The Silex Book

generated on February 27, 2017

The Silex Book

Contents at a Glance

Chapter 1
Introduction

Silex is a PHP microframework. It is built on the shoulders of *Symfony*[1] and *Pimple*[2] and also inspired by *Sinatra*[3].

Silex aims to be:

- *Concise*: Silex exposes an intuitive and concise API.
- *Extensible*: Silex has an extension system based around the Pimple service-container that makes it easy to tie in third party libraries.
- *Testable*: Silex uses Symfony'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.

Usage

Listing 1-1

```php
<?php

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

1. http://symfony.com/
2. http://pimple.sensiolabs.org/
3. http://www.sinatrarb.com/

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

Chapter 2
Usage

Installation

If you want to get started fast, use the *Silex Skeleton*[1]:

Listing 2-1
```
composer create-project fabpot/silex-skeleton path/to/install "~2.0"
```

If you want more flexibility, use *Composer*[2] instead:

Listing 2-2
```
composer require silex/silex:~2.0
```

Web Server

All examples in the documentation rely on a well-configured web server; read the webserver documentation to check yours.

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 2-3
```
// web/index.php
require_once __DIR__.'/../vendor/autoload.php';

$app = new Silex\Application();

// ... definitions
```

1. http://github.com/silexphp/Silex-Skeleton
2. http://getcomposer.org/

```
7
8 $app->run();
```

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

Listing 2-4
```
$app['debug'] = true;
```

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

Listing 2-5
```
use Symfony\Component\HttpFoundation\Request;

Request::setTrustedProxies(array($ip));
$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`, `PATCH`, or `OPTIONS`. This describes the interaction with the resource.

The controller is defined using a closure like this:

Listing 2-6
```
function () {
    // ... do something
}
```

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

Example GET Route

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

Listing 2-7
```
1  $blogPosts = array(
2      1 => array(
3          'date'      => '2011-03-29',
4          'author'    => 'igorw',
5          'title'     => 'Using Silex',
6          'body'      => '...',
7      ),
8  );
9
10 $app->get('/blog', function () use ($blogPosts) {
11     $output = '';
12     foreach ($blogPosts as $post) {
13         $output .= $post['title'];
14         $output .= '<br />';
15     }
16
17     return $output;
18 });
```

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 2-8

```
$app->get('/blog/{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.

The current `Application` is automatically injected by Silex to the Closure thanks to the type hinting.

When the post does not exist, you are using `abort()` to stop the request early. It actually throws an exception, which you 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. You will use the `mail` function to send an e-mail:

Listing 2-9

```
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*[3], so you can fetch variables using the request `get` method.

Instead of returning a string you are returning an instance of *Response*[4]. 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`.

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

Other methods

You can create controllers for most HTTP methods. Just call one of these methods on your application: `get`, `post`, `put`, `delete`, `patch`, `options`:

Listing 2-10
```
$app->put('/blog/{id}', function ($id) {
    // ...
});

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

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

Forms in most web browsers do not directly support the use of other HTTP methods. To use methods other than GET and POST you can utilize a special form field with a name of `_method`. The form's `method` attribute must be set to POST when using this field:

Listing 2-11
```
<form action="/my/target/route/" method="post">
    <!-- ... -->
    <input type="hidden" id="_method" name="_method" value="PUT" />
</form>
```

You need to explicitly enable this method override:

Listing 2-12
```
use Symfony\Component\HttpFoundation\Request;

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

You can also call `match`, which will match all methods. This can be restricted via the `method` method:

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

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

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

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 2-14
```
$app->get('/blog/{id}', function ($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 2-15
```
$app->get('/blog/{postId}/{commentId}', function ($postId, $commentId) {
    // ...
});
```

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

Listing 2-16
```
$app->get('/blog/{postId}/{commentId}', function ($commentId, $postId) {
    // ...
});
```

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

Listing 2-17
```
$app->get('/blog/{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 2-18
```
$app->get('/blog/{id}', function (Application $foo, Request $bar, $id) {
    // ...
});
```

Route Variable Converters

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

Listing 2-19
```
$app->get('/user/{id}', function ($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 2-20
```
1  $userProvider = function ($id) {
2      return new User($id);
3  };
4
5  $app->get('/user/{user}', function (User $user) {
6      // ...
7  })->convert('user', $userProvider);
8
9  $app->get('/user/{user}/edit', function (User $user) {
10     // ...
11 })->convert('user', $userProvider);
```

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

Listing 2-21
```
1  $callback = function ($post, Request $request) {
2      return new Post($request->attributes->get('slug'));
3  };
4
5  $app->get('/blog/{id}/{slug}', function (Post $post) {
6      // ...
7  })->convert('post', $callback);
```

A converter can also be defined as a service. For example, here is a user converter based on Doctrine ObjectManager:

Listing 2-22

```php
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class UserConverter
{
    private $om;

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

    public function convert($id)
    {
        if (null === $user = $this->om->find('User', (int) $id)) {
            throw new NotFoundHttpException(sprintf('User %d does not exist', $id));
        }

        return $user;
    }
}
```

The service will now be registered in the application, and the `convert()` method will be used as converter (using the syntax `service_name:method_name`):

Listing 2-23
```php
$app['converter.user'] = function () {
    return new UserConverter();
};

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

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 a positive integer, since `\d+` matches any amount of digits:

Listing 2-24
```php
$app->get('/blog/{id}', function ($id) {
    // ...
})
->assert('id', '\d+');
```

You can also chain these calls:

Listing 2-25
```php
$app->get('/blog/{postId}/{commentId}', function ($postId, $commentId) {
    // ...
})
->assert('postId', '\d+')
->assert('commentId', '\d+');
```

Conditions

Besides restricting route matching based on the HTTP method or parameter requirements, you can set conditions on any part of the request by calling `when` on the `Controller` object, which is returned by the routing methods:

Listing 2-26
```php
$app->get('/blog/{id}', function ($id) {
    // ...
```

```
})
->when("request.headers.get('User-Agent') matches '/firefox/i'");
```

The `when` argument is a Symfony *Expression*[5] , which means that you need to add `symfony/expression-language` as a dependency of your project.

Default Values

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

Listing 2-27
```
$app->get('/{pageName}', function ($pageName) {
    // ...
})
->value('pageName', 'index');
```

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

Named Routes

Some providers can make use of named routes. By default Silex will generate an internal route name for you but you can give an explicit route name by calling `bind`:

Listing 2-28
```
$app->get('/', function () {
    // ...
})
->bind('homepage');

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

Controllers as Classes

Instead of anonymous functions, you can also define your controllers as methods. By using the `ControllerClass::methodName` syntax, you can tell Silex to lazily create the controller object for you:

Listing 2-29
```
$app->get('/', 'Acme\\Foo::bar');

use Silex\Application;
use Symfony\Component\HttpFoundation\Request;

namespace Acme
{
    class Foo
    {
        public function bar(Request $request, Application $app)
        {
            // ...
        }
    }
}
```

This will load the `Acme\Foo` class on demand, create an instance and call the `bar` method to get the response. You can use `Request` and `Silex\Application` type hints to get `$request` and `$app` injected.

It is also possible to define your controllers as services.

5. https://symfony.com/doc/current/book/routing.html#completely-customized-route-matching-with-conditions

Global Configuration

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

Listing 2-30
```
$app['controllers']
    ->value('id', '1')
    ->assert('id', '\d+')
    ->requireHttps()
    ->method('get')
    ->convert('id', function () { /* ... */ })
    ->before(function () { /* ... */ })
    ->when('request.isSecure() == true')
;
```

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 (read the dedicated chapter for more information).

Error Handlers

When an exception is thrown, error handlers allow you to display a custom error page to the user. They can also be used 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:

Listing 2-31
```
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;

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

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

Listing 2-32
```
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;

$app->error(function (\Exception $e, Request $request, $code) {
    switch ($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);
});
```

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

Listing 2-33
```
use Symfony\Component\HttpFoundation\Request;

$app->error(function (\LogicException $e, Request $request, $code) {
```

```
    // this handler will only handle \LogicException exceptions
    // and exceptions that extend \LogicException
});
```

As Silex ensures that the Response status code is set to the most appropriate one depending on the exception, setting the status on the response won't work. If you want to overwrite the status code, set the `X-Status-Code` header:

Listing 2-34
```
return new Response('Error', 404 /* ignored */, array('X-Status-Code' => 200));
```

If you want to use a separate error handler for logging, make sure you register it with a higher priority than response error handlers, because once a response is returned, the following handlers are ignored.

Silex ships with a provider for *Monolog*[6] 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 2-35
```
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;

$app->error(function (\Exception $e, Request $request, $code) use ($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 2-36
```
$app->get('/blog/{id}', function (Silex\Application $app, $id) use ($blogPosts) {
    if (!isset($blogPosts[$id])) {
        $app->abort(404, "Post $id does not exist.");
    }

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

You can convert errors to `Exceptions`, check out the cookbook chapter for details.

