', '\d+')
    ->requireHttps()
    ->method('get')
    ->convert('id', function () { // ... })
    ->before(function () { // ... })
;
```
*Listing 2-32*

These settings are applied to already registered controllers and they become the defaults for new controllers.

The global configuration does not apply to controller providers you might mount as they have their own global configuration (see the Modularity paragraph below).

## Error handlers

If some part of your code throws an exception you will want to display some kind of error page to the user. This is what error handlers do. You can also use them to do additional things, such as logging.

To register an error handler, pass a closure to the `error` method which takes an `Exception` argument and returns a response:

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

$app->error(function (\Exception $e, $code) {
    return new Response('We are sorry, but something went terribly wrong.', $code);
});
```
*Listing 2-33*

You can also check for specific errors by using the `$code` argument, and handle them differently:

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

$app->error(function (\Exception $e, $code) {
    switch ($code) {
        case 404:
            $message = 'The requested page could not be found.';
            break;
        default:
            $message = 'We are sorry, but something went terribly wrong.';
    }

    return new Response($message, $code);
});
```
*Listing 2-34*

You can restrict an error handler to only handle some Exception classes by setting a more specific type hint for the Closure argument:

```php
$app->error(function (\LogicException $e, $code) {
    // this handler will only \LogicException exceptions
    // and exceptions that extends \LogicException
});
```
*Listing 2-35*

If you want to set up logging you can use a separate error handler for that. Just make sure you register it before the response error handlers, because once a response is returned, the following handlers are ignored.

Silex ships with a provider for *Monolog*[4] which handles logging of errors. Check out the *Providers* chapter for details.

Silex comes with a default error handler that displays a detailed error message with the stack trace when **debug** is true, and a simple error message otherwise. Error handlers registered via the `error()` method always take precedence but you can keep the nice error messages when debug is turned on like this:

*Listing 2-36*
```
use Symfony\Component\HttpFoundation\Response;

$app->error(function (\Exception $e, $code) use ($app) {
    if ($app['debug']) {
        return;
    }

    // logic to handle the error and return a Response
});
```

The error handlers are also called when you use `abort` to abort a request early:

*Listing 2-37*
```
$app->get('/blog/show/{id}', function (Silex\Application $app, $id) use ($blogPosts) {
    if (!isset($blogPosts[$id])) {
        $app->abort(404, "Post $id does not exist.");
    }

    return new Response(...);
});
```

## Redirects

You can redirect to another page by returning a redirect response, which you can create by calling the `redirect` method:

*Listing 2-38*
```
$app->get('/', function () use ($app) {
    return $app->redirect('/hello');
});
```

This will redirect from `/` to `/hello`.

## Forwards

When you want to delegate the rendering to another controller, without a round-trip to the browser (as for a redirect), use an internal sub-request:

*Listing 2-39*
```
use Symfony\Component\HttpKernel\HttpKernelInterface;

$app->get('/', function () use ($app) {
    // redirect to /hello
    $subRequest = Request::create('/hello', 'GET');
```

4. `https://github.com/Seldaek/monolog`

```
    return $app->handle($subRequest, HttpKernelInterface::SUB_REQUEST);
});
```

If you are using `UrlGeneratorProvider`, you can also generate the URI:

Listing 2-40

```
$request = Request::create($app['url_generator']->generate('hello'), 'GET');
```

## Modularity

When your application starts to define too many controllers, you might want to group them logically:

Listing 2-41

```
// define controllers for a blog
$blog = $app['controllers_factory'];
$blog->get('/', function () {
    return 'Blog home page';
});
// ...

// define controllers for a forum
$forum = $app['controllers_factory'];
$forum->get('/', function () {
    return 'Forum home page';
});

// define "global" controllers
$app->get('/', function () {
    return 'Main home page';
});

$app->mount('/blog', $blog);
$app->mount('/forum', $forum);
```

`$app['controllers_factory']` is a factory that returns a new instance of `ControllerCollection` when used.

`mount()` prefixes all routes with the given prefix and merges them into the main Application. So, `/` will map to the main home page, `/blog/` to the blog home page, and `/forum/` to the forum home page.

When mounting a route collection under `/blog`, it is not possible to define a route for the `/blog` URL. The shortest possible URL is `/blog/`.

When calling `get()`, `match()`, or any other HTTP methods on the Application, you are in fact calling them on a default instance of `ControllerCollection` (stored in `$app['controllers']`).

Another benefit is the ability to apply settings on a set of controllers very easily. Building on the example from the middleware section, here is how you would secure all controllers for the backend collection:

Listing 2-42

```php
$backend = $app['controllers_factory'];

// ensure that all controllers require logged-in users
$backend->before($mustBeLogged);
```

For a better readability, you can split each controller collection into a separate file:

Listing 2-43

```php
// blog.php
$blog = $app['controllers_factory'];
$blog->get('/', function () { return 'Blog home page'; });

return $blog;

// app.php
$app->mount('/blog', include 'blog.php');
```

Instead of requiring a file, you can also create a `Controller provider`.

## JSON

If you want to return JSON data, you can use the `json` helper method. Simply pass it your data, status code and headers, and it will create a JSON response for you:

Listing 2-44

```php
$app->get('/users/{id}', function ($id) use ($app) {
    $user = getUser($id);

    if (!$user) {
        $error = array('message' => 'The user was not found.');
        return $app->json($error, 404);
    }

    return $app->json($user);
});
```

## Streaming

It's possible to create a streaming response, which is important in cases when you cannot buffer the data being sent:

Listing 2-45

```php
$app->get('/images/{file}', function ($file) use ($app) {
    if (!file_exists(__DIR__.'/images/'.$file)) {
        return $app->abort(404, 'The image was not found.');
    }

    $stream = function () use ($file) {
        readfile($file);
    };

    return $app->stream($stream, 200, array('Content-Type' => 'image/png'));
});
```

If you need to send chunks, make sure you call `ob_flush` and `flush` after every chunk:

Listing 2-46

```php
$stream = function () {
    $fh = fopen('http://www.example.com/', 'rb');
    while (!feof($fh)) {
      echo fread($fh, 1024);
      ob_flush();
      flush();
    }
    fclose($fh);
};
```

## Traits

Silex comes with PHP traits that define shortcut methods.

⚠ You need to use PHP 5.4 or later to benefit from this feature.

Almost all built-in service providers have some corresponding PHP traits. To use them, define your own Application class and include the traits you want:

Listing 2-47

```
use Silex\Application;

class MyApplication extends Application
{
    use Application\TwigTrait;
    use Application\SecurityTrait;
    use Application\FormTrait;
    use Application\UrlGeneratorTrait;
    use Application\SwiftmailerTrait;
    use Application\MonologTrait;
    use Application\TranslationTrait;
}
```

You can also define your own Route class and use some traits:

Listing 2-48

```
use Silex\Route;

class MyRoute extends Route
{
    use Route\SecurityTrait;
}
```

To use your newly defined route, override the `$app['route_class']` setting:

Listing 2-49

```
$app['route_class'] = 'MyRoute';
```

Read each provider chapter to learn more about the added methods.

## Security

Make sure to protect your application against attacks.

### Escaping

When outputting any user input (either route variables GET/POST variables obtained from the request), you will have to make sure to escape it correctly, to prevent Cross-Site-Scripting attacks.

- **Escaping HTML**: PHP provides the `htmlspecialchars` function for this. Silex provides a shortcut `escape` method:

*Listing 2-50*

```
$app->get('/name', function (Silex\Application $app) {
    $name = $app['request']->get('name');
    return "You provided the name {$app->escape($name)}.";
});
```

  If you use the Twig template engine you should use its escaping or even auto-escaping mechanisms.

- **Escaping JSON**: If you want to provide data in JSON format you should use the Silex `json` function:

*Listing 2-51*

```
$app->get('/name.json', function (Silex\Application $app) {
    $name = $app['request']->get('name');
    return $app->json(array('name' => $name));
});
```

# Chapter 3

# Services

Silex is not only a microframework. It is also a micro service container. It does this by extending *Pimple*[1] which provides service goodness in just 44 NCLOC.

## Dependency Injection

You can skip this if you already know what Dependency Injection is.

Dependency Injection is a design pattern where you pass dependencies to services instead of creating them from within the service or relying on globals. This generally leads to code that is decoupled, re-usable, flexible and testable.

Here is an example of a class that takes a `User` object and stores it as a file in JSON format:

*Listing 3-1*

```
class JsonUserPersister
{
    private $basePath;

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

    public function persist(User $user)
    {
        $data = $user->getAttributes();
        $json = json_encode($data);
        $filename = $this->basePath.'/'.$user->id.'.json';
        file_put_contents($filename, $json, LOCK_EX);
    }
}
```

1. `http://pimple.sensiolabs.org`

In this simple example the dependency is the `basePath` property. It is passed to the constructor. This means you can create several independent instances with different base paths. Of course dependencies do not have to be simple strings. More often they are in fact other services.

### Container

A DIC or service container is responsible for creating and storing services. It can recursively create dependencies of the requested services and inject them. It does so lazily, which means a service is only created when you actually need it.

Most containers are quite complex and are configured through XML or YAML files.

Pimple is different.

## Pimple

Pimple is probably the simplest service container out there. It makes strong use of closures and implements the ArrayAccess interface.

We will start off by creating a new instance of Pimple -- and because `Silex\Application` extends `Pimple` all of this applies to Silex as well:

*Listing 3-2*
```
$container = new Pimple();
```

or:

*Listing 3-3*
```
$app = new Silex\Application();
```

### Parameters

You can set parameters (which are usually strings) by setting an array key on the container:

*Listing 3-4*
```
$app['some_parameter'] = 'value';
```

The array key can be anything, by convention periods are used for namespacing:

*Listing 3-5*
```
$app['asset.host'] = 'http://cdn.mysite.com/';
```

Reading parameter values is possible with the same syntax:

*Listing 3-6*
```
echo $app['some_parameter'];
```

### Service definitions

Defining services is no different than defining parameters. You just set an array key on the container to be a closure. However, when you retrieve the service, the closure is executed. This allows for lazy service creation:

*Listing 3-7*
```
$app['some_service'] = function () {
    return new Service();
};
```

And to retrieve the service, use:

*Listing 3-8*
```
$service = $app['some_service'];
```

Every time you call `$app['some_service']`, a new instance of the service is created.

## Shared services

You may want to use the same instance of a service across all of your code. In order to do that you can make a *shared* service:

*Listing 3-9*

```
$app['some_service'] = $app->share(function () {
    return new Service();
});
```

This will create the service on first invocation, and then return the existing instance on any subsequent access.

## Access container from closure

In many cases you will want to access the service container from within a service definition closure. For example when fetching services the current service depends on.

Because of this, the container is passed to the closure as an argument:

*Listing 3-10*

```
$app['some_service'] = function ($app) {
    return new Service($app['some_other_service'], $app['some_service.config']);
};
```

Here you can see an example of Dependency Injection. `some_service` depends on `some_other_service` and takes `some_service.config` as configuration options. The dependency is only created when `some_service` is accessed, and it is possible to replace either of the dependencies by simply overriding those definitions.

This also works for shared services.

## Protected closures

Because the container sees closures as factories for services, it will always execute them when reading them.

In some cases you will however want to store a closure as a parameter, so that you can fetch it and execute it yourself -- with your own arguments.

This is why Pimple allows you to protect your closures from being executed, by using the `protect` method:

*Listing 3-11*

```
$app['closure_parameter'] = $app->protect(function ($a, $b) {
    return $a + $b;
});

// will not execute the closure
$add = $app['closure_parameter'];

// calling it now
echo $add(2, 3);
```

Note that protected closures do not get access to the container.

## Core services

Silex defines a range of services which can be used or replaced. You probably don't want to mess with most of them.

- **request**: Contains the current request object, which is an instance of *Request*[2]. It gives you access to `GET`, `POST` parameters and lots more!

  Example usage:

Listing 3-12

```
$id = $app['request']->get('id');
```

  This is only available when a request is being served, you can only access it from within a controller, before filter, after filter or error handler.

- **routes**: The *RouteCollection*[3] that is used internally. You can add, modify, read routes.
- **controllers**: The `Silex\ControllerCollection` that is used internally. Check the *Internals* chapter for more information.
- **dispatcher**: The *EventDispatcher*[4] that is used internally. It is the core of the Symfony2 system and is used quite a bit by Silex.
- **resolver**: The *ControllerResolver*[5] that is used internally. It takes care of executing the controller with the right arguments.
- **kernel**: The *HttpKernel*[6] that is used internally. The HttpKernel is the heart of Symfony2, it takes a Request as input and returns a Response as output.
- **request_context**: The request context is a simplified representation of the request that is used by the Router and the UrlGenerator.
- **exception_handler**: The Exception handler is the default handler that is used when you don't register one via the `error()` method or if your handler does not return a Response. Disable it with `unset($app['exception_handler'])`.
- **logger**: A *LoggerInterface*[7] instance. By default, logging is disabled as the value is set to `null`. When the Symfony2 Monolog bridge is installed, Monolog is automatically used as the default logger.

All of these Silex core services are shared.

## Core parameters

- **request.http_port** (optional): Allows you to override the default port for non-HTTPS URLs. If the current request is HTTP, it will always use the current port.

  Defaults to 80.

  This parameter can be used by the `UrlGeneratorProvider`.

2. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/Request.html`
3. `http://api.symfony.com/master/Symfony/Component/Routing/RouteCollection.html`
4. `http://api.symfony.com/master/Symfony/Component/EventDispatcher/EventDispatcher.html`
5. `http://api.symfony.com/master/Symfony/Component/HttpKernel/Controller/ControllerResolver.html`
6. `http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpKernel.html`
7. `http://api.symfony.com/master/Symfony/Component/HttpKernel/Log/LoggerInterface.html`

- **request.https_port** (optional): Allows you to override the default port for HTTPS URLs. If the current request is HTTPS, it will always use the current port.

  Defaults to 443.

  This parameter can be used by the `UrlGeneratorProvider`.
- **locale** (optional): The locale of the user. When set before any request handling, it defines the default locale (`en` by default). When a request is being handled, it is automatically set according to the `_locale` request attribute of the current route.
- **debug** (optional): Returns whether or not the application is running in debug mode.

  Defaults to false.
- **charset** (optional): The charset to use for Responses.

  Defaults to UTF-8.

Chapter 4

# Providers

Providers allow the developer to reuse parts of an application into another one. Silex provides two types of providers defined by two interfaces: `ServiceProviderInterface` for services and `ControllerProviderInterface` for controllers.

## Service Providers

### Loading providers

In order to load and use a service provider, you must register it on the application:

Listing 4-1

```
$app = new Silex\Application();

$app->register(new Acme\DatabaseServiceProvider());
```

You can also provide some parameters as a second argument. These will be set **before** the provider is registered:

Listing 4-2

```
$app->register(new Acme\DatabaseServiceProvider(), array(
    'database.dsn'      => 'mysql:host=localhost;dbname=myapp',
    'database.user'     => 'root',
    'database.password' => 'secret_root_password',
));
```

### Conventions

You need to watch out in what order you do certain things when interacting with providers. Just keep to these rules:

- Overriding existing services must occur **after** the provider is registered.

  *Reason: If the services already exist, the provider will overwrite it.*
- You can set parameters any time before the service is accessed.

Make sure to stick to this behavior when creating your own providers.

## Included providers

There are a few provider that you get out of the box. All of these are within the `Silex\Provider` namespace:

- *DoctrineServiceProvider*
- *MonologServiceProvider*
- *SessionServiceProvider*
- *SwiftmailerServiceProvider*
- *TwigServiceProvider*
- *TranslationServiceProvider*
- *UrlGeneratorServiceProvider*
- *ValidatorServiceProvider*
- *HttpCacheServiceProvider*
- *FormServiceProvider*

## Third party providers

Some service providers are developed by the community. Those third-party providers are listed on *Silex' repository wiki*[1].

You are encouraged to share yours.

## Creating a provider

Providers must implement the `Silex\ServiceProviderInterface`:

*Listing 4-3*

```
interface ServiceProviderInterface
{
    function register(Application $app);

    function boot(Application $app);
}
```

This is very straight forward, just create a new class that implements the two methods. In the `register()` method, you can define services on the application which then may make use of other services and parameters. In the `boot()` method, you can configure the application, just before it handles a request.

Here is an example of such a provider:

*Listing 4-4*

```
namespace Acme;

use Silex\Application;
use Silex\ServiceProviderInterface;

class HelloServiceProvider implements ServiceProviderInterface
{
    public function register(Application $app)
    {
        $app['hello'] = $app->protect(function ($name) use ($app) {
            $default = $app['hello.default_name'] ? $app['hello.default_name'] : '';
            $name = $name ?: $default;

            return 'Hello '.$app->escape($name);
        });
    }
```

1. `https://github.com/fabpot/Silex/wiki/Third-Party-ServiceProviders`

```
    public function boot(Application $app)
    {
    }
}
```

This class provides a `hello` service which is a protected closure. It takes a `name` argument and will return `hello.default_name` if no name is given. If the default is also missing, it will use an empty string.

You can now use this provider as follows:

*Listing 4-5*

```
$app = new Silex\Application();

$app->register(new Acme\HelloServiceProvider(), array(
    'hello.default_name' => 'Igor',
));

$app->get('/hello', function () use ($app) {
    $name = $app['request']->get('name');

    return $app['hello']($name);
});
```

In this example we are getting the `name` parameter from the query string, so the request path would have to be `/hello?name=Fabien`.

## Controllers providers

### Loading providers

In order to load and use a controller provider, you must "mount" its controllers under a path:

*Listing 4-6*

```
$app = new Silex\Application();

$app->mount('/blog', new Acme\BlogControllerProvider());
```

All controllers defined by the provider will now be available under the `/blog` path.

### Creating a provider

Providers must implement the `Silex\ControllerProviderInterface`:

*Listing 4-7*

```
interface ControllerProviderInterface
{
    function connect(Application $app);
}
```

Here is an example of such a provider:

*Listing 4-8*

```
namespace Acme;

use Silex\Application;
use Silex\ControllerProviderInterface;
use Silex\ControllerCollection;

class HelloControllerProvider implements ControllerProviderInterface
{
    public function connect(Application $app)
    {
        // creates a new controller based on the default route
        $controllers = $app['controllers_factory'];

        $controllers->get('/', function (Application $app) {
            return $app->redirect('/hello');
        });

        return $controllers;
    }
}
```

The `connect` method must return an instance of `ControllerCollection`. `ControllerCollection` is the class where all controller related methods are defined (like `get`, `post`, `match`, ...).

The `Application` class acts in fact as a proxy for these methods.

You can now use this provider as follows:

Listing 4-9

```
$app = new Silex\Application();

$app->mount('/blog', new Acme\HelloControllerProvider());
```

In this example, the `/blog/` path now references the controller defined in the provider.

You can also define a provider that implements both the service and the controller provider interface and package in the same class the services needed to make your controllers work.

# Chapter 5
# Testing

Because Silex is built on top of Symfony2, it is very easy to write functional tests for your application. Functional tests are automated software tests that ensure that your code is working correctly. They go through the user interface, using a fake browser, and mimic the actions a user would do.

## Why

If you are not familiar with software tests, you may be wondering why you would need this. Every time you make a change to your application, you have to test it. This means going through all the pages and making sure they are still working. Functional tests save you a lot of time, because they enable you to test your application in usually under a second by running a single command.

For more information on functional testing, unit testing, and automated software tests in general, check out *PHPUnit*[1] and Bulat Shakirzyanov's talk on Clean Code.

## PHPUnit

*PHPUnit*[2] is the de-facto standard testing framework for PHP. It was built for writing unit tests, but it can be used for functional tests too. You write tests by creating a new class, that extends the `PHPUnit_Framework_TestCase`. Your test cases are methods prefixed with `test`:

*Listing 5-1*

```php
class ContactFormTest extends PHPUnit_Framework_TestCase
{
    public function testInitialPage()
    {
        ...
    }
}
```

In your test cases, you do assertions on the state of what you are testing. In this case we are testing a contact form, so we would want to assert that the page loaded correctly and contains our form:

1. `https://github.com/sebastianbergmann/phpunit`
2. `https://github.com/sebastianbergmann/phpunit`

```php
public function testInitialPage()
{
    $statusCode = ...
    $pageContent = ...

    $this->assertEquals(200, $statusCode);
    $this->assertContains('Contact us', $pageContent);
    $this->assertContains('<form', $pageContent);
}
```
*Listing 5-2*

Here you see some of the available assertions. There is a full list available in the *Writing Tests for PHPUnit*[3] section of the PHPUnit documentation.

## WebTestCase

Symfony2 provides a WebTestCase class that can be used to write functional tests. The Silex version of this class is `Silex\WebTestCase`, and you can use it by making your test extend it:

```php
use Silex\WebTestCase;

class ContactFormTest extends WebTestCase
{
    ...
}
```
*Listing 5-3*

To make your application testable, you need to make sure you follow "Reusing applications" instructions from *Usage*.

For your WebTestCase, you will have to implement a `createApplication` method, which returns your application. It will probably look like this:

```php
public function createApplication()
{
    return require __DIR__.'/path/to/app.php';
}
```
*Listing 5-4*

Make sure you do **not** use `require_once` here, as this method will be executed before every test.

By default, the application behaves in the same way as when using it from a browser. But when an error occurs, it is sometimes easier to get raw exceptions instead of HTML pages. It is rather simple if you tweak the application configuration in the `createApplication()` method like follows:

```php
public function createApplication()
{
    $app = require __DIR__.'/path/to/app.php';
    $app['debug'] = true;
    unset($app['exception_handler']);

    return $app;
}
```
*Listing 5-5*

3. `http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html`

If your application use sessions, set `session.test` to `true` to simulate sessions:

Listing 5-6
```
public function createApplication()
{
    // ...