View Handlers

View Handlers allow you to intercept a controller result that is not a `Response` and transform it before it gets returned to the kernel.

To register a view handler, pass a callable (or string that can be resolved to a callable) to the `view()` method. The callable should accept some sort of result from the controller:

Listing 2-37

6. https://github.com/Seldaek/monolog

```php
$app->view(function (array $controllerResult) use ($app) {
    return $app->json($controllerResult);
});
```

View Handlers also receive the `Request` as their second argument, making them a good candidate for basic content negotiation:

Listing 2-38
```php
$app->view(function (array $controllerResult, Request $request) use ($app) {
    $acceptHeader = $request->headers->get('Accept');
    $bestFormat = $app['negotiator']->getBestFormat($acceptHeader, array('json', 'xml'));

    if ('json' === $bestFormat) {
        return new JsonResponse($controllerResult);
    }

    if ('xml' === $bestFormat) {
        return $app['serializer.xml']->renderResponse($controllerResult);
    }

    return $controllerResult;
});
```

View Handlers will be examined in the order they are added to the application and Silex will use type hints to determine if a view handler should be used for the current result, continuously using the return value of the last view handler as the input for the next.

You must ensure that Silex receives a `Response` or a string as the result of the last view handler (or controller) to be run.

Redirects

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

Listing 2-39
```php
$app->get('/', function () use ($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 2-40
```php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\HttpKernelInterface;

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

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

You can also generate the URI via the built-in URL generator:

Listing 2-41
```
$request = Request::create($app['url_generator']->generate('hello'), 'GET');
```

There's some more things that you need to keep in mind though. In most cases you will want to forward some parts of the current master request to the sub-request. That includes: Cookies, server information, session. Read more on how to make sub-requests.

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 2-42
```
$app->get('/users/{id}', function ($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 stream a response, which is important in cases when you don't want to buffer the data being sent:

Listing 2-43
```
$app->get('/images/{file}', function ($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 2-44
```
$stream = function () {
    $fh = fopen('http://www.example.com/', 'rb');
    while (!feof($fh)) {
        echo fread($fh, 1024);
        ob_flush();
        flush();
    }
    fclose($fh);
};
```

Sending a file

If you want to return a file, you can use the `sendFile` helper method. It eases returning files that would otherwise not be publicly available. Simply pass it your file path, status code, headers and the content disposition and it will create a `BinaryFileResponse` response for you:

Listing 2-45
```
$app->get('/files/{path}', function ($path) use ($app) {
    if (!file_exists('/base/path/' . $path)) {
        $app->abort(404);
    }

    return $app->sendFile('/base/path/' . $path);
});
```

To further customize the response before returning it, check the API doc for *SymfonyComponentHttpFoundationBinaryFileResponse*[7]:

Listing 2-46
```
return $app
    ->sendFile('/base/path/' . $path)
    ->setContentDisposition(ResponseHeaderBag::DISPOSITION_ATTACHMENT, 'pic.jpg')
;
```

Traits

Silex comes with PHP traits that define shortcut methods.

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.

7. http://api.symfony.com/master/Symfony/Component/HttpFoundation/BinaryFileResponse.html

Security

Make sure to protect your application against attacks.

Escaping

When outputting any user input, 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
```
use Symfony\Component\HttpFoundation\Request;

$app->get('/name', function (Request $request, Silex\Application $app) {
    $name = $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. Check out the *Providers* chapter for details.

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

Listing 2-51
```
use Symfony\Component\HttpFoundation\Request;

$app->get('/name.json', function (Request $request, Silex\Application $app) {
    $name = $request->get('name');

    return $app->json(array('name' => $name));
});
```

Chapter 3
Middleware

Silex allows you to run code, that changes the default Silex behavior, at different stages during the handling of a request through *middleware*:

- *Application middleware* is triggered independently of the current handled request;
- *Route middleware* is triggered when its associated route is matched.

Application Middleware

Application middleware is only run for the "master" Request.

Before Middleware

A *before* application middleware allows you to tweak the Request before the controller is executed:

Listing 3-1
```
$app->before(function (Request $request, Application $app) {
    // ...
});
```

By default, the middleware is run after the routing and the security.

If you want your middleware to be run even if an exception is thrown early on (on a 404 or 403 error for instance), then, you need to register it as an early event:

Listing 3-2
```
$app->before(function (Request $request, Application $app) {
    // ...
}, Application::EARLY_EVENT);
```

In this case, the routing and the security won't have been executed, and so you won't have access to the locale, the current route, or the security user.

The before middleware is an event registered on the Symfony *request* event.

After Middleware

An *after* application middleware allows you to tweak the Response before it is sent to the client:

Listing 3-3
```
$app->after(function (Request $request, Response $response) {
    // ...
});
```

The after middleware is an event registered on the Symfony *response* event.

Finish Middleware

A *finish* application middleware allows you to execute tasks after the Response has been sent to the client (like sending emails or logging):

Listing 3-4
```
$app->finish(function (Request $request, Response $response) {
    // ...
    // Warning: modifications to the Request or Response will be ignored
});
```

The finish middleware is an event registered on the Symfony *terminate* event.

Route Middleware

Route middleware is added to routes or route collections and it is only triggered when the corresponding route is matched. You can also stack them:

Listing 3-5
```
1  $app->get('/somewhere', function () {
2      // ...
3  })
4  ->before($before1)
5  ->before($before2)
6  ->after($after1)
7  ->after($after2)
8  ;
```

Before Middleware

A *before* route middleware is fired just before the route callback, but after the *before* application middleware:

Listing 3-6
```
1  $before = function (Request $request, Application $app) {
2      // ...
3  };
4
5  $app->get('/somewhere', function () {
6      // ...
7  })
8  ->before($before);
```

After Middleware

An *after* route middleware is fired just after the route callback, but before the application *after* application middleware:

Listing 3-7
```php
$after = function (Request $request, Response $response, Application $app) {
    // ...
};

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

Middleware Priority

You can add as much middleware as you want, in which case they are triggered in the same order as you added them.

You can explicitly control the priority of your middleware by passing an additional argument to the registration methods:

Listing 3-8
```php
$app->before(function (Request $request) {
    // ...
}, 32);
```

As a convenience, two constants allow you to register an event as early as possible or as late as possible:

Listing 3-9
```php
$app->before(function (Request $request) {
    // ...
}, Application::EARLY_EVENT);

$app->before(function (Request $request) {
    // ...
}, Application::LATE_EVENT);
```

Short-circuiting the Controller

If a *before* middleware returns a `Response` object, the request handling is short-circuited (the next middleware won't be run, nor the route callback), and the Response is passed to the *after* middleware right away:

Listing 3-10
```php
$app->before(function (Request $request) {
    // redirect the user to the login screen if access to the Resource is protected
    if (...) {
        return new RedirectResponse('/login');
    }
});
```

A `RuntimeException` is thrown if a before middleware does not return a Response or `null`.

Chapter 4
Organizing Controllers

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

Listing 4-1
```php
// 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);

// define controllers for a admin
$app->mount('/admin', function ($admin) {
    // recursively mount
    $admin->mount('/blog', function ($user) {
        $user->get('/', function () {
            return 'Admin Blog home page';
        });
    });
});
```

`$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, `/forum/` to the forum home page, and `/admin/blog/` to the admin blog 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 4-2

```
$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 4-3

```
// 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.

Chapter 5
Services

Silex is not only a framework, it is also a service container. It does this by extending *Pimple*[1] which provides a very simple service container.

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 5-1

```
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.

A 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.

Pimple

Pimple 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\Container` all of this applies to Silex as well:

Listing 5-2
```php
$container = new Pimple\Container();
```

or:

Listing 5-3
```php
$app = new Silex\Application();
```

Parameters

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

Listing 5-4
```php
$app['some_parameter'] = 'value';
```

The array key can be any value. By convention dots are used for namespacing:

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

Reading parameter values is possible with the same syntax:

Listing 5-6
```php
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 5-7
```php
$app['some_service'] = function () {
    return new Service();
};
```

And to retrieve the service, use:

Listing 5-8
```php
$service = $app['some_service'];
```

On first invocation, this will create the service; the same instance will then be returned on any subsequent access.

Factory services

If you want a different instance to be returned for each service access, wrap the service definition with the `factory()` method:

Listing 5-9
```php
$app['some_service'] = $app->factory(function () {
    return new Service();
});
```

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

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 5-10
```
$app['some_service'] = function ($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.

Going back to our initial example, here's how we could use the container to manage its dependencies:

Listing 5-11
```
$app['user.persist_path'] = '/tmp/users';
$app['user.persister'] = function ($app) {
    return new JsonUserPersister($app['user.persist_path']);
};
```

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 5-12
```
$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.