    $this->app['session.test'] = true;

    // ...
}
```

The WebTestCase provides a `createClient` method. A client acts as a browser, and allows you to interact with your application. Here's how it works:

Listing 5-7
```
public function testInitialPage()
{
    $client = $this->createClient();
    $crawler = $client->request('GET', '/');

    $this->assertTrue($client->getResponse()->isOk());
    $this->assertCount(1, $crawler->filter('h1:contains("Contact us")'));
    $this->assertCount(1, $crawler->filter('form'));
    ...
}
```

There are several things going on here. You have both a `Client` and a `Crawler`.

You can also access the application through `$this->app`.

## Client

The client represents a browser. It holds your browsing history, cookies and more. The `request` method allows you to make a request to a page on your application.

You can find some documentation for it in the client section of the testing chapter of the Symfony2 documentation.

## Crawler

The crawler allows you to inspect the content of a page. You can filter it using CSS expressions and lots more.

You can find some documentation for it in the crawler section of the testing chapter of the Symfony2 documentation.

## Configuration

The suggested way to configure PHPUnit is to create a `phpunit.xml.dist` file, a `tests` folder and your tests in `tests/YourApp/Tests/YourTest.php`. The `phpunit.xml.dist` file should look like this:

*Listing 5-8*

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false"
         syntaxCheck="false"
>
    <testsuites>
        <testsuite name="YourApp Test Suite">
            <directory>./tests/</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

You can also configure a bootstrap file for autoloading and whitelisting for code coverage reports.

Your `tests/YourApp/Tests/YourTest.php` should look like this:

*Listing 5-9*

```
namespace YourApp\Tests;

use Silex\WebTestCase;

class YourTest extends WebTestCase
{
    public function createApplication()
    {
        return require __DIR__.'/../../../app.php';
    }

    public function testFooBar()
    {
        ...
    }
}
```

Now, when running `phpunit` on the command line, your tests should run.

# Chapter 6
# Accepting a JSON request body

A common need when building a restful API is the ability to accept a JSON encoded entity from the request body.

An example for such an API could be a blog post creation.

## Example API

In this example we will create an API for creating a blog post. The following is a spec of how we want it to work.

### Request

In the request we send the data for the blog post as a JSON object. We also indicate that using the `Content-Type` header:

*Listing 6-1*

```
POST /blog/posts
Accept: application/json
Content-Type: application/json
Content-Length: 57

{"title":"Hello World!","body":"This is my first post!"}
```

### Response

The server responds with a 201 status code, telling us that the post was created. It tells us the `Content-Type` of the response, which is also JSON:

*Listing 6-2*

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 65
Connection: close

{"id":"1","title":"Hello World!","body":"This is my first post!"}
```

## Parsing the request body

The request body should only be parsed as JSON if the `Content-Type` header begins with `application/json`. Since we want to do this for every request, the easiest solution is to use a before filter.

We simply use `json_decode` to parse the content of the request and then replace the request data on the `$request` object:

*Listing 6-3*

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

$app->before(function (Request $request) {
    if (0 === strpos($request->headers->get('Content-Type'), 'application/json')) {
        $data = json_decode($request->getContent(), true);
        $request->request->replace(is_array($data) ? $data : array());
    }
});
```

## Controller implementation

Our controller will create a new blog post from the data provided and will return the post object, including its `id`, as JSON:

*Listing 6-4*

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

$app->post('/blog/posts', function (Request $request) {
    $post = array(
        'title' => $request->request->get('title'),
        'body'  => $request->request->get('body'),
    );

    $post['id'] = createPost($post);

    return $app->json($post, 201);
});
```

## Manual testing

In order to manually test our API, we can use the `curl` command line utility, which allows sending HTTP requests:

*Listing 6-5*

```
$ curl http://blog.lo/blog/posts -d '{"title":"Hello World!","body":"This is my first post!"}'
-H 'Content-Type: application/json'
{"id":"1","title":"Hello World!","body":"This is my first post!"}
```

Chapter 7

# Translating Validation Messages

When working with Symfony2 validator, a common task would be to show localized validation messages. In order to do that, you will need to register translator and point to translated resources:

*Listing 7-1*

```
$app->register(new Silex\Provider\TranslationServiceProvider(), array(
    'locale' => 'sr_Latn',
    'translator.domains' => array(),
));

$app->before(function () use ($app) {
    $app['translator']->addLoader('xlf', new
Symfony\Component\Translation\Loader\XliffFileLoader());
    $app['translator']->addResource('xlf', __DIR__.'/vendor/symfony/src/Symfony/Bundle/
FrameworkBundle/Resources/translations/validators.sr_Latn.xlf', 'sr_Latn', 'validators');
});
```

And that's all you need to load translations from Symfony2 `xlf` files.

Chapter 8

# How to use PdoSessionStorage to store sessions in the database

By default, the *SessionServiceProvider* writes session information in files using Symfony2 NativeFileSessionStorage. Most medium to large websites use a database to store sessions instead of files, because databases are easier to use and scale in a multi-webserver environment.

Symfony2's *NativeSessionStorage*[1] has multiple storage handlers and one of them uses PDO to store sessions, *PdoSessionHandler*[2]. To use it, replace the `session.storage.handler` service in your application like explained below.

## Example

Listing 8-1

```
use Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler;

$app->register(new Silex\Provider\SessionServiceProvider());

$app['pdo.dsn'] = 'mysql:dbname=mydatabase';
$app['pdo.user'] = 'myuser';
$app['pdo.password'] = 'mypassword';

$app['pdo.db_options'] = array(
    'db_table'      => 'session',
    'db_id_col'     => 'session_id',
    'db_data_col'   => 'session_value',
    'db_time_col'   => 'session_time',
);

$app['pdo'] = $app->share(function () use ($app) {
    return new PDO(
        $app['pdo.dsn'],
        $app['pdo.user'],
```

1. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
2. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session/Storage/Handler/PdoSessionHandler.html

```
        $app['pdo.password']
    );
});

$app['session.storage.handler'] = $app->share(function () use ($app) {
    return new PdoSessionHandler(
        $app['pdo'],
        $app['pdo.db_options'],
        $app['session.storage.options']
    );
});
```

## Database structure

PdoSessionStorage needs a database table with 3 columns:

- `session_id`: ID column (VARCHAR(255) or larger)
- `session_value`: Value column (TEXT or CLOB)
- `session_time`: Time column (INTEGER)

You can find examples of SQL statements to create the session table in the *Symfony2 cookbook*[3]

3. `http://symfony.com/doc/current/cookbook/configuration/pdo_session_storage.html`

Chapter 9

# Disable CSRF Protection on a form using the FormExtension

The *FormExtension* provides a service for building form in your application with the Symfony2 Form component. By default, the *FormExtension* uses the CSRF Protection avoiding Cross-site request forgery, a method by which a malicious user attempts to make your legitimate users unknowingly submit data that they don't intend to submit.

You can find more details about CSRF Protection and CSRF token in the *Symfony2 Book*[1].

In some cases (for example, when embedding a form in an html email) you might want not to use this protection. The easiest way to avoid this is to understand that it is possible to give specific options to your form builder through the `createBuilder()` function.

## Example

Listing 9-1

```
$form = $app['form.factory']->createBuilder('form', null, array('csrf_protection' => false));
```

That's it, your form could be submited from everywhere without CSRF Protection.

## Going further

This specific example showed how to change the `csrf_protection` in the `$options` parameter of the `createBuilder()` function. More of them could be passed through this parameter, it is as simple as using the Symfony2 `getDefaultOptions()` method in your form classes. *See more here*[2].

1. http://symfony.com/doc/current/book/forms.html#csrf-protection
2. http://symfony.com/doc/current/book/forms.html#book-form-creating-form-classes

Chapter 10

# How to use YAML to configure validation

Simplicity is at the heart of Silex so there is no out of the box solution to use YAML files for validation. But this doesn't mean that this is not possible. Let's see how to do it.

First, you need to install the YAML Component. Declare it as a dependency in your `composer.json` file:

*Listing 10-1*

```
"require": {
    "symfony/yaml": "2.1.*"
}
```

Next, you need to tell the Validation Service that you are not using `StaticMethodLoader` to load your class metadata but a YAML file:

*Listing 10-2*

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

$app['validator.mapping.class_metadata_factory'] = new
Symfony\Component\Validator\Mapping\ClassMetadataFactory(
    new Symfony\Component\Validator\Mapping\Loader\YamlFileLoader(__DIR__.'/validation.yml')
);
```

Now, we can replace the usage of the static method and move all the validation rules to `validation.yml`:

*Listing 10-3*

```
# validation.yml
Post:
  properties:
    title:
      - NotNull: ~
      - NotBlank: ~
    body:
      - Min: 100
```

Chapter 11

# Internals

This chapter will tell you a bit about how Silex works internally.

## Silex

### Application

The application is the main interface to Silex. It implements Symfony2's *HttpKernelInterface*[1], so you can pass a *Request*[2] to the `handle` method and it will return a *Response*[3].

It extends the `Pimple` service container, allowing for flexibility on the outside as well as the inside. you could replace any service, and you are also able to read them.

The application makes strong use of the *EventDispatcher*[4] to hook into the Symfony2 *HttpKernel*[5] events. This allows fetching the `Request`, converting string responses into `Response` objects and handling Exceptions. We also use it to dispatch some custom events like before/after filters and errors.

### Controller

The Symfony2 *Route*[6] is actually quite powerful. Routes can be named, which allows for URL generation. They can also have requirements for the variable parts. In order to allow settings these through a nice interface the `match` method (which is used by `get`, `post`, etc.) returns an instance of the `Controller`, which wraps a route.

1. `http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpKernelInterface.html`
2. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/Request.html`
3. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html`
4. `http://api.symfony.com/master/Symfony/Component/EventDispatcher/EventDispatcher.html`
5. `http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpKernel.html`
6. `http://api.symfony.com/master/Symfony/Component/Routing/Route.html`

### ControllerCollection

One of the goals of exposing the *RouteCollection*[7] was to make it mutable, so providers could add stuff to it. The challenge here is the fact that routes know nothing about their name. The name only has meaning in context of the `RouteCollection` and cannot be changed.

To solve this challenge we came up with a staging area for routes. The `ControllerCollection` holds the controllers until `flush` is called, at which point the routes are added to the `RouteCollection`. Also, the controllers are then frozen. This means that they can no longer be modified and will throw an Exception if you try to do so.

Unfortunately no good way for flushing implicitly could be found, which is why flushing is now always explicit. The Application will flush, but if you want to read the `ControllerCollection` before the request takes place, you will have to call flush yourself.

The `Application` provides a shortcut `flush` method for flushing the `ControllerCollection`.

## Symfony2

Following Symfony2 components are used by Silex:

- **HttpFoundation**: For `Request` and `Response`.
- **HttpKernel**: Because we need a heart.
- **Routing**: For matching defined routes.
- **EventDispatcher**: For hooking into the HttpKernel.

For more information, *check out the Symfony website*[8].

7. `http://api.symfony.com/master/Symfony/Component/Routing/RouteCollection.html`
8. `http://symfony.com/`

Chapter 12

# Contributing

We are open to contributions to the Silex code. If you find a bug or want to contribute a provider, just follow these steps.

- Fork *the Silex repository*[1] on github.
- Make your feature addition or bug fix.
- Add tests for it. This is important so we don't break it in a future version unintentionally.
- Send a pull request. Bonus points for topic branches.

If you have a big change or would like to discuss something, please join us on the *mailing list*[2].

Any code you contribute must be licensed under the MIT License.

1. `https://github.com/fabpot/Silex`
2. `http://groups.google.com/group/silex-php`

Chapter 13

# DoctrineServiceProvider

The *DoctrineServiceProvider* provides integration with the *Doctrine DBAL*[1] for easy database acccess.

There is only a Doctrine DBAL. An ORM service is **not** supplied.

## Parameters

- **db.options**: Array of Doctrine DBAL options.

  These options are available:

  - **driver**: The database driver to use, defaults to `pdo_mysql`. Can be any of: `pdo_mysql`, `pdo_sqlite`, `pdo_pgsql`, `pdo_oci`, `oci8`, `ibm_db2`, `pdo_ibm`, `pdo_sqlsrv`.
  - **dbname**: The name of the database to connect to.
  - **host**: The host of the database to connect to. Defaults to localhost.
  - **user**: The user of the database to connect to. Defaults to root.
  - **password**: The password of the database to connect to.
  - **path**: Only relevant for `pdo_sqlite`, specifies the path to the SQLite database.

  These and additional options are described in detail in the Doctrine DBAL configuration documentation.

## Services

- **db**: The database connection, instance of `Doctrine\DBAL\Connection`.
- **db.config**: Configuration object for Doctrine. Defaults to an empty `Doctrine\DBAL\Configuration`.

1. `http://www.doctrine-project.org/projects/dbal`

- **db.event_manager**: Event Manager for Doctrine.

## Registering

Listing 13-1

```
$app->register(new Silex\Provider\DoctrineServiceProvider(), array(
    'db.options' => array(
        'driver'   => 'pdo_sqlite',
        'path'     => __DIR__.'/app.db',
    ),
));
```

Doctrine DBAL comes with the "fat" Silex archive but not with the regular one. If you are using Composer, add it as a dependency to your `composer.json` file:

Listing 13-2

```
"require": {
    "doctrine/dbal": "2.2.*",
 }
```

## Usage

The Doctrine provider provides a `db` service. Here is a usage example:

Listing 13-3

```
$app->get('/blog/show/{id}', function ($id) use ($app) {
    $sql = "SELECT * FROM posts WHERE id = ?";
    $post = $app['db']->fetchAssoc($sql, array((int) $id));

    return  "<h1>{$post['title']}</h1>".
            "<p>{$post['body']}</p>";
});
```

## Using multiple databases

The Doctrine provider can allow access to multiple databases. In order to configure the data sources, replace the **db.options** with **dbs.options**. **dbs.options** is an array of configurations where keys are connection names and values are options:

Listing 13-4

```
$app->register(new Silex\Provider\DoctrineServiceProvider(), array(
    'dbs.options' => array (
        'mysql_read' => array(
            'driver'    => 'pdo_mysql',
            'host'      => 'mysql_read.someplace.tld',
            'dbname'    => 'my_database',
            'user'      => 'my_username',
            'password'  => 'my_password',
        ),
        'mysql_write' => array(
            'driver'    => 'pdo_mysql',
            'host'      => 'mysql_write.someplace.tld',
            'dbname'    => 'my_database',
            'user'      => 'my_username',
            'password'  => 'my_password',
        ),
    ),
));
```

The first registered connection is the default and can simply be accessed as you would if there was only one connection. Given the above configuration, these two lines are equivalent:

*Listing 13-5*
```
$app['db']->fetchAssoc('SELECT * FROM table');

$app['dbs']['mysql_read']->fetchAssoc('SELECT * FROM table');
```

Using multiple connections:

*Listing 13-6*
```
$app->get('/blog/show/{id}', function ($id) use ($app) {
    $sql = "SELECT * FROM posts WHERE id = ?";
    $post = $app['dbs']['mysql_read']->fetchAssoc($sql, array((int) $id));

    $sql = "UPDATE posts SET value = ? WHERE id = ?";
    $app['dbs']['mysql_write']->execute($sql, array('newValue', (int) $id));

    return  "<h1>{$post['title']}</h1>".
            "<p>{$post['body']}</p>";
});
```

For more information, consult the *Doctrine DBAL documentation*[2].

2. `http://www.doctrine-project.org/docs/dbal/2.0/en/`

Chapter 14

# MonologServiceProvider

The *MonologServiceProvider* provides a default logging mechanism through Jordi Boggiano's *Monolog*[1] library.

It will log requests and errors and allow you to add debug logging to your application, so you don't have to use `var_dump` so much anymore. You can use the grown-up version called `tail -f`.

## Parameters

- **monolog.logfile**: File where logs are written to.
- **monolog.level** (optional): Level of logging defaults to `DEBUG`. Must be one of `Logger::DEBUG`, `Logger::INFO`, `Logger::WARNING`, `Logger::ERROR`. `DEBUG` will log everything, `INFO` will log everything except `DEBUG`, etc.
- **monolog.name** (optional): Name of the monolog channel, defaults to `myapp`.

## Services

- **monolog**: The monolog logger instance.

  Example usage:

  Listing 14-1

  ```
  $app['monolog']->addDebug('Testing the Monolog logging.');
  ```

- **monolog.configure**: Protected closure that takes the logger as an argument. You can override it if you do not want the default behavior.

## Registering

Listing 14-2

```
$app->register(new Silex\Provider\MonologServiceProvider(), array(
    'monolog.logfile' => __DIR__.'/development.log',
));
```

1. `https://github.com/Seldaek/monolog`

Monolog comes with the "fat" Silex archive but not with the regular one. If you are using Composer, add it as a dependency to your `composer.json` file:

*Listing 14-3*

```
"require": {
    "monolog/monolog": ">=1.0.0",
}
```

## Usage

The MonologServiceProvider provides a `monolog` service. You can use it to add log entries for any logging level through `addDebug()`, `addInfo()`, `addWarning()` and `addError()`:

*Listing 14-4*

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

$app->post('/user', function () use ($app) {
    // ...

    $app['monolog']->addInfo(sprintf("User '%s' registered.", $username));

    return new Response('', 201);
});
```

## Traits

`Silex\Application\MonologTrait` adds the following shortcuts:

- **log**: Logs a message.

*Listing 14-5*

```
$app->log(sprintf("User '%s' registered.", $username));
```

For more information, check out the *Monolog documentation*[2].

2. `https://github.com/Seldaek/monolog`

# Chapter 15
# SessionServiceProvider

The *SessionServiceProvider* provides a service for storing data persistently between requests.

## Parameters

- **session.storage.save_path** (optional): The path for the `FileSessionHandler`, defaults to the value of `sys_get_temp_dir()`.
- **session.storage.options**: An array of options that is passed to the constructor of the `session.storage` service.

  In case of the default *NativeSessionStorage*[1], the possible options are:

  - **name**: The cookie name (_SESS by default)
  - **id**: The session id (null by default)
  - **cookie_lifetime**: Cookie lifetime
  - **path**: Cookie path
  - **domain**: Cookie domain
  - **secure**: Cookie secure (HTTPS)
  - **httponly**: Whether the cookie is http only

  However, all of these are optional. Sessions last as long as the browser is open. To override this, set the `lifetime` option.
- **session.test**: Whether to simulate sessions or not (useful when writing functional tests).

## Services

- **session**: An instance of Symfony2's *Session*[2].
- **session.storage**: A service that is used for persistence of the session data.
- **session.storage.handler**: A service that is used by the `session.storage` for data access. Defaults to a *FileSessionHandler*[3] storage handler.

1. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html`
2. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session/Session.html`

## Registering

*Listing 15-1*

```
$app->register(new Silex\Provider\SessionServiceProvider());
```

## Usage

The Session provider provides a `session` service. Here is an example that authenticates a user and creates a session for him:

*Listing 15-2*

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

$app->get('/login', function () use ($app) {
    $username = $app['request']->server->get('PHP_AUTH_USER', false);
    $password = $app['request']->server->get('PHP_AUTH_PW');

    if ('igor' === $username && 'password' === $password) {
        $app['session']->set('user', array('username' => $username));
        return $app->redirect('/account');
    }

    $response = new Response();
    $response->headers->set('WWW-Authenticate', sprintf('Basic realm="%s"', 'site_login'));
    $response->setStatusCode(401, 'Please sign in.');
    return $response;
});

$app->get('/account', function () use ($app) {
    if (null === $user = $app['session']->get('user')) {
        return $app->redirect('/login');
    }

    return "Welcome {$user['username']}!";
});
```

3. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session/Storage/Handler/FileSessionHandler.html`

## Chapter 16

# SwiftmailerServiceProvider

The *SwiftmailerServiceProvider* provides a service for sending email through the *Swift Mailer*[1] library.

You can use the `mailer` service to send messages easily. By default, it will attempt to send emails through SMTP.

## Parameters

- **swiftmailer.options**: An array of options for the default SMTP-based configuration.

  The following options can be set:

  - **host**: SMTP hostname, defaults to 'localhost'.
  - **port**: SMTP port, defaults to 25.
  - **username**: SMTP username, defaults to an empty string.
  - **password**: SMTP password, defaults to an empty string.
  - **encryption**: SMTP encryption, defaults to null.
  - **auth_mode**: SMTP authentication mode, defaults to null.

## Services

- **mailer**: The mailer instance.

  Example usage:

  ```
  $message = \Swift_Message::newInstance();

  // ...

  $app['mailer']->send($message);
  ```

  *Listing 16-1*

- **swiftmailer.transport**: The transport used for e-mail delivery. Defaults to a `Swift_Transport_EsmtpTransport`.

1. `http://swiftmailer.org`

- **swiftmailer.transport.buffer**: StreamBuffer used by the transport.
- **swiftmailer.transport.authhandler**: Authentication handler used by the transport. Will try the following by default: CRAM-MD5, login, plaintext.
- **swiftmailer.transport.eventdispatcher**: Internal event dispatcher used by Swiftmailer.

## Registering

Listing 16-2

```
$app->register(new Silex\Provider\SwiftmailerServiceProvider());
```

SwiftMailer comes with the "fat" Silex archive but not with the regular one. If you are using Composer, add it as a dependency to your `composer.json` file:

Listing 16-3

```
"require": {
    "swiftmailer/swiftmailer": ">=4.1.2,<4.2-dev"
}
```

## Usage

The Swiftmailer provider provides a `mailer` service:

Listing 16-4

```
$app->post('/feedback', function () use ($app) {
    $request = $app['request'];

    $message = \Swift_Message::newInstance()
        ->setSubject('[YourSite] Feedback')
        ->setFrom(array('noreply@yoursite.com'))
        ->setTo(array('feedback@yoursite.com'))
        ->setBody($request->get('message'));

    $app['mailer']->send($message);

    return new Response('Thank you for your feedback!', 201);
});
```

## Traits

`Silex\Application\SwiftmailerTrait` adds the following shortcuts:

- **mail**: Sends an email.

Listing 16-5

```
$app->mail(\Swift_Message::newInstance()
    ->setSubject('[YourSite] Feedback')
    ->setFrom(array('noreply@yoursite.com'))
    ->setTo(array('feedback@yoursite.com'))
    ->setBody($request->get('message')));
```

For more information, check out the *Swift Mailer documentation*[2].

2. `http://swiftmailer.org`

# Chapter 17
# TranslationServiceProvider

The *TranslationServiceProvider* provides a service for translating your application into different languages.

## Parameters

- **translator.domains** (optional): A mapping of domains/locales/messages. This parameter contains the translation data for all languages and domains.
- **locale** (optional): The locale for the translator. You will most likely want to set this based on some request parameter. Defaults to `en`.
- **locale_fallback** (optional): Fallback locale for the translator. It will be used when the current locale has no messages set.

## Services

- **translator**: An instance of *Translator*[1], that is used for translation.
- **translator.loader**: An instance of an implementation of the translation *LoaderInterface*[2], defaults to an *ArrayLoader*[3].
- **translator.message_selector**: An instance of *MessageSelector*[4].

## Registering

*Listing 17-1*

```
$app->register(new Silex\Provider\TranslationServiceProvider(), array(
    'locale_fallback' => 'en',
));
```

1. `http://api.symfony.com/master/Symfony/Component/Translation/Translator.html`
2. `http://api.symfony.com/master/Symfony/Component/Translation/Loader/LoaderInterface.html`
3. `http://api.symfony.com/master/Symfony/Component/Translation/Loader/ArrayLoader.html`
4. `http://api.symfony.com/master/Symfony/Component/Translation/MessageSelector.html`

The Symfony Translation Component comes with the "fat" Silex archive but not with the regular one. If you are using Composer, add it as a dependency to your `composer.json` file:

*Listing 17-2*
```
"require": {
    "symfony/translation": "2.1.*"
}
```

## Usage

The Translation provider provides a `translator` service and makes use of the `translator.domains` parameter:

*Listing 17-3*
```
$app['translator.domains'] = array(
    'messages' => array(
        'en' => array(
            'hello'     => 'Hello %name%',
            'goodbye'   => 'Goodbye %name%',
        ),
        'de' => array(
            'hello'     => 'Hallo %name%',
            'goodbye'   => 'Tschüss %name%',
        ),
        'fr' => array(
            'hello'     => 'Bonjour %name%',
            'goodbye'   => 'Au revoir %name%',
        ),
    ),
    'validators' => array(
        'fr' => array(
            'This value should be a valid number.' => 'Cette valeur doit être un nombre.',
        ),
    ),
);

$app->get('/{locale}/{message}/{name}', function ($message, $name) use ($app) {
    return $app['translator']->trans($message, array('%name%' => $name));
});
```

The above example will result in following routes:

- `/en/hello/igor` will return `Hello igor`.
- `/de/hello/igor` will return `Hallo igor`.
- `/fr/hello/igor` will return `Bonjour igor`.
- `/it/hello/igor` will return `Hello igor` (because of the fallback).

## Traits

`Silex\Application\TranslationTrait` adds the following shortcuts:

- **trans**: Translates the given message.
- **transChoice**: Translates the given choice message by choosing a translation according to a number.