- **request_stack**: Controls the lifecycle of requests, an instance of *RequestStack*[2]. It gives you access to `GET`, `POST` parameters and lots more!

  Example usage:

Listing 5-13
```
$id = $app['request_stack']->getCurrentRequest()->get('id');
```

2. http://api.symfony.com/master/Symfony/Component/HttpFoundation/RequestStack.html

A request is only available when a request is being served; you can only access it from within a controller, an application before/after middlewares, or an error handler.

- **routes**: The *RouteCollection*[3] that is used internally. You can add, modify, read routes.
- **url_generator**: An instance of *UrlGenerator*[4], using the *RouteCollection*[5] 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.
- **controllers**: The `Silex\ControllerCollection` that is used internally. Check the Internals chapter for more information.
- **dispatcher**: The *EventDispatcher*[6] that is used internally. It is the core of the Symfony system and is used quite a bit by Silex.
- **resolver**: The *ControllerResolver*[7] that is used internally. It takes care of executing the controller with the right arguments.
- **kernel**: The *HttpKernel*[8] that is used internally. The HttpKernel is the heart of Symfony, 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 URL generator.
- **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*[9] instance. By default, logging is disabled as the value is set to `null`. To enable logging you can either use the MonologServiceProvider or define your own `logger` service that conforms to the PSR logger interface.

Core traits

- `Silex\Application\UrlGeneratorTrait` adds the following shortcuts:
  - **path**: Generates a path.
  - **url**: Generates an absolute URL.

Listing 5-14
```
1  $app->path('homepage');
2  $app->url('homepage');
```

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 when generating URLs.

3. http://api.symfony.com/master/Symfony/Component/Routing/RouteCollection.html
4. http://api.symfony.com/master/Symfony/Component/Routing/Generator/UrlGenerator.html
5. http://api.symfony.com/master/Symfony/Component/Routing/RouteCollection.html
6. http://api.symfony.com/master/Symfony/Component/EventDispatcher/EventDispatcher.html
7. http://api.symfony.com/master/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
8. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpKernel.html
9. https://github.com/php-fig/log/blob/master/Psr/Log/LoggerInterface.php

- **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 when generating URLs.
- **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 6
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 6-1
```
$app = new Silex\Application();

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

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

Listing 6-2
```
$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 these rules in mind:

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

  Reason: If the service already exists, the provider will overwrite it.
- You can set parameters any time **after** the provider is registered, but **before** the service is accessed.

  Reason: Providers can set default values for parameters. Just like with services, the provider will overwrite existing values.

Included providers

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

- DoctrineServiceProvider
- FormServiceProvider
- HttpCacheServiceProvider
- MonologServiceProvider
- RememberMeServiceProvider
- SecurityServiceProvider
- SerializerServiceProvider
- ServiceControllerServiceProvider
- SessionServiceProvider
- SwiftmailerServiceProvider
- TranslationServiceProvider
- TwigServiceProvider
- ValidatorServiceProvider

The Silex core team maintains a *WebProfiler*[1] provider that helps debug code in the development environment thanks to the Symfony web debug toolbar and the Symfony profiler.

Third party providers

Some service providers are developed by the community. Those third-party providers are listed on *Silex' repository wiki*[2].

You are encouraged to share yours.

Creating a provider

Providers must implement the `Pimple\ServiceProviderInterface`:

Listing 6-3

```
interface ServiceProviderInterface
{
    public function register(Container $container);
}
```

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

The `Pimple\ServiceProviderInterface` belongs to the Pimple package, so take care to only use the API of `Pimple\Container` within your `register` method. Not only is this a good practice due to the way Pimple and Silex work, but may allow your provider to be used outside of Silex.

Optionally, your service provider can implement the `Silex\Api\BootableProviderInterface`. A bootable provider must implement the `boot()` method, with which you can configure the application, just before it handles a request:

Listing 6-4

```
interface BootableProviderInterface
{
```

1. https://github.com/silexphp/Silex-WebProfiler
2. https://github.com/silexphp/Silex/wiki/Third-Party-ServiceProviders-for-Silex-2.x

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

Another optional interface, is the `Silex\Api\EventListenerProviderInterface`. This interface contains the `subscribe()` method, which allows your provider to subscribe event listener with Silex's EventDispatcher, just before it handles a request:

Listing 6-5
```
interface EventListenerProviderInterface
{
    function subscribe(Container $app, EventDispatcherInterface $dispatcher);
}
```

Here is an example of such a provider:

Listing 6-6
```
namespace Acme;

use Pimple\Container;
use Pimple\ServiceProviderInterface;
use Silex\Application;
use Silex\Api\BootableProviderInterface;
use Silex\Api\EventListenerProviderInterface;
use Symfony\Component\EventDispatcher\EventDispatcherInterface;
use Symfony\Component\HttpKernel\KernelEvents;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class HelloServiceProvider implements ServiceProviderInterface, BootableProviderInterface,
EventListenerProviderInterface
{
    public function register(Container $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);
        });
    }

    public function boot(Application $app)
    {
        // do something
    }

    public function subscribe(Container $app, EventDispatcherInterface $dispatcher)
    {
        $dispatcher->addListener(KernelEvents::REQUEST, function(FilterResponseEvent $event) use ($app) {
            // do something
        });
    }
}
```

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 6-7
```
use Symfony\Component\HttpFoundation\Request;

$app = new Silex\Application();

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

$app->get('/hello', function (Request $request) use ($app) {
    $name = $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`.

Controller Providers

Loading providers

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

Listing 6-8
```
$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\Api\ControllerProviderInterface`:

Listing 6-9
```
interface ControllerProviderInterface
{
    public function connect(Application $app);
}
```

Here is an example of such a provider:

Listing 6-10
```
namespace Acme;

use Silex\Application;
use Silex\Api\ControllerProviderInterface;

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 use this provider as follows:

Listing 6-11
```
$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 7
Testing

Because Silex is built on top of Symfony, 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 7-1
```
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

Listing 7-2
```php
public function testInitialPage()
{
    $statusCode = ...
    $pageContent = ...

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

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

Symfony 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:

Listing 7-3
```php
use Silex\WebTestCase;

class ContactFormTest extends WebTestCase
{
    ...
}
```

If you need to override the `setUp()` method, don't forget to call the parent (`parent::setUp()`) to call the Silex default setup.

If you want to use the Symfony `WebTestCase` class you will need to explicitly install its dependencies for your project:

Listing 7-4
```
composer require --dev symfony/browser-kit symfony/css-selector
```

For your WebTestCase, you will have to implement a `createApplication` method, which returns your application instance:

Listing 7-5
```php
public function createApplication()
{
    // app.php must return an Application instance
    return require __DIR__.'/path/to/app.php';
}
```

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

3. https://phpunit.de/manual/current/en/writing-tests-for-phpunit.html

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:

Listing 7-6
```php
public function createApplication()
{
    $app = require __DIR__.'/path/to/app.php';
    $app['debug'] = true;
    unset($app['exception_handler']);

    return $app;
}
```

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

Listing 7-7
```php
public function createApplication()
{
    // ...

    $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 7-8
```php
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 Symfony 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 Symfony 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 7-9

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="./vendor/autoload.php"
         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>
```

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

Listing 7-10

```
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, tests should run.

Chapter 8
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 8-1

```
1 POST /blog/posts
2 Accept: application/json
3 Content-Type: application/json
4 Content-Length: 57
5
6 {"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 8-2

```
1 HTTP/1.1 201 Created
2 Content-Type: application/json
3 Content-Length: 65
4 Connection: close
5
6 {"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 an application before middleware.

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

Listing 8-3

```
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 8-4

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

$app->post('/blog/posts', function (Request $request) use ($app) {
    $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 8-5

```
$ 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 9
Using PdoSessionStorage to store Sessions in the Database

By default, the SessionServiceProvider writes session information in files using Symfony 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.

Symfony'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.