```
$app->trans('Hello World');

$app->transChoice('Hello World');
```

Listing 17-4

## Recipes

### YAML-based language files

Having your translations in PHP files can be inconvenient. This recipe will show you how to load translations from external YAML files.

First, add the Symfony2 `Config` and `Yaml` components in your composer file:

```
"require": {
    "symfony/config": "2.1.*",
    "symfony/yaml": "2.1.*"
}
```

Listing 17-5

Next, you have to create the language mappings in YAML files. A naming you can use is `locales/en.yml`. Just do the mapping in this file as follows:

```
hello: Hello %name%
goodbye: Goodbye %name%
```

Listing 17-6

Then, register the `YamlFileLoader` on the `translator` and add all your translation files:

```
use Symfony\Component\Translation\Loader\YamlFileLoader;

$app['translator'] = $app->share($app->extend('translator', function($translator, $app) {
    $translator->addLoader('yaml', new YamlFileLoader());

    $translator->addResource('yaml', __DIR__.'/locales/en.yml', 'en');
    $translator->addResource('yaml', __DIR__.'/locales/de.yml', 'de');
    $translator->addResource('yaml', __DIR__.'/locales/fr.yml', 'fr');

    return $translator;
}));
```

Listing 17-7

### XLIFF-based language files

Just as you would do with YAML translation files, you first need to add the Symfony2 `Config` component as a dependency (see above for details).

Then, similarly, create XLIFF files in your locales directory and add them to the translator:

```
$translator->addResource('xliff', __DIR__.'/locales/en.xlf', 'en');
$translator->addResource('xliff', __DIR__.'/locales/de.xlf', 'de');
$translator->addResource('xliff', __DIR__.'/locales/fr.xlf', 'fr');
```

Listing 17-8

The XLIFF loader is already pre-configured by the extension.

### Accessing translations in Twig templates

Once loaded, the translation service provider is available from within Twig templates:

Listing 17-9

```
{{ app.translator.trans('translation_key') }}
```

Moreover, when using the Twig bridge provided by Symfony (see *TwigServiceProvider*), you will be allowed to translate strings in the Twig way:

Listing 17-10

```
{{ 'translation_key'|trans }}
{{ 'translation_key'|transchoice }}
{% trans %}translation_key{% endtrans %}
```

Chapter 18

# TwigServiceProvider

The *TwigServiceProvider* provides integration with the *Twig*[1] template engine.

## Parameters

- **twig.path** (optional): Path to the directory containing twig template files (it can also be an array of paths).
- **twig.templates** (optional): An associative array of template names to template contents. Use this if you want to define your templates inline.
- **twig.options** (optional): An associative array of twig options. Check out the twig documentation for more information.
- **twig.form.templates** (optional): An array of templates used to render forms (only available when the `FormServiceProvider` is enabled).

## Services

- **twig**: The `Twig_Environment` instance. The main way of interacting with Twig.
- **twig.loader**: The loader for Twig templates which uses the `twig.path` and the `twig.templates` options. You can also replace the loader completely.

## Registering

Listing 18-1

```
$app->register(new Silex\Provider\TwigServiceProvider(), array(
    'twig.path' => __DIR__.'/views',
));
```

1. `http://twig.sensiolabs.org/`

Twig comes with the "fat" Silex archive but not with the regular one. If you are using Composer, add it as a dependency to your composer.json file:

Listing 18-2
```
"require": {
    "twig/twig": ">=1.8,<2.0-dev"
}
```

## Symfony2 Components Integration

Symfony provides a Twig bridge that provides additional integration between some Symfony2 components and Twig. Add it as a dependency to your composer.json file:

Listing 18-3
```
"require": {
    "symfony/twig-bridge": "2.1.*",
}
```

When present, the TwigServiceProvider will provide you with the following additional capabilities:

- **UrlGeneratorServiceProvider**: If you are using the UrlGeneratorServiceProvider, you will have access to the path() and url() functions. You can find more information in the Symfony2 Routing documentation.
- **TranslationServiceProvider**: If you are using the TranslationServiceProvider, you will get the trans() and transchoice() functions for translation in Twig templates. You can find more information in the *Symfony2 Translation documentation*[2].
- **FormServiceProvider**: If you are using the FormServiceProvider, you will get a set of helpers for working with forms in templates. You can find more information in the *Symfony2 Forms reference*[3].
- **SecurityServiceProvider**: If you are using the SecurityServiceProvider, you will have access to the is_granted() function in templates. You can find more information in the Symfony2 Security documentation.

## Usage

The Twig provider provides a twig service:

Listing 18-4
```
$app->get('/hello/{name}', function ($name) use ($app) {
    return $app['twig']->render('hello.twig', array(
        'name' => $name,
    ));
});
```

This will render a file named views/hello.twig.

In any Twig template, the app variable refers to the Application object. So you can access any services from within your view. For example to access $app['request']->getHost(), just put this in your template:

Listing 18-5
```
{{ app.request.host }}
```

A render function is also registered to help you render another controller from a template:

2. http://symfony.com/doc/current/book/translation.html#twig-templates
3. http://symfony.com/doc/current/reference/forms/twig_reference.html

```
{{ render('/sidebar') }}

{# or if you are also using UrlGeneratorServiceProvider with the SymfonyBridgesServiceProvider
#}
{{ render(path('sidebar')) }}
```
*Listing 18-6*

## Traits

`Silex\Application\TwigTrait` adds the following shortcuts:

- **render**: Renders a view with the given parameters and returns a Response object.

```
return $app->render('index.html', ['name': 'Fabien']);

$response = new Response();
$response->setTtl(10);

return $app->render('index.html', ['name': 'Fabien'], $response);
```
*Listing 18-7*

```
// stream a view
use Symfony\Component\HttpFoundation\StreamedResponse;

return $app->render('index.html', ['name': 'Fabien'], new StreamedResponse());
```
*Listing 18-8*

## Customization

You can configure the Twig environment before using it by extending the `twig` service:

```
$app['twig'] = $app->share($app->extend('twig', function($twig, $app) {
    $twig->addGlobal('pi', 3.14);
    $twig->addFilter('levenshtein', new \Twig_Filter_Function('levenshtein'));

    return $twig;
}));
```
*Listing 18-9*

For more information, check out the *Twig documentation*[4].

4. `http://twig.sensiolabs.org`

Chapter 19

# UrlGeneratorServiceProvider

The *UrlGeneratorServiceProvider* provides a service for generating URLs for named routes.

## Parameters

None.

## Services

- **url_generator**: An instance of *UrlGenerator*[1], using the *RouteCollection*[2] that is provided through the `routes` service. It has a `generate` method, which takes the route name as an argument, followed by an array of route parameters.

## Registering

Listing 19-1

```
$app->register(new Silex\Provider\UrlGeneratorServiceProvider());
```

## Usage

The UrlGenerator provider provides a `url_generator` service:

Listing 19-2

```
$app->get('/', function () {
    return 'welcome to the homepage';
})
->bind('homepage');

$app->get('/hello/{name}', function ($name) {
```

1. `http://api.symfony.com/master/Symfony/Component/Routing/Generator/UrlGenerator.html`
2. `http://api.symfony.com/master/Symfony/Component/Routing/RouteCollection.html`

```
    return "Hello $name!";
})
->bind('hello');

$app->get('/navigation', function () use ($app) {
    return '<a href="'.$app['url_generator']->generate('homepage').'">Home</a>'.
           ' | '.
           '<a href="'.$app['url_generator']->generate('hello', array('name' =>
'Igor')).'">Hello Igor</a>';
});
```

When using Twig, the service can be used like this:

*Listing 19-3*

```
{{ app.url_generator.generate('homepage') }}
```

Moreover, if you use Twig, you will have access to the `path()` and `url()` functions:

*Listing 19-4*

```
{{ path('homepage') }}
{{ url('homepage') }} {# generates the absolute url http://example.org/ #}
```

## Traits

`Silex\Application\UrlGeneratorTrait` adds the following shortcuts:

- **path**: Generates a path.
- **url**: Generates an absolute URL.

*Listing 19-5*

```
$app->path('homepage');
$app->url('homepage');
```

# Chapter 20
# ValidatorServiceProvider

The *ValidatorServiceProvider* provides a service for validating data. It is most useful when used with the *FormServiceProvider*, but can also be used standalone.

## Parameters

none

## Services

- **validator**: An instance of *Validator*[1].
- **validator.mapping.class_metadata_factory**: Factory for metadata loaders, which can read validation constraint information from classes. Defaults to StaticMethodLoader--ClassMetadataFactory.

  This means you can define a static `loadValidatorMetadata` method on your data class, which takes a ClassMetadata argument. Then you can set constraints on this ClassMetadata instance.
- **validator.validator_factory**: Factory for ConstraintValidators. Defaults to a standard `ConstraintValidatorFactory`. Mostly used internally by the Validator.

## Registering

Listing 20-1

```
$app->register(new Silex\Provider\ValidatorServiceProvider());
```

The Symfony Validator Component comes with the "fat" Silex archive but not with the regular one. If you are using Composer, add it as a dependency to your `composer.json` file:

1. `http://api.symfony.com/master/Symfony/Component/Validator/Validator.html`

```
"require": {
    "symfony/validator": "2.1.*"
}
```

Listing 20-2

## Usage

The Validator provider provides a `validator` service.

### Validating Values

You can validate values directly using the `validateValue` validator method:

Listing 20-3

```
use Symfony\Component\Validator\Constraints as Assert;

$app->get('/validate/{email}', function ($email) use ($app) {
    $errors = $app['validator']->validateValue($email, new Assert\Email());

    if (count($errors) > 0) {
        return (string) $errors;
    } else {
        return 'The email is valid';
    }
});
```

### Validating Associative Arrays

Validating associative arrays is like validating simple values, with a collection of constraints:

Listing 20-4

```
use Symfony\Component\Validator\Constraints as Assert;

class Book
{
    public $title;
    public $author;
}

class Author
{
    public $first_name;
    public $last_name;
}

$book = array(
    'title' => 'My Book',
    'author' => array(
        'first_name' => 'Fabien',
        'last_name'  => 'Potencier',
    ),
);

$constraint = new Assert\Collection(array(
    'title' => new Assert\MinLength(10),
    'author' => new Assert\Collection(array(
        'first_name' => array(new Assert\NotBlank(), new Assert\MinLength(10)),
        'last_name'  => new Assert\MinLength(10),
    )),
));
$errors = $app['validator']->validateValue($book, $constraint);

if (count($errors) > 0) {
    foreach ($errors as $error) {
        echo $error->getPropertyPath().' '.$error->getMessage()."\n";
    }
} else {
    echo 'The book is valid';
}
```

## Validating Objects

If you want to add validations to a class, you can define the constraint for the class properties and getters, and then call the `validate` method:

*Listing 20-5*

```
use Symfony\Component\Validator\Constraints as Assert;

$author = new Author();
$author->first_name = 'Fabien';
$author->last_name = 'Potencier';

$book = new Book();
$book->title = 'My Book';
$book->author = $author;

$metadata = $app['validator.mapping.class_metadata_factory']->getClassMetadata('Author');
$metadata->addPropertyConstraint('first_name', new Assert\NotBlank());
$metadata->addPropertyConstraint('first_name', new Assert\MinLength(10));
$metadata->addPropertyConstraint('last_name', new Assert\MinLength(10));

$metadata = $app['validator.mapping.class_metadata_factory']->getClassMetadata('Book');
$metadata->addPropertyConstraint('title', new Assert\MinLength(10));
$metadata->addPropertyConstraint('author', new Assert\Valid());

$errors = $app['validator']->validate($book);

if (count($errors) > 0) {
    foreach ($errors as $error) {
        echo $error->getPropertyPath().' '.$error->getMessage()."\n";
    }
} else {
    echo 'The author is valid';
}
```

You can also declare the class constraint by adding a static `loadValidatorMetadata` method to your classes:

*Listing 20-6*

```
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints as Assert;

class Book
{
    public $title;
    public $author;

    static public function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('title', new Assert\MinLength(10));
        $metadata->addPropertyConstraint('author', new Assert\Valid());
    }
}

class Author
{
    public $first_name;
    public $last_name;

    static public function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('first_name', new Assert\NotBlank());
        $metadata->addPropertyConstraint('first_name', new Assert\MinLength(10));
        $metadata->addPropertyConstraint('last_name', new Assert\MinLength(10));
    }
}

$app->get('/validate/{email}', function ($email) use ($app) {
    $author = new Author();
    $author->first_name = 'Fabien';
    $author->last_name = 'Potencier';

    $book = new Book();
    $book->title = 'My Book';
    $book->author = $author;

    $errors = $app['validator']->validate($book);

    if (count($errors) > 0) {
        foreach ($errors as $error) {
            echo $error->getPropertyPath().' '.$error->getMessage()."\n";
        }
    } else {
        echo 'The author is valid';
    }
});
```

Use `addGetterConstraint()` to add constraints on getter methods and `addConstraint()` to add constraints on the class itself.

## Translation

To be able to translate the error messages, you can use the translator provider and register the messages under the `validators` domain:

Listing 20-7

```
$app['translator.domains'] = array(
    'validators' => array(
        'fr' => array(
            'This value should be a valid number.' => 'Cette valeur doit être un nombre.',
        ),
    ),
);
```

For more information, consult the *Symfony2 Validation documentation*[2].

2. `http://symfony.com/doc/2.0/book/validation.html`

# Chapter 21
# FormServiceProvider

The *FormServiceProvider* provides a service for building forms in your application with the Symfony2 Form component.

## Parameters

- **form.secret**: This secret value is used for generating and validating the CSRF token for a specific page. It is very important for you to set this value to a static randomly generated value, to prevent hijacking of your forms. Defaults to `md5(__DIR__)`.

## Services

- **form.factory**: An instance of *FormFactory*[1], that is used for build a form.
- **form.csrf_provider**: An instance of an implementation of the *CsrfProviderInterface*[2], defaults to a *DefaultCsrfProvider*[3].

## Registering

*Listing 21-1*

```
use Silex\Provider\FormServiceProvider;

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

The Symfony Form Component comes with the "fat" Silex archive but not with the regular one. If you are using Composer, add it as a dependency to your `composer.json` file:

1. http://api.symfony.com/master/Symfony/Component/Form/FormFactory.html
2. http://api.symfony.com/master/Symfony/Component/Form/Extension/Csrf/CsrfProvider/CsrfProviderInterface.html
3. http://api.symfony.com/master/Symfony/Component/Form/Extension/Csrf/CsrfProvider/DefaultCsrfProvider.html

```
    "require": {
        "symfony/form": "2.1.*"
    }
```
*Listing 21-2*

If you are going to use the validation extension with forms, you must also register the `symfony/config` and `` `symfony/translation `` components:

```
"require": {
    "symfony/config": "2.1.*",
    "symfony/translation": "2.1.*"
}
```
*Listing 21-3*

The Symfony Form Component relies on the PHP intl extension. If you don't have it, you can install the Symfony Locale Component as a replacement:

```
"require": {
    "symfony/locale": "2.1.*"
}
```
*Listing 21-4*

If you want to benefit from the internationalization of your form, you must install the PHP intl extension.

## Usage

The FormServiceProvider provides a `form.factory` service. Here is a usage example:

```
$app->match('/form', function (Request $request) use ($app) {
    // some default data for when the form is displayed the first time
    $data = array(
        'name' => 'Your name',
        'email' => 'Your email',
    );

    $form = $app['form.factory']->createBuilder('form', $data)
        ->add('name')
        ->add('email')
        ->add('gender', 'choice', array(
            'choices' => array(1 => 'male', 2 => 'female'),
            'expanded' => true,
        ))
        ->getForm();

    if ('POST' == $request->getMethod()) {
        $form->bindRequest($request);

        if ($form->isValid()) {
            $data = $form->getData();

            // do something with the data

            // redirect somewhere
            return $app->redirect('...');
        }
```
*Listing 21-5*

```
    }

    // display the form
    return $app['twig']->render('index.twig', array('form' => $form->createView()));
});
```

And here is the index.twig form template:

Listing 21-6

```
<form action="#" method="post">
    {{ form_widget(form) }}

    <input type="submit" name="submit" />
</form>
```

If you are using the validator provider, you can also add validation to your form by adding constraints on the fields:

Listing 21-7

```
use Symfony\Component\Validator\Constraints as Assert;

$app->register(new Silex\Provider\ValidatorServiceProvider());
$app->register(new Silex\Provider\TranslationServiceProvider(), array(
    'translator.messages' => array(),
));

$form = $app['form.factory']->createBuilder('form')
    ->add('name', 'text', array(
        'constraints' => array(new Assert\NotBlank(), new Assert\MinLength(5))
    ))
    ->add('email', 'text', array(
        'constraints' => new Assert\Email()
    ))
    ->add('gender', 'choice', array(
        'choices' => array(1 => 'male', 2 => 'female'),
        'expanded' => true,
        'constraints' => new Assert\Choice(array(1, 2)),
    ))
    ->getForm();
```

You can register form extensions by extending form.extensions:

Listing 21-8

```
$app['form.extensions'] = $app->share($app->extend('form.extensions', function ($extensions)
use ($app) {
    $extensions[] = new YourTopFormExtension();

    return $extensions;
}));
```

## Traits

Silex\Application\FormTrait adds the following shortcuts:

- **form**: Creates a FormBuilder instance.

Listing 21-9

```
$app->form('form', $data);
```

For more information, consult the *Symfony2 Forms documentation*[4].

4. http://symfony.com/doc/2.1/book/forms.html

Chapter 22

# HttpCacheServiceProvider

The *HttpCacheProvider* provides support for the Symfony2 Reverse Proxy.

## Parameters

- **http_cache.cache_dir**: The cache directory to store the HTTP cache data.
- **http_cache.options** (optional): An array of options for the *HttpCache*[1] constructor.

## Services

- **http_cache**: An instance of *HttpCache*[2].

## Registering

```
$app->register(new Silex\Provider\HttpCacheServiceProvider(), array(
    'http_cache.cache_dir' => __DIR__.'/cache/',
));
```

*Listing 22-1*

## Usage

Silex already supports any reverse proxy like Varnish out of the box by setting Response HTTP cache headers:

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

$app->get('/', function() {
    return new Response('Foo', 200, array(
```

*Listing 22-2*

1. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpCache/HttpCache.html
2. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpCache/HttpCache.html

```
        'Cache-Control' => 's-maxage=5',
    ));
});
```

If you want Silex to trust the `X-Forwarded-For*` headers from your reverse proxy, you will need to run your application like this:

Listing 22-3

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

Request::trustProxyData();
$app->run();
```

This provider allows you to use the Symfony2 reverse proxy natively with Silex applications by using the `http_cache` service:

Listing 22-4

```
$app['http_cache']->run();
```

The provider also provides ESI support:

Listing 22-5

```
$app->get('/', function() {
    return new Response(<<<EOF
<html>
    <body>
        Hello
        <esi:include src="/included" />
    </body>
</html>

EOF
    , 200, array(
        'Cache-Control' => 's-maxage=20',
        'Surrogate-Control' => 'content="ESI/1.0"',
    ));
});

$app->get('/included', function() {
    return new Response('Foo', 200, array(
        'Cache-Control' => 's-maxage=5',
    ));
});

$app['http_cache']->run();
```

To help you debug caching issues, set your application `debug` to true. Symfony automatically adds a `X-Symfony-Cache` header to each response with useful information about cache hits and misses.

If you are *not* using the Symfony Session provider, you might want to set the PHP `session.cache_limiter` setting to an empty value to avoid the default PHP behavior.

Finally, check that your Web server does not override your caching strategy.

For more information, consult the *Symfony2 HTTP Cache documentation*[3].

3. `http://symfony.com/doc/current/book/http_cache.html`

Chapter 23

# SecurityServiceProvider

The *SecurityServiceProvider* manages authentication and authorization for your applications.

## Parameters

n/a

## Services

- **security**: The main entry point for the security provider. Use it to get the current user token.
- **security.authentication_manager**: An instance of *AuthenticationProviderManager*[1], responsible for authentication.
- **security.access_manager**: An instance of *AccessDecisionManager*[2], responsible for authorization.
- **security.session_strategy**: Define the session strategy used for authentication (default to a migration strategy).
- **security.user_checker**: Checks user flags after authentication.
- **security.last_error**: Returns the last authentication errors when given a Request object.
- **security.encoder_factory**: Defines the encoding strategies for user passwords (default to use a digest algorithm for all users).

The service provider defines many other services that are used internally but rarely need to be customized.

1. `http://api.symfony.com/master/Symfony/Component/Security/Core/Authentication/AuthenticationProviderManager.html`
2. `http://api.symfony.com/master/Symfony/Component/Security/Core/Authorization/AccessDecisionManager.html`

## Registering

Listing 23-1

```php
$app->register(new Silex\Provider\SecurityServiceProvider());
```

The Symfony Security Component comes with the "fat" Silex archive but not with the regular one. If you are using Composer, add it as a dependency to your `composer.json` file:

Listing 23-2

```json
"require": {
    "symfony/security": "2.1.*"
}
```

## Usage

The Symfony Security component is powerful. To learn more about it, read the *Symfony2 Security documentation*[3].

When a security configuration does not behave as expected, enable logging (with the Monolog extension for instance) as the Security Component logs a lot of interesting information about what it does and why.

Below is a list of recipes that cover some common use cases.