With a dedicated PDO service

Listing 9-1

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

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

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

```
};

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

Using the DoctrineServiceProvider

When using the DoctrineServiceProvider You don't have to make another database connection, simply pass the getWrappedConnection method.

Listing 9-2

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

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

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

$app['session.storage.handler'] = function () use ($app) {
    return new PdoSessionHandler(
        $app['db']->getWrappedConnection(),
        $app['session.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_lifetime`: Lifetime column (INTEGER)
- `session_time`: Time column (INTEGER)

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

3. http://symfony.com/doc/current/cookbook/configuration/pdo_session_storage.html#example-sql-statements

Chapter 10

Disabling CSRF Protection on a Form using the FormExtension

The *FormExtension* provides a service for building form in your application with the Symfony Form component. When the CSRF Service Provider is registered, 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 *Symfony 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 10-1

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

That's it, your form could be submitted 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 Symfony `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 11
Using 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:

Listing 11-1
```
composer require symfony/yaml
```

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

Listing 11-2
```
$app->register(new ValidatorServiceProvider());

$app['validator.mapping.class_metadata_factory'] = new
Symfony\Component\Validator\Mapping\Factory\LazyLoadingMetadataFactory(
    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 11-3
```
# validation.yml
Post:
  properties:
    title:
      - NotNull: ~
      - NotBlank: ~
    body:
      - Min: 100
```

Chapter 12
Making sub-Requests

Since Silex is based on the `HttpKernelInterface`, it allows you to simulate requests against your application. This means that you can embed a page within another, it also allows you to forward a request which is essentially an internal redirect that does not change the URL.

Basics

You can make a sub-request by calling the `handle` method on the `Application`. This method takes three arguments:

- **`$request`: An instance of the `Request` class which represents the** HTTP request.
- `$type`: Must be either `HttpKernelInterface::MASTER_REQUEST` or `HttpKernelInterface::SUB_REQUEST`. Certain listeners are only executed for the master request, so it's important that this is set to `SUB_REQUEST`.
- `$catch`: Catches exceptions and turns them into a response with status code `500`. This argument defaults to `true`. For sub-requests you will most likely want to set it to `false`.

By calling `handle`, you can make a sub-request manually. Here's an example:

Listing 12-1
```php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\HttpKernelInterface;

$subRequest = Request::create('/');
$response = $app->handle($subRequest, HttpKernelInterface::SUB_REQUEST, false);
```

There's some more things that you need to keep in mind though. In most cases you will want to forward some parts of the current master request to the sub-request like cookies, server information, or the session.

Here is a more advanced example that forwards said information (`$request` holds the master request):

Listing 12-2
```php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\HttpKernelInterface;

$subRequest = Request::create('/', 'GET', array(), $request->cookies->all(), array(), $request->server->all());
```

```
5  if ($request->getSession()) {
6      $subRequest->setSession($request->getSession());
7  }
8
9  $response = $app->handle($subRequest, HttpKernelInterface::SUB_REQUEST, false);
```

To forward this response to the client, you can simply return it from a controller:

Listing 12-3
```
1  use Silex\Application;
2  use Symfony\Component\HttpFoundation\Request;
3  use Symfony\Component\HttpKernel\HttpKernelInterface;
4
5  $app->get('/foo', function (Application $app, Request $request) {
6      $subRequest = Request::create('/', ...);
7      $response = $app->handle($subRequest, HttpKernelInterface::SUB_REQUEST, false);
8
9      return $response;
10 });
```

If you want to embed the response as part of a larger page you can call `Response::getContent`:

Listing 12-4
```
1  $header = ...;
2  $footer = ...;
3  $body = $response->getContent();
4
5  return $header.$body.$footer;
```

Rendering pages in Twig templates

The TwigServiceProvider provides a `render` function that you can use in Twig templates. It gives you a convenient way to embed pages.

Listing 12-5
```
1  {{ render('/sidebar') }}
```

For details, refer to the TwigServiceProvider docs.

Edge Side Includes

You can use ESI either through the HttpCacheServiceProvider or a reverse proxy cache such as Varnish. This also allows you to embed pages, however it also gives you the benefit of caching parts of the page.

Here is an example of how you would embed a page via ESI:

Listing 12-6
```
1  <esi:include src="/sidebar" />
```

For details, refer to the HttpCacheServiceProvider docs.

Dealing with the request base URL

One thing to watch out for is the base URL. If your application is not hosted at the webroot of your web server, then you may have an URL like `http://example.org/foo/index.php/articles/42`.

In this case, `/foo/index.php` is your request base path. Silex accounts for this path prefix in the routing process, it reads it from `$request->server`. In the context of sub-requests this can lead to

issues, because if you do not prepend the base path the request could mistake a part of the path you want to match as the base path and cut it off.

You can prevent that from happening by always prepending the base path when constructing a request:

Listing 12-7
```
$url = $request->getUriForPath('/');
$subRequest = Request::create($url, 'GET', array(), $request->cookies->all(), array(), $request->server->all());
```

This is something to be aware of when making sub-requests by hand.

Services depending on the Request

The container is a concept that is global to a Silex application, since the application object **is** the container. Any request that is run against an application will re-use the same set of services. Since these services are mutable, code in a master request can affect the sub-requests and vice versa. Any services depending on the `request` service will store the first request that they get (could be master or sub-request), and keep using it, even if that request is already over.

Instead of injecting the `request` service, you should always inject the `request_stack` one instead.

Chapter 13
Converting Errors to Exceptions

Silex catches exceptions that are thrown from within a request/response cycle. However, it does *not* catch PHP errors and notices. This recipe tells you how to catch them by converting them to exceptions.

Registering the ErrorHandler

The `Symfony/Debug` package has an `ErrorHandler` class that solves this problem. It converts all errors to exceptions, and exceptions are then caught by Silex.

Register it by calling the static `register` method:

Listing 13-1
```
use Symfony\Component\Debug\ErrorHandler;

ErrorHandler::register();
```

It is recommended that you do this as early as possible.

Handling fatal errors

To handle fatal errors, you can additionally register a global `ExceptionHandler`:

Listing 13-2
```
use Symfony\Component\Debug\ExceptionHandler;

ExceptionHandler::register();
```

In production you may want to disable the debug output by passing `false` as the `$debug` argument:

Listing 13-3
```
use Symfony\Component\Debug\ExceptionHandler;

ExceptionHandler::register(false);
```

Chapter 14
Using multiple Monolog Loggers

Having separate instances of Monolog for different parts of your system is often desirable and allows you to configure them independently, allowing for fine grained control of where your logging goes and in what detail.

This simple example allows you to quickly configure several monolog instances, using the bundled handler, but each with a different channel.

Listing 14-1
```php
$app['monolog.factory'] = $app->protect(function ($name) use ($app) {
    $log = new $app['monolog.logger.class']($name);
    $log->pushHandler($app['monolog.handler']);

    return $log;
});

foreach (array('auth', 'payments', 'stats') as $channel) {
    $app['monolog.'.$channel] = function ($app) use ($channel) {
        return $app['monolog.factory']($channel);
    };
}
```

As your application grows, or your logging needs for certain areas of the system become apparent, it should be straightforward to then configure that particular service separately, including your customizations.

Listing 14-2
```php
use Monolog\Handler\StreamHandler;

$app['monolog.payments'] = function ($app) {
    $log = new $app['monolog.logger.class']('payments');
    $handler = new StreamHandler($app['monolog.payments.logfile'], $app['monolog.payment.level']);
    $log->pushHandler($handler);

    return $log;
};
```

Alternatively, you could attempt to make the factory more complicated, and rely on some conventions, such as checking for an array of handlers registered with the container with the channel name, defaulting to the bundled handler.

Listing 14-3

```php
use Monolog\Handler\StreamHandler;
use Monolog\Logger;

$app['monolog.factory'] = $app->protect(function ($name) use ($app) {
    $log = new $app['monolog.logger.class']($name);

    $handlers = isset($app['monolog.'.$name.'.handlers'])
        ? $app['monolog.'.$name.'.handlers']
        : array($app['monolog.handler']);

    foreach ($handlers as $handler) {
        $log->pushHandler($handler);
    }

    return $log;
});

$app['monolog.payments.handlers'] = function ($app) {
    return array(
        new StreamHandler(__DIR__.'/../payments.log', Logger::DEBUG),
    );
};
```

Chapter 15

How to Create a Custom Authentication System with Guard

Whether you need to build a traditional login form, an API token authentication system or you need to integrate with some proprietary single-sign-on system, the Guard component can make it easy... and fun!

In this example, you'll build an API token authentication system and learn how to work with Guard.

Step 1) Create the Authenticator Class

Suppose you have an API where your clients will send an X-AUTH-TOKEN header on each request. This token is composed of the username followed by a password, separated by a colon (e.g. `X-AUTH-TOKEN: coolguy:awesomepassword`). Your job is to read this, find the associated user (if any) and check the password.