### Accessing the current User

The current user information is stored in a token that is accessible via the `security` service:

Listing 23-3

```php
$token = $app['security']->getToken();
```

If there is no information about the user, the token is `null`. If the user is known, you can get it with a call to `getUser()`:

Listing 23-4

```php
if (null !== $token) {
    $user = $token->getUser();
}
```

The user can be a string, and object with a `__toString()` method, or an instance of *UserInterface*[4].

### Securing a Path with HTTP Authentication

The following configuration uses HTTP basic authentication to secure URLs under `/admin/`:

Listing 23-5

```php
$app['security.firewalls'] = array(
    'admin' => array(
        'pattern' => '^/admin/',
        'http' => true,
        'users' => array(
            // raw password is foo
            'admin' => array('ROLE_ADMIN',
'5FZ2Z8QIkA7UTZ4BYkoC+GsReLf569mSKDsfods6LYQ8t+a8EW9oaircfMpmaLbPBh4FOBiiFyLfuZmTSUwzZg=='),
```

3. `http://symfony.com/doc/2.1/book/security.html`
4. `http://api.symfony.com/master/Symfony/Component/Security/Core/User/UserInterface.html`

```
        ),
    ),
);
```

The `pattern` is a regular expression (it can also be a *RequestMatcher*[5] instance); the `http` setting tells the security layer to use HTTP basic authentication and the `users` entry defines valid users.

Each user is defined with the following information:

- The role or an array of roles for the user (roles are strings beginning with `ROLE_` and ending with anything you want);
- The user encoded password.

⚠ All users must at least have one role associated with them.

The default configuration of the extension enforces encoded passwords. To generate a valid encoded password from a raw password, use the `security.encoder_factory` service:

*Listing 23-6*

```
// find the encoder for a UserInterface instance
$encoder = $app['security.encoder_factory']->getEncoder($user);

// compute the encoded password for foo
$password = $encoder->encodePassword('foo', $user->getSalt());
```

When the user is authenticated, the user stored in the token is an instance of *User*[6]

## Securing a Path with a Form

Using a form to authenticate users is very similar to the above configuration. Instead of using the `http` setting, use the `form` one and define these two parameters:

- **login_path**: The login path where the user is redirected when he is accessing a secured area without being authenticated so that he can enter his credentials;
- **check_path**: The check URL used by Symfony to validate the credentials of the user.

Here is how to secure all URLs under `/admin/` with a form:

*Listing 23-7*

```
$app['security.firewalls'] = array(
    'admin' => array(
        'pattern' => '^/admin/',
        'form' => array('login_path' => '/login', 'check_path' => '/admin/login_check'),
        'users' => array(
            'admin' => array('ROLE_ADMIN',
'5FZ2Z8QIkA7UTZ4BYkoC+GsReLf569mSKDsfods6LYQ8t+a8EW9oaircfMpmaLbPBh4FOBiiFyLfuZmTSUwzZg=='),
        ),
    ),
);
```

Always keep in mind the following two golden rules:

- The `login_path` path must always be defined **outside** the secured area (or if it is in the secured area, the `anonymous` authentication mechanism must be enabled -- see below);
- The `check_path` path must always be defined **inside** the secured area.

5. http://api.symfony.com/master/Symfony/Component/HttpFoundation/RequestMatcher.html
6. http://api.symfony.com/master/Symfony/Component/Security/Core/User/User.html

For the login form to work, create a controller like the following:

Listing 23-8
```
$app->get('/login', function(Request $request) use ($app) {
    return $app['twig']->render('login.html', array(
        'error'         => $app['security.last_error']($request),
        'last_username' => $app['session']->get('_security.last_username'),
    ));
});
```

The `error` and `last_username` variables contain the last authentication error and the last username entered by the user in case of an authentication error.

Create the associated template:

Listing 23-9
```
<form action="{{ path('admin_login_check') }}" method="post">
    {{ error }}
    <input type="text" name="_username" value="{{ last_username }}" />
    <input type="password" name="_password" value="" />
    <input type="submit" />
</form>
```

The `admin_login_check` route is automatically defined by Silex and its name is derived from the `check_path` value (all `/` are replaced with `_` and the leading `/` is stripped).

## Defining more than one Firewall

You are not limited to define one firewall per project.

Configuring several firewalls is useful when you want to secure different parts of your website with different authentication strategies or for different users (like using an HTTP basic authentication for the website API and a form to secure your website administration area).

It's also useful when you want to secure all URLs except the login form:

Listing 23-10
```
$app['security.firewalls'] = array(
    'login' => array(
        'pattern' => '^/login$',
    ),
    'secured' => array(
        'pattern' => '^.*$',
        'form' => array('login_path' => '/login', 'check_path' => '/login_check'),
        'users' => array(
            'admin' => array('ROLE_ADMIN',
'5FZ2Z8QIkA7UTZ4BYkoC+GsReLf569mSKDsfods6LYQ8t+a8EW9oaircfMpmaLbPBh4FOBiiFyLfuZmTSUwzZg=='),
        ),
    ),
);
```

The order of the firewall configurations is significant as the first one to match wins. The above configuration first ensures that the `/login` URL is not secured (no authentication settings), and then it secures all other URLs.

## Adding a Logout

When using a form for authentication, you can let users log out if you add the `logout` setting:

Listing 23-11
```
$app['security.firewalls'] = array(
    'secured' => array(
        'form' => array('login_path' => '/login', 'check_path' => '/admin/login_check'),
        'logout' => array('logout_path' => '/logout'),

        // ...
    ),
);
```

A route is automatically generated, based on the configured path (all / are replaced with _ and the leading / is stripped):

Listing 23-12

```
<a href="{{ path('logout') }}">Logout</a>
```

## Allowing Anonymous Users

When securing only some parts of your website, the user information are not available in non-secured areas. To make the user accessible in such areas, enabled the `anonymous` authentication mechanism:

Listing 23-13

```
$app['security.firewalls'] = array(
    'unsecured' => array(
        'anonymous' => true,

        // ...
    ),
);
```

When enabling the anonymous setting, a user will always be accessible from the security context; if the user is not authenticated, it returns the `anon.` string.

## Checking User Roles

To check if a user is granted some role, use the `isGranted()` method on the security context:

Listing 23-14

```
if ($app['security']->isGranted('ROLE_ADMIN') {
    // ...
}
```

You can check roles in Twig templates too:

Listing 23-15

```
{% if is_granted('ROLE_ADMIN') %}
    <a href="/secured?_switch_user=fabien">Switch to Fabien</a>
{% endif %}
```

You can check is a user is "fully authenticated" (not an anonymous user for instance) with the special `IS_AUTHENTICATED_FULLY` role:

Listing 23-16

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <a href="{{ path('logout') }}">Logout</a>
{% else %}
    <a href="{{ path('login') }}">Login</a>
{% endif %}
```

Don't use the `getRoles()` method to check user roles.

⚠ `isGranted()` throws an exception when no authentication information is available (which is the case on non-secured area).

## Impersonating a User

If you want to be able to switch to another user (without knowing the user credentials), enable the `switch_user` authentication strategy:

Listing 23-17

```
$app['security.firewalls'] = array(
    'unsecured' => array(
        'switch_user' => array('parameter' => '_switch_user', 'role' =>
'ROLE_ALLOWED_TO_SWITCH'),

        // ...
    ),
);
```

Switching to another user is now a matter of adding the `_switch_user` query parameter to any URL when logged in as a user who has the `ROLE_ALLOWED_TO_SWITCH` role:

Listing 23-18

```
{% if is_granted('ROLE_ALLOWED_TO_SWITCH') %}
    <a href="?_switch_user=fabien">Switch to user Fabien</a>
{% endif %}
```

You can check that you are impersonating a user by checking the special `ROLE_PREVIOUS_ADMIN`. This is useful for instance to allow the user to switch back to his primary account:

Listing 23-19

```
{% if is_granted('ROLE_PREVIOUS_ADMIN') %}
    You are an admin but you've switched to another user,
    <a href="?_switch_user=_exit"> exit</a> the switch.
{% endif %}
```

## Defining a Role Hierarchy

Defining a role hierarchy allows to automatically grant users some additional roles:

Listing 23-20

```
$app['security.role_hierarchy'] = array(
    'ROLE_ADMIN' => array('ROLE_USER', 'ROLE_ALLOWED_TO_SWITCH'),
);
```

With this configuration, all users with the `ROLE_ADMIN` role also automatically have the `ROLE_USER` and `ROLE_ALLOWED_TO_SWITCH` roles.

## Defining Access Rules

Roles are a great way to adapt the behavior of your website depending on groups of users, but they can also be used to further secure some areas by defining access rules:

Listing 23-21

```
$app['security.access_rules'] = array(
    array('^/admin', 'ROLE_ADMIN', 'https'),
    array('^.*$', 'ROLE_USER'),
);
```

With the above configuration, users must have the `ROLE_ADMIN` to access the `/admin` section of the website, and `ROLE_USER` for everything else. Furthermore, the admin section can only be accessible via HTTPS (if that's not the case, the user will be automatically redirected).

The first argument can also be a *RequestMatcher*[7] instance.

## Defining a custom User Provider

Using an array of users is simple and useful when securing an admin section of a personal website, but you can override this default mechanism with you own.

The `users` setting can be defined as a service that returns an instance of *UserProvider*[8]:

Listing 23-22

```
'users' => $app->share(function () use ($app) {
    return new UserProvider($app['db']);
}),
```

Here is a simple example of a user provider, where Doctrine DBAL is used to store the users:

Listing 23-23

```
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\User;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
use Doctrine\DBAL\Connection;
use Doctrine\DBAL\Schema\Table;

class UserProvider implements UserProviderInterface
{
    private $conn;

    public function __construct(Connection $conn)
    {
        $this->conn = $conn;
    }

    public function loadUserByUsername($username)
    {
        $stmt = $this->conn->executeQuery('SELECT * FROM users WHERE username = ?',
array(strtolower($username)));

        if (!$user = $stmt->fetch()) {
            throw new UsernameNotFoundException(sprintf('Username "%s" does not exist.',
$username));
        }

        return new User($user['username'], $user['password'], explode(',', $user['roles']),
true, true, true, true);
    }

    public function refreshUser(UserInterface $user)
    {
        if (!$user instanceof User) {
            throw new UnsupportedUserException(sprintf('Instances of "%s" are not supported.',
get_class($user)));
```

7. `http://api.symfony.com/master/Symfony/Component/HttpFoundation/RequestMatcher.html`
8. `http://api.symfony.com/master/Symfony/Component/Security/Core/User/UserProviderInterface.html`

```
        }

        return $this->loadUserByUsername($user->getUsername());
    }

    public function supportsClass($class)
    {
        return $class === 'Symfony\Component\Security\Core\User\User';
    }
}
```

In this example, instances of the default `User` class are created for the users, but you can define your own class; the only requirement is that the class must implement *UserInterface*[9]

And here is the code that you can use to create the database schema and some sample users:

Listing 23-24

```
$schema = $conn->getSchemaManager();
if (!$schema->tablesExist('users')) {
    $users = new Table('users');
    $users->addColumn('id', 'integer', array('unsigned' => true));
    $users->setPrimaryKey(array('id'));
    $users->addColumn('username', 'string', array('length' => 32));
    $users->addUniqueIndex(array('username'));
    $users->addColumn('password', 'string', array('length' => 255));
    $users->addColumn('roles', 'string', array('length' => 255));

    $schema->createTable($users);

    $this->conn->executeQuery('INSERT INTO users (username, password, roles) VALUES ("fabien",
"5FZ2Z8QIkA7UTZ4BYkoC+GsReLf569mSKDsfods6LYQ8t+a8EW9oaircfMpmaLbPBh4FOBiiFyLfuZmTSUwzZg==",
"ROLE_USER")');
    $this->conn->executeQuery('INSERT INTO users (username, password, roles) VALUES ("admin",
"5FZ2Z8QIkA7UTZ4BYkoC+GsReLf569mSKDsfods6LYQ8t+a8EW9oaircfMpmaLbPBh4FOBiiFyLfuZmTSUwzZg==",
"ROLE_ADMIN")');
}
```

If you are using the Doctrine ORM, the Symfony bridge for Doctrine provides a user provider class that is able to load users from your entities.

## Defining a custom Authentication Provider

The Symfony Security component provides a lot of ready-to-use authentication providers (form, HTTP, X509, remember me, ...), but you can add new ones easily. To register a new authentication provider, create a service named `security.authentication.factory.XXX` where `XXX` is the name you want to use in your configuration:

Listing 23-25

```
$app['security.authentication_listener.factory.wsse'] = $app->protect(function ($name,
$options) use ($app) {
    // define the authentication provider object
    $app['security.authentication_provider.'.$name.'.wsse'] = $app->share(function () use
($app) {
        return new WsseProvider($app['security.user_provider.default'],
__DIR__.'/security_cache');
    });
```

9. `http://api.symfony.com/master/Symfony/Component/Security/Core/User/UserInterface.html`

```
    // define the authentication listener object
    $app['security.authentication_listener.'.$name.'.wsse'] = $app->share(function () use
($app) {
        return new WsseListener($app['security'], $app['security.authentication_manager']);
    });

    return array(
        // the authentication provider id
        'security.authentication_provider.'.$name.'.wsse',
        // the authentication listener id
        'security.authentication_listener.'.$name.'.wsse',
        // the entry point id
        null,
        // the position of the listener in the stack
        'pre_auth'
    );
});
```

You can now use it in your configuration like any other built-in authentication provider:

Listing 23-26

```
$app->register(new SecurityServiceProvider(), array(
    'security.firewalls' => array(
        'default' => array(
            'wsse' => true,

            // ...
        ),
    ),
));
```

Instead of `true`, you can also define an array of options that customize the behavior of your authentication factory; it will be passed as the second argument of your authentication factory (see above).

This example uses the authentication provider classes as described in the Symfony *cookbook*[10].

## Traits

`Silex\Application\SecurityTrait` adds the following shortcuts:

- **user**: Returns the current user.
- **encodePassword**: Encode a given password.

Listing 23-27

```
$user = $app->user();

$encoded = $app->encodePassword($user, 'foo');
```

`Silex\Route\SecurityTrait` adds the following methods to the controllers:

- **secure**: Secures a controller for the given roles.

Listing 23-28

```
$app->get('/', function () {
    // do something but only for admins
})->secure('ROLE_ADMIN');
```

10. http://symfony.com/doc/current/cookbook/security/custom_authentication_provider.html

# Chapter 24
# Webserver Configuration

## Apache

If you are using Apache you can use a `.htaccess` file for this:

Listing 24-1

```
<IfModule mod_rewrite.c>
    Options -MultiViews

    RewriteEngine On
    #RewriteBase /path/to/app
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

If your site is not at the webroot level you will have to uncomment the `RewriteBase` statement and adjust the path to point to your directory, relative from the webroot.

Alternatively, if you use Apache 2.2.16 or higher, you can use the *FallbackResource directive*[1] so make your .htaccess even easier:

Listing 24-2

```
FallbackResource index.php
```

## nginx

If you are using nginx, configure your vhost to forward non-existent resources to `index.php`:

Listing 24-3

```
server {
    index index.php
```

1. `http://www.adayinthelifeof.nl/2012/01/21/apaches-fallbackresource-your-new-htaccess-command/`

```
    location / {
        try_files $uri $uri/ /index.php;
    }

    location ~ index\.php$ {
        fastcgi_pass   /var/run/php5-fpm.sock;
        fastcgi_index  index.php;
        include fastcgi_params;
    }
}
```

## IIS

If you are using the Internet Information Services from Windows, you can use this sample `web.config` file:

Listing 24-4

```
<?xml version="1.0"?>
<configuration>
    <system.webServer>
        <defaultDocument>
            <files>
                <clear />
                <add value="index.php" />
            </files>
        </defaultDocument>
        <rewrite>
            <rules>
                <rule name="Silex Front Controller" stopProcessing="true">
                    <match url="^(.*)$" ignoreCase="false" />
                    <conditions logicalGrouping="MatchAll">
                        <add input="{REQUEST_FILENAME}" matchType="IsFile" ignoreCase="false"
negate="true" />
                    </conditions>
                    <action type="Rewrite" url="index.php" appendQueryString="true" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

## Lighttpd

If you are using lighttpd, use this sample `simple-vhost` as a starting point:

Listing 24-5

```
server.document-root = "/path/to/app"

url.rewrite-once = (
    # configure some static files
    "^/assets/.+" => "$0",
    "^/favicon\.ico$" => "$0",

    "^(/[^\?]*)(\?.*)?" => "/index.php$1$2"
)
```

## PHP 5.4

PHP 5.4 ships with a built-in webserver for development. This server allows you to run silex without any configuration. Assuming your front controller is at `web/index.php`, you can start the server from the command-line with this command:

*Listing 24-6*

```
$ php -S localhost:8080 -t web
```

Now the application should be running at `http://localhost:8080`.

This server is for development only. It is **not** recommended to use it in production.

Chapter 25

# Changelog

- **2012-06-17**: `ControllerCollection` now takes a required route instance as a constructor argument.

  Before:

  *Listing 25-1*

  ```
  $controllers = new ControllerCollection();
  ```

  After:

  *Listing 25-2*

  ```
  $controllers = new ControllerCollection(new Route());

  // or even better
  $controllers = $app['controllers_factory'];
  ```

- **2012-06-17**: added application traits for PHP 5.4
- **2012-06-16**: renamed `request.default_locale` to `locale`
- **2012-06-16**: Removed the `translator.loader` service. See documentation for how to use XLIFF or YAML-based translation files.
- **2012-06-15**: removed the `twig.configure` service. Use the `extend` method instead:

  Before:

  *Listing 25-3*

  ```
  $app['twig.configure'] = $app->protect(function ($twig) use ($app) {
      // do something
  });
  ```

  After:

  *Listing 25-4*

  ```
  $app['twig'] = $app->share($app->extend('twig', function($twig, $app) {
      // do something

      return $twig;
  }));
  ```

- **2012-06-13**: Added a route `before` middleware
- **2012-06-13**: Renamed the route `middleware` to `before`
- **2012-06-13**: Added an extension for the Symfony Security component
- **2012-05-31**: Made the `BrowserKit`, `CssSelector`, `DomCrawler`, `Finder` and `Process` components optional dependencies. Projects that depend on them (e.g. through functional tests) should add those dependencies to their `composer.json`.
- **2012-05-26**: added `boot()` to `ServiceProviderInterface`.
- **2012-05-26**: Removed `SymfonyBridgesServiceProvider`. It is now implicit by checking the existence of the bridge.
- **2012-05-26**: Removed the `translator.messages` parameter (use `translator.domains` instead).
- **2012-05-24**: Removed the `autoloader` service (use composer instead). The `*.class_path` settings on all the built-in providers have also been removed in favor of Composer.
- **2012-05-21**: Changed error() to allow handling specific exceptions.
- **2012-05-20**: Added a way to define settings on a controller collection.
- **2012-05-20**: The Request instance is not available anymore from the Application after it has been handled.
- **2012-04-01**: Added `finish` filters.
- **2012-03-20**: Added `json` helper:

Listing 25-5

```
$data = array('some' => 'data');
$response = $app->json($data);
```

- **2012-03-11**: Added route middlewares.
- **2012-03-02**: Switched to use Composer for dependency management.
- **2012-02-27**: Updated to Symfony 2.1 session handling.
- **2012-01-02**: Introduced support for streaming responses.
- **2011-09-22**: `ExtensionInterface` has been renamed to `ServiceProviderInterface`. All built-in extensions have been renamed accordingly (for instance, `Silex\Extension\TwigExtension` has been renamed to `Silex\Provider\TwigServiceProvider`).
- **2011-09-22**: The way reusable applications work has changed. The `mount()` method now takes an instance of `ControllerCollection` instead of an `Application` one.

  Before:

  Listing 25-6

  ```
  $app = new Application();
  $app->get('/bar', function() { return 'foo'; });

  return $app;
  ```

  After:

  Listing 25-7

  ```
  $app = new ControllerCollection();
  $app->get('/bar', function() { return 'foo'; });

  return $app;
  ```

- **2011-08-08**: The controller method configuration is now done on the Controller itself

  Before:

```
$app->match('/', function () { echo 'foo'; }, 'GET|POST');
```

*Listing 25-8*

After:

```
$app->match('/', function () { echo 'foo'; })->method('GET|POST');
```

*Listing 25-9*

# Chapter 26

# Phar File

Using the Silex `phar` file is deprecated. You should use Composer instead to install Silex and its dependencies or download one of the archives.

## Installing

Installing Silex is as easy as downloading the *phar*[1] and storing it somewhere on the disk. Then, require it in your script:

*Listing 26-1*

```php
<?php

require_once __DIR__.'/silex.phar';

$app = new Silex\Application();

$app->get('/hello/{name}', function ($name) use ($app) {
    return 'Hello '.$app->escape($name);
});

$app->run();
```

## Console

Silex includes a lightweight console for updating to the latest version.

To find out which version of Silex you are using, invoke `silex.phar` on the command-line with `version` as an argument:

*Listing 26-2*

```
$ php silex.phar version
Silex version 0a243d3 2011-04-17 14:49:31 +0200
```

1. `http://silex.sensiolabs.org/get/silex.phar`

To check that your are using the latest version, run the `check` command:

```
$ php silex.phar check
```

Listing 26-3

To update `silex.phar` to the latest version, invoke the `update` command:

```
$ php silex.phar update
```

Listing 26-4

This will automatically download a new `silex.phar` from `silex.sensiolabs.org` and replace the existing one.

## Pitfalls

There are some things that can go wrong. Here we will try and outline the most frequent ones.

### PHP configuration

Certain PHP distributions have restrictive default Phar settings. Setting the following may help.

```
detect_unicode = Off
phar.readonly = Off
phar.require_hash = Off
```

Listing 26-5

If you are on Suhosin you will also have to set this:

```
suhosin.executor.include.whitelist = phar
```

Listing 26-6

Ubuntu's PHP ships with Suhosin, so if you are using Ubuntu, you will need this change.

### Phar-Stub bug

Some PHP installations have a bug that throws a `PharException` when trying to include the Phar. It will also tell you that `Silex\Application` could not be found. A workaround is using the following include line:

```
require_once 'phar://'.__DIR__.'/silex.phar/autoload.php';
```

Listing 26-7

The exact cause of this issue could not be determined yet.

### ioncube loader bug

Ioncube loader is an extension that can decode PHP encoded file. Unfortunately, old versions (prior to version 4.0.9) are not working well with phar archives. You must either upgrade Ioncube loader to version 4.0.9 or newer or disable it by commenting or removing this line in your php.ini file:

```
zend_extension = /usr/lib/php5/20090626+lfs/ioncube_loader_lin_5.3.so
```

Listing 26-8