To create a custom authentication system, just create a class and make it implement GuardAuthenticatorInterface. Or, extend the simpler AbstractGuardAuthenticator. This requires you to implement six methods:

Listing 15-1

```php
<?php

namespace App\Security;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Guard\AbstractGuardAuthenticator;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;

class TokenAuthenticator extends AbstractGuardAuthenticator
{
    private $encoderFactory;

    public function __construct(EncoderFactoryInterface $encoderFactory)
    {
```

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

    public function getCredentials(Request $request)
    {
        // Checks if the credential header is provided
        if (!$token = $request->headers->get('X-AUTH-TOKEN')) {
            return;
        }

        // Parse the header or ignore it if the format is incorrect.
        if (false === strpos($token, ':')) {
            return;
        }
        list($username, $secret) = explode(':', $token, 2);

        return array(
            'username' => $username,
            'secret' => $secret,
        );
    }

    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        return $userProvider->loadUserByUsername($credentials['username']);
    }

    public function checkCredentials($credentials, UserInterface $user)
    {
        // check credentials - e.g. make sure the password is valid
        // return true to cause authentication success

        $encoder = $this->encoderFactory->getEncoder($user);

        return $encoder->isPasswordValid(
            $user->getPassword(),
            $credentials['secret'],
            $user->getSalt()
        );
    }

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
    {
        // on success, let the request continue
        return;
    }

    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        $data = array(
            'message' => strtr($exception->getMessageKey(), $exception->getMessageData()),

            // or to translate this message
            // $this->translator->trans($exception->getMessageKey(), $exception->getMessageData())
        );

        return new JsonResponse($data, 403);
    }

    /**
     * Called when authentication is needed, but it's not sent
     */
    public function start(Request $request, AuthenticationException $authException = null)
    {
        $data = array(
            // you might translate this message
            'message' => 'Authentication Required',
        );

        return new JsonResponse($data, 401);
    }
```

```
90
91      public function supportsRememberMe()
92      {
93          return false;
94      }
95  }
```

Step 2) Configure the Authenticator

To finish this, register the class as a service:

Listing 15-2
```
1  $app['app.token_authenticator'] = function ($app) {
2      return new App\Security\TokenAuthenticator($app['security.encoder_factory']);
3  };
```

Finally, configure your *security.firewalls* key to use this authenticator:

Listing 15-3
```
1  $app['security.firewalls'] = array(
2      'main' => array(
3          'guard' => array(
4              'authenticators' => array(
5                  'app.token_authenticator'
6              ),
7
8              // Using more than 1 authenticator, you must specify
9              // which one is used as entry point.
10             // 'entry_point' => 'app.token_authenticator',
11         ),
12         // configure where your users come from. Hardcode them, or load them from somewhere
13         // http://silex.sensiolabs.org/doc/providers/security.html#defining-a-custom-user-provider
14         'users' => array(
15             'victoria' => array('ROLE_USER', 'randomsecret'),
16         ),
17         // 'anonymous' => true
18     ),
19 );
```

You can use many authenticators, they are executed by the order they are configured.

You did it! You now have a fully-working API token authentication system. If your homepage required ROLE_USER, then you could test it under different conditions:

Listing 15-4
```
1  # test with no token
2  curl http://localhost:8000/
3  # {"message":"Authentication Required"}
4
5  # test with a bad token
6  curl -H "X-AUTH-TOKEN: alan" http://localhost:8000/
7  # {"message":"Username could not be found."}
8
9  # test with a working token
10 curl -H "X-AUTH-TOKEN: victoria:randomsecret" http://localhost:8000/
11 # the homepage controller is executed: the page loads normally
```

For more details read the Symfony cookbook entry on *How to Create a Custom Authentication System with Guard*[1].

1. http://symfony.com/doc/current/cookbook/security/guard-authentication.html

Chapter 16
Internals

This chapter will tell you how Silex works internally.

Silex

Application

The application is the main interface to Silex. It implements Symfony'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 Symfony *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 middlewares and errors.

Controller

The Symfony *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 setting 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`.

Instead of creating an instance of `RouteCollection` yourself, use the `$app['controllers_factory']` factory instead.

Symfony

Following Symfony 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 17
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];
- Make your feature addition or bug fix;
- Add tests for it;
- Optionally, add some documentation;
- *Send a pull request*[2], to the correct target branch (1.3 for bug fixes, master for new features).

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

1. https://github.com/silexphp/Silex
2. https://help.github.com/articles/creating-a-pull-request

Chapter 18
Writing Documentation

The documentation is written in *reStructuredText*[3] and can be generated using *sphinx*[4].

Listing 18-1
```
$ cd doc
$ sphinx-build -b html . build
```

3. http://docutils.sourceforge.net/rst.html
4. http://sphinx-doc.org

Chapter 19
Twig

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*[2] for more information.
- **twig.form.templates** (optional): An array of templates used to render forms (only available when the `FormServiceProvider` is enabled). The default theme is `form_div_layout.html.twig`, but you can use the other built-in themes: `form_table_layout.html.twig`, `bootstrap_3_layout.html.twig`, and `bootstrap_3_horizontal_layout.html.twig`.

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 19-1
```
$app->register(new Silex\Provider\TwigServiceProvider(), array(
    'twig.path' => __DIR__.'/views',
));
```

1. http://twig.sensiolabs.org/
2. http://twig.sensiolabs.org/doc/api.html#environment-options

Add Twig as a dependency:

Listing 19-2
```
composer require twig/twig
```

Usage

The Twig provider provides a `twig` service that can render templates:

Listing 19-3
```
$app->get('/hello/{name}', function ($name) use ($app) {
    return $app['twig']->render('hello.twig', array(
        'name' => $name,
    ));
});
```

Symfony Components Integration

Symfony provides a Twig bridge that provides additional integration between some Symfony components and Twig. Add it as a dependency:

Listing 19-4
```
composer require symfony/twig-bridge
```

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

- Access to the `path()` and `url()` functions. You can find more information in the *Symfony Routing documentation*[3]:

Listing 19-5
```
{{ path('homepage') }}
{{ url('homepage') }} {# generates the absolute url http://example.org/ #}
{{ path('hello', {name: 'Fabien'}) }}
{{ url('hello', {name: 'Fabien'}) }} {# generates the absolute url http://example.org/hello/Fabien #}
```

- Access to the `absolute_url()` and `relative_path()` Twig functions.

Translations Support

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 *Symfony Translation documentation*[4].

Form Support

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 Symfony Forms reference.

Security Support

If you are using the `SecurityServiceProvider`, you will have access to the `is_granted()` function in templates. You can find more information in the *Symfony Security documentation*[5].

3. http://symfony.com/doc/current/book/routing.html#generating-urls-from-a-template
4. http://symfony.com/doc/current/book/translation.html#twig-templates
5. http://symfony.com/doc/current/book/security.html#access-control-in-templates

Global Variable

When the Twig bridge is available, the `global` variable refers to an instance of *AppVariable*[6]. It gives access to the following methods:

Listing 19-6
```
{# The current Request #}
{{ global.request }}

{# The current User (when security is enabled) #}
{{ global.user }}

{# The current Session #}
{{ global.session }}

{# The debug flag #}
{{ global.debug }}
```

Rendering a Controller

A `render` function is also registered to help you render another controller from a template (available when the HttpFragment Service Provider is registered):

Listing 19-7
```
{{ render(url('sidebar')) }}

{# or you can reference a controller directly without defining a route for it #}
{{ render(controller(controller)) }}
```

You must prepend the `app.request.baseUrl` to render calls to ensure that the render works when deployed into a sub-directory of the docroot.

Read the Twig *reference*[7] for Symfony document to learn more about the various Twig functions.

Traits

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

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

Listing 19-8
```
return $app->render('index.html', ['name' => 'Fabien']);

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

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

Listing 19-9
```
// stream a view
use Symfony\Component\HttpFoundation\StreamedResponse;

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

6. http://api.symfony.com/master/Symfony/Bridge/Twig/AppVariable.html
7. https://symfony.com/doc/current/reference/twig_reference.html#controller

- **renderView**: Renders a view with the given parameters and returns a string.

Listing 19-10
```php
$content = $app->renderView('index.html', ['name' => 'Fabien']);
```

Customization

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

Listing 19-11
```php
$app->extend('twig', function($twig, $app) {
    $twig->addGlobal('pi', 3.14);
    $twig->addFilter('levenshtein', new \Twig_Filter_Function('levenshtein'));

    return $twig;
});
```

For more information, check out the *official Twig documentation*[8].

8. http://twig.sensiolabs.org

Chapter 20
Asset

The *AssetServiceProvider* provides a way to manage URL generation and versioning of web assets such as CSS stylesheets, JavaScript files and image files.

Parameters

- **assets.version**: Default version for assets.
- **assets.format_version** (optional): Default format for assets.
- **assets.named_packages** (optional): Named packages. Keys are the package names and values the configuration (supported keys are `version`, `version_format`, `base_urls`, and `base_path`).

Services

- **assets.packages**: The asset service.

Registering

Listing 20-1
```php
$app->register(new Silex\Provider\AssetServiceProvider(), array(
    'assets.version' => 'v1',
    'assets.version_format' => '%s?version=%s',
    'assets.named_packages' => array(
        'css' => array('version' => 'css2', 'base_path' => '/whatever-makes-sense'),
        'images' => array('base_urls' => array('https://img.example.com')),
    ),
));
```

Add the Symfony Asset Component as a dependency:

Listing 20-2
```
composer require symfony/asset
```

If you want to use assets in your Twig templates, you must also install the Symfony Twig Bridge:

Listing 20-3
```
composer require symfony/twig-bridge
```

Usage

The AssetServiceProvider is mostly useful with the Twig provider:

Listing 20-4
```
{{ asset('/css/foo.png') }}
{{ asset('/css/foo.css', 'css') }}
{{ asset('/img/foo.png', 'images') }}

{{ asset_version('/css/foo.png') }}
```

For more information, check out the *Asset Component documentation*[1].

1. https://symfony.com/doc/current/components/asset/introduction.html

Chapter 21
Monolog

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 logging to your application. This allows you to debug and monitor the behaviour, even in production.

Parameters

- **monolog.logfile**: File where logs are written to.
- **monolog.bubble**: (optional) Whether the messages that are handled can bubble up the stack or not.
- **monolog.permission**: (optional) File permissions default (null), nothing change.
- **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.

  In addition to the `Logger::` constants, it is also possible to supply the level in string form, for example: `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"`.
- **monolog.name** (optional): Name of the monolog channel, defaults to `myapp`.
- **monolog.exception.logger_filter** (optional): An anonymous function that returns an error level for on uncaught exception that should be logged.
- **monolog.use_error_handler** (optional): Whether errors and uncaught exceptions should be handled by the Monolog `ErrorHandler` class and added to the log. By default the error handler is enabled unless the application `debug` parameter is set to true.

  Please note that enabling the error handler may silence some errors, ignoring the PHP `display_errors` configuration setting.

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

Services

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

  Example usage:

Listing 21-1
```
$app['monolog']->debug('Testing the Monolog logging.');
```

- **monolog.listener**: An event listener to log requests, responses and errors.

Registering

Listing 21-2
```
$app->register(new Silex\Provider\MonologServiceProvider(), array(
    'monolog.logfile' => __DIR__.'/development.log',
));
```

Add Monolog as a dependency:

Listing 21-3
```
composer require monolog/monolog
```

Usage

The MonologServiceProvider provides a `monolog` service. You can use it to add log entries for any logging level through `debug()`, `info()`, `warning()` and `error()`:

Listing 21-4
```
use Symfony\Component\HttpFoundation\Response;

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

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

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

Customization

You can configure Monolog (like adding or changing the handlers) before using it by extending the `monolog` service:

Listing 21-5
```
$app->extend('monolog', function($monolog, $app) {
    $monolog->pushHandler(...);

    return $monolog;
});
```

By default, all requests, responses and errors are logged by an event listener registered as a service called *monolog.listener*. You can replace or remove this service if you want to modify or disable the logged information.

Traits

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

- **log**: Logs a message.

Listing 21-6
```
$app->log(sprintf("User '%s' registered.", $username));
```

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

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

Chapter 22
Session

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

Parameters

- **session.storage.save_path** (optional): The path for the `NativeFileSessionHandler`, 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 most useful options are:

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

  However, all of these are optional. Default Sessions life time is 1800 seconds (30 minutes). To override this, set the `lifetime` option.

  For a full list of available options, read the *PHP*[2] official documentation.
- **session.test**: Whether to simulate sessions or not (useful when writing functional tests).

Services

- **session**: An instance of Symfony's *Session*[3].

1. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session/Storage/NativeSessionStorage.html
2. http://php.net/session.configuration
3. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session/Session.html

- **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 *NativeFileSessionHandler*[4] storage handler.

Registering

Listing 22-1
```php
$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 them:

Listing 22-2
```php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$app->get('/login', function (Request $request) use ($app) {
    $username = $request->server->get('PHP_AUTH_USER', false);
    $password = $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']}!";
});
```

Custom Session Configurations

If your system is using a custom session configuration (such as a redis handler from a PHP extension) then you need to disable the NativeFileSessionHandler by setting `session.storage.handler` to null. You will have to configure the `session.save_path` ini setting yourself in that case.

Listing 22-3
```php
$app['session.storage.handler'] = null;
```

4. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session/Storage/Handler/NativeFileSessionHandler.html

Chapter 23
Swiftmailer

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.use_spool**: A boolean to specify whether or not to use the memory spool, defaults to true.
- **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. Valid values are 'tls', 'ssl', or null (indicating no encryption).
  - **auth_mode**: SMTP authentication mode, defaults to null. Valid values are 'plain', 'login', 'cram-md5', or null.

  Example usage:

Listing 23-1
```php
$app['swiftmailer.options'] = array(
    'host' => 'host',
    'port' => '25',
    'username' => 'username',
    'password' => 'password',
    'encryption' => null,
    'auth_mode' => null
);
```

1. http://swiftmailer.org

- **swiftmailer.sender_address**: If set, all messages will be delivered with this address as the "return path" address.
- **swiftmailer.delivery_addresses**: If not empty, all email messages will be sent to those addresses instead of being sent to their actual recipients. This is often useful when developing.
- **swiftmailer.delivery_whitelist**: Used in combination with `delivery_addresses`. If set, emails matching any of these patterns will be delivered like normal, as well as being sent to `delivery_addresses`.
- **swiftmailer.plugins**: Array of SwiftMailer plugins.

  Example usage:

Listing 23-2
```
$app['swiftmailer.plugins'] = function ($app) {
    return array(
        new \Swift_Plugins_PopBeforeSmtpPlugin('pop3.example.com'),
    );
};
```

Services

- **mailer**: The mailer instance.

  Example usage:

Listing 23-3
```
$message = \Swift_Message::newInstance();

// ...

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

- **swiftmailer.transport**: The transport used for e-mail delivery. Defaults to a `Swift_Transport_EsmtpTransport`.
- **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 23-4
```
$app->register(new Silex\Provider\SwiftmailerServiceProvider());
```

Add SwiftMailer as a dependency:

Listing 23-5
```
composer require swiftmailer/swiftmailer
```

Usage

The Swiftmailer provider provides a `mailer` service:

Listing 23-6

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

$app->post('/feedback', function (Request $request) use ($app) {
    $message = \Swift_Message::newInstance()
        ->setSubject('[YourSite] Feedback')
        ->setFrom(array('noreply@yoursite.com'))
        ->setTo(array('feedback@yoursite.com'))
        ->setBody($request->get('message'));

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

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

Usage in commands

By default, the Swiftmailer provider sends the emails using the `KernelEvents::TERMINATE` event, which is fired after the response has been sent. However, as this event isn't fired for console commands, your emails won't be sent.

For that reason, if you send emails using a command console, it is recommended that you disable the use of the memory spool (before accessing `$app['mailer']`):

Listing 23-7
```php
$app['swiftmailer.use_spool'] = false;
```

Alternatively, you can just make sure to flush the message spool by hand before ending the command execution. To do so, use the following code:

Listing 23-8
```php
$app['swiftmailer.spooltransport']
    ->getSpool()
    ->flushQueue($app['swiftmailer.transport'])
;
```

Traits

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

- **mail**: Sends an email.

Listing 23-9
```php
$app->mail(\Swift_Message::newInstance()
    ->setSubject('[YourSite] Feedback')
    ->setFrom(array('noreply@yoursite.com'))
    ->setTo(array('feedback@yoursite.com'))
    ->setBody($request->get('message')));
```

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

2. http://swiftmailer.org

Chapter 24
Locale

The *LocaleServiceProvider* manages the locale of an application.

Parameters

- **locale**: 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.

Services

- n/a

Registering

Listing 24-1
```
1 $app->register(new Silex\Provider\LocaleServiceProvider());
```

Chapter 25
Translation

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_fallbacks** (optional): Fallback locales for the translator. It will be used when the current locale has no messages set. Defaults to `en`.

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 25-1
```php
$app->register(new Silex\Provider\LocaleServiceProvider());
$app->register(new Silex\Provider\TranslationServiceProvider(), array(
    'locale_fallbacks' => array('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

Add the Symfony Translation Component as a dependency:

Listing 25-2
```
composer require symfony/translation
```

Usage

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

Listing 25-3
```
$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).

Using Resources

When translations are stored in a file, you can load them as follows:

Listing 25-4
```
$app = new Application();

$app->register(new TranslationServiceProvider());
$app->extend('translator.resources', function ($resources, $app) {
    $resources = array_merge($resources, array(
        array('array', array('This value should be a valid number.' => 'Cette valeur doit être un nombre.'),
'fr', 'validators'),
    ));

    return $resources;
});
```

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.

Listing 25-5
```
$app->trans('Hello World');

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

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 Symfony `Config` and `Yaml` components as dependencies:

Listing 25-6
```
composer require symfony/config symfony/yaml
```

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:

Listing 25-7
```
hello: Hello %name%
goodbye: Goodbye %name%
```

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

Listing 25-8
```
use Symfony\Component\Translation\Loader\YamlFileLoader;

$app->extend('translator', function($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;
});
```

XLIFF-based language files

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

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

Listing 25-9
```
$translator->addResource('xliff', __DIR__.'/locales/en.xlf', 'en');
$translator->addResource('xliff', __DIR__.'/locales/de.xlf', 'de');
$translator->addResource('xliff', __DIR__.'/locales/fr.xlf', 'fr');
```

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 when using the Twig bridge provided by Symfony (see TwigServiceProvider):

Listing 25-10
```
{{ 'translation_key'|trans }}
{{ 'translation_key'|transchoice }}
{% trans %}translation_key{% endtrans %}
```

Chapter 26
Validator

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

Parameters

- **validator.validator_service_ids**: An array of service names representing validators.

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.

Registering

Listing 26-1
```
1  $app->register(new Silex\Provider\ValidatorServiceProvider());
```

Add the Symfony Validator Component as a dependency:

Listing 26-2
```
1  composer require symfony/validator
```

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

Usage

The Validator provider provides a `validator` service.

Validating Values

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

Listing 26-3
```
use Symfony\Component\Validator\Constraints as Assert;

$app->get('/validate/{email}', function ($email) use ($app) {
    $errors = $app['validator']->validate($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 26-4
```
use Symfony\Component\Validator\Constraints as Assert;

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

$constraint = new Assert\Collection(array(
    'title' => new Assert\Length(array('min' => 10)),
    'author' => new Assert\Collection(array(
        'first_name' => array(new Assert\NotBlank(), new Assert\Length(array('min' => 10))),
        'last_name'  => new Assert\Length(array('min' => 10)),
    )),
));
$errors = $app['validator']->validate($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 26-5
```
use Symfony\Component\Validator\Constraints as Assert;

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

```
}

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

$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']->getMetadataFor('Author');
$metadata->addPropertyConstraint('first_name', new Assert\NotBlank());
$metadata->addPropertyConstraint('first_name', new Assert\Length(array('min' => 10)));
$metadata->addPropertyConstraint('last_name', new Assert\Length(array('min' => 10)));

$metadata = $app['validator.mapping.class_metadata_factory']->getMetadataFor('Book');
$metadata->addPropertyConstraint('title', new Assert\Length(array('min' => 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 26-6

```
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\Length(array('min' => 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\Length(array('min' => 10)));
        $metadata->addPropertyConstraint('last_name', new Assert\Length(array('min' => 10)));
    }
}

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

```
33
34      $book = new Book();
35      $book->title = 'My Book';
36      $book->author = $author;
37
38      $errors = $app['validator']->validate($book);
39
40      if (count($errors) > 0) {
41          foreach ($errors as $error) {
42              echo $error->getPropertyPath().' '.$error->getMessage()."\n";
43          }
44      } else {
45          echo 'The author is valid';
46      }
47  });
```

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 26-7

```
1  $app['translator.domains'] = array(
2      'validators' => array(
3          'fr' => array(
4              'This value should be a valid number.' => 'Cette valeur doit être un nombre.',
5          ),
6      ),
7  );
```

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

2. http://symfony.com/doc/master/book/validation.html

Chapter 27
Form

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

Parameters

- none

Services

- **form.factory**: An instance of *FormFactory*[1], that is used to build a form.

Registering

Listing 27-1
```
use Silex\Provider\FormServiceProvider;

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

If you don't want to create your own form layout, it's fine: a default one will be used. But you will have to register the translation provider as the default form layout requires it:

Listing 27-2
```
$app->register(new Silex\Provider\TranslationServiceProvider(), array(
    'translator.domains' => array(),
));
```

If you want to use validation with forms, do not forget to register the Validator provider.

1. http://api.symfony.com/master/Symfony/Component/Form/FormFactory.html

Add the Symfony Form Component as a dependency:

Listing 27-3
```
composer require symfony/form
```

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

Listing 27-4
```
composer require symfony/validator symfony/config symfony/translation
```

If you want to use forms in your Twig templates, you can also install the Symfony Twig Bridge. Make sure to install, if you didn't do that already, the Translation component in order for the bridge to work:

Listing 27-5
```
composer require symfony/twig-bridge symfony/config symfony/translation
```

Usage

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

Listing 27-6
```
use Symfony\Component\Form\Extension\Core\Type\FormType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;

$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(FormType::class, $data)
        ->add('name')
        ->add('email')
        ->add('billing_plan', ChoiceType::class, array(
            'choices' => array(1 => 'free', 2 => 'small_business', 3 => 'corporate'),
            'expanded' => true,
        ))
        ->getForm();

    $form->handleRequest($request);

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

        // do something with the data

        // redirect somewhere
        return $app->redirect('...');
    }

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

And here is the `index.twig` form template (requires `symfony/twig-bridge`):

Listing 27-7
```
<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 27-8
```
use Symfony\Component\Form\Extension\Core\Type\FormType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Validator\Constraints as Assert;

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

$form = $app['form.factory']->createBuilder(FormType::class)
    ->add('name', TextType::class, array(
        'constraints' => array(new Assert\NotBlank(), new Assert\Length(array('min' => 5)))
    ))
    ->add('email', TextType::class, array(
        'constraints' => new Assert\Email()
    ))
    ->add('billing_plan', ChoiceType::class, array(
        'choices' => array(1 => 'free', 2 => 'small_business', 3 => 'corporate'),
        'expanded' => true,
        'constraints' => new Assert\Choice(array(1, 2, 3)),
    ))
    ->getForm();
```

You can register form types by extending `form.types`:

Listing 27-9
```
$app['your.type.service'] = function ($app) {
    return new YourServiceFormType();
};
$app->extend('form.types', function ($types) use ($app) {
    $types[] = new YourFormType();
    $types[] = 'your.type.service';

    return $types;
}));
```

You can register form extensions by extending `form.extensions`:

Listing 27-10
```
$app->extend('form.extensions', function ($extensions) use ($app) {
    $extensions[] = new YourTopFormExtension();

    return $extensions;
});
```

You can register form type extensions by extending `form.type.extensions`:

Listing 27-11
```
$app['your.type.extension.service'] = function ($app) {
    return new YourServiceFormTypeExtension();
};
$app->extend('form.type.extensions', function ($extensions) use ($app) {
    $extensions[] = new YourFormTypeExtension();
    $extensions[] = 'your.type.extension.service';

    return $extensions;
});
```

You can register form type guessers by extending `form.type.guessers`:

Listing 27-12
```
$app['your.type.guesser.service'] = function ($app) {
    return new YourServiceFormTypeGuesser();
};
$app->extend('form.type.guessers', function ($guessers) use ($app) {
    $guessers[] = new YourFormTypeGuesser();
```

```
    $guessers[] = 'your.type.guesser.service';

    return $guessers;
});
```

CSRF protection is only available and automatically enabled when the CSRF Service Provider is registered.

Traits

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

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

Listing 27-13
```
$app->form($data);
```

For more information, consult the *Symfony Forms documentation*[2].

2. http://symfony.com/doc/2.8/book/forms.html

Chapter 28
CSRF

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

Parameters

- none

Services

- **csrf.token_manager**: An instance of an implementation of the *CsrfProviderInterface*[1], defaults to a *DefaultCsrfProvider*[2].

Registering

Listing 28-1
```
1  use Silex\Provider\CsrfServiceProvider;
2
3  $app->register(new CsrfServiceProvider());
```

Add the Symfony's *Security CSRF Component*[3] as a dependency:

Listing 28-2
```
1  composer require symfony/security-csrf
```

1. http://api.symfony.com/master/Symfony/Component/Form/Extension/Csrf/CsrfProvider/CsrfProviderInterface.html
2. http://api.symfony.com/master/Symfony/Component/Form/Extension/Csrf/CsrfProvider/DefaultCsrfProvider.html
3. http://symfony.com/doc/current/components/security/index.html

Usage

When the CSRF Service Provider is registered, all forms created via the Form Service Provider are protected against CSRF by default.

You can also use the CSRF protection even without using the Symfony Form component. If, for example, you're doing a DELETE action, you can check the CSRF token:

Listing 28-3
```
use Symfony\Component\Security\Csrf\CsrfToken;

$app['csrf.token_manager']->isTokenValid(new CsrfToken('token_id', 'TOKEN'));
```

Chapter 29
HTTP Cache

The *HttpCacheServiceProvider* provides support for the Symfony 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].
- **http_cache.esi**: An instance of *Esi*[3], that implements the ESI capabilities to Request and Response instances.
- **http_cache.store**: An instance of *Store*[4], that implements all the logic for storing cache metadata (Request and Response headers).

Registering

Listing 29-1
```
$app->register(new Silex\Provider\HttpCacheServiceProvider(), array(
    'http_cache.cache_dir' => __DIR__.'/cache/',
));
```

1. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpCache/HttpCache.html
2. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpCache/HttpCache.html
3. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpCache/Esi.html
4. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpCache/Store.html

Usage

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

Listing 29-2
```
use Symfony\Component\HttpFoundation\Response;

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

If you want Silex to trust the `X-Forwarded-For*` headers from your reverse proxy at address $ip, you will need to whitelist it as documented in *Trusting Proxies*[5].

If you would be running Varnish in front of your application on the same machine:

Listing 29-3
```
use Symfony\Component\HttpFoundation\Request;

Request::setTrustedProxies(array('127.0.0.1', '::1'));
$app->run();
```

This provider allows you to use the Symfony reverse proxy natively with Silex applications by using the `http_cache` service. The Symfony reverse proxy acts much like any other proxy would, so you will want to whitelist it:

Listing 29-4
```
use Symfony\Component\HttpFoundation\Request;

Request::setTrustedProxies(array('127.0.0.1'));
$app['http_cache']->run();
```

The provider also provides ESI support:

Listing 29-5
```
$app->get('/', function() {
    $response = new Response(<<<EOF
<html>
    <body>
        Hello
        <esi:include src="/included" />
    </body>
</html>

EOF
    , 200, array(
        'Surrogate-Control' => 'content="ESI/1.0"',
    ));

    $response->setTtl(20);

    return $response;
});

$app->get('/included', function() {
    $response = new Response('Foo');
    $response->setTtl(5);

    return $response;
});

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

If your application doesn't use ESI, you can disable it to slightly improve the overall performance:

5. http://symfony.com/doc/current/components/http_foundation/trusting_proxies.html

Listing 29-6
```
$app->register(new Silex\Provider\HttpCacheServiceProvider(), array(
    'http_cache.cache_dir' => __DIR__.'/cache/',
    'http_cache.esi'       => null,
));
```

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 *Symfony HTTP Cache documentation*[6].

6. http://symfony.com/doc/current/book/http_cache.html

Chapter 30
HTTP Fragment

The *HttpFragmentServiceProvider* provides support for the Symfony fragment sub-framework, which allows you to embed fragments of HTML in a template.

Parameters

- **fragment.path**: The path to use for the URL generated for ESI and HInclude URLs (`/_fragment` by default).
- **uri_signer.secret**: The secret to use for the URI signer service (used for the HInclude renderer).
- **fragment.renderers.hinclude.global_template**: The content or Twig template to use for the default content when using the HInclude renderer.

Services

- **fragment.handler**: An instance of *FragmentHandler*[1].
- **fragment.renderers**: An array of fragment renderers (by default, the inline, ESI, and HInclude renderers are pre-configured).

Registering

Listing 30-1
```
1  $app->register(new Silex\Provider\HttpFragmentServiceProvider());
```

Usage

1. http://api.symfony.com/master/Symfony/Component/HttpKernel/Fragment/FragmentHandler.html

This section assumes that you are using Twig for your templates.

Instead of building a page out of a single request/controller/template, the fragment framework allows you to build a page from several controllers/sub-requests/sub-templates by using **fragments**.

Including "sub-pages" in the main page can be done with the Twig `render()` function:

Listing 30-2
```
The main page content.

{{ render('/foo') }}

The main page content resumes here.
```

The `render()` call is replaced by the content of the `/foo` URL (internally, a sub-request is handled by Silex to render the sub-page).

Instead of making internal sub-requests, you can also use the ESI (the sub-request is handled by a reverse proxy) or the HInclude strategies (the sub-request is handled by a web browser):

Listing 30-3
```
{{ render(url('route_name')) }}

{{ render_esi(url('route_name')) }}

{{ render_hinclude(url('route_name')) }}
```

Chapter 31
Security

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

Parameters

- **security.hide_user_not_found** (optional): Defines whether to hide user not found exception or not. Defaults to `true`.
- **security.encoder.bcrypt.cost** (optional): Defines BCrypt password encoder cost. Defaults to 13.

Services

- **security.token_storage**: Gives access to the user token.
- **security.authorization_checker**: Allows to check authorizations for the users.
- **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 (uses `security.default_encoder`).
- **security.default_encoder**: The encoder to use by default for all users (BCrypt).
- **security.encoder.digest**: Digest password encoder.
- **security.encoder.bcrypt**: BCrypt password encoder.
- **security.encoder.pbkdf2**: Pbkdf2 password encoder.
- **user**: Returns the current user

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`

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

Registering

Listing 31-1
```
$app->register(new Silex\Provider\SecurityServiceProvider(), array(
    'security.firewalls' => // see below
));
```

Add the Symfony Security Component as a dependency:

Listing 31-2
```
composer require symfony/security
```

If you're using a form to authenticate users, you need to enable `SessionServiceProvider`.

The security features are only available after the Application has been booted. So, if you want to use it outside of the handling of a request, don't forget to call `boot()` first:

Listing 31-3
```
$app->boot();
```

Usage

The Symfony Security component is powerful. To learn more about it, read the *Symfony 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 31-4
```
$token = $app['security.token_storage']->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 31-5
```
if (null !== $token) {
    $user = $token->getUser();
}
```

3. http://symfony.com/doc/2.8/book/security.html

The user can be a string, an 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 31-6
```
$app['security.firewalls'] = array(
    'admin' => array(
        'pattern' => '^/admin',
        'http' => true,
        'users' => array(
            // raw password is foo
            'admin' => array('ROLE_ADMIN', '$2y$10$3i9/lVd8UOFIJ6PAMFt8gu3/r5gOqeCJvoSlLCsvMTythye19F77a'),
        ),
    ),
);
```

The `pattern` is a regular expression on the URL path; the `http` setting tells the security layer to use HTTP basic authentication and the `users` entry defines valid users.

If you want to restrict the firewall by more than the URL pattern (like the HTTP method, the client IP, the hostname, or any Request attributes), use an instance of a *RequestMatcher*[5] for the `pattern` option:

Listing 31-7
```
use Symfony/Component/HttpFoundation/RequestMatcher;

$app['security.firewalls'] = array(
    'admin' => array(
        'pattern' => new RequestMatcher('^/admin', 'example.com', 'POST'),
        // ...
    ),
);
```

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 31-8
```
// 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]

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

If you are using php-cgi under Apache, you need to add this configuration to make things work correctly:

Listing 31-9
```
RewriteEngine On
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ app.php [QSA,L]
```

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 they are accessing a secured area without being authenticated so that they can enter their 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 31-10
```
$app['security.firewalls'] = array(
    'admin' => array(
        'pattern' => '^/admin/',
        'form' => array('login_path' => '/login', 'check_path' => '/admin/login_check'),
        'users' => array(
            'admin' => array('ROLE_ADMIN', '$2y$10$3i9/lVd8UOFIJ6PAMFt8gu3/r5gOqeCJvoSlLCsvMTythye19F77a'),
        ),
    ),
);
```

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.

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

Listing 31-11
```
use Symfony\Component\HttpFoundation\Request;

$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 31-12
```
<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 31-13
```php
$app['security.firewalls'] = array(
    'login' => array(
        'pattern' => '^/login$',
    ),
    'secured' => array(
        'pattern' => '^.*$',
        'form' => array('login_path' => '/login', 'check_path' => '/login_check'),
        'users' => array(
            'admin' => array('ROLE_ADMIN', '$2y$10$3i9/lVd8UOFIJ6PAMFt8gu3/r5g0qeCJvoSlLCsvMTythye19F77a'),
        ),
    ),
);
```

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.

You can toggle all registered authentication mechanisms for a particular area on and off with the `security` flag:

Listing 31-14
```php
$app['security.firewalls'] = array(
    'api' => array(
        'pattern' => '^/api',
        'security' => $app['debug'] ? false : true,
        'wsse' => true,

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

Adding a Logout

When using a form for authentication, you can let users log out if you add the `logout` setting, where `logout_path` must match the main firewall pattern:

Listing 31-15
```php
$app['security.firewalls'] = array(
    'secured' => array(
        'pattern' => '^/admin/',
        'form' => array('login_path' => '/login', 'check_path' => '/admin/login_check'),
        'logout' => array('logout_path' => '/admin/logout', 'invalidate_session' => true),

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

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

Listing 31-16
```
<a href="{{ path('admin_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 31-17
```
$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 31-18
```
if ($app['security.authorization_checker']->isGranted('ROLE_ADMIN')) {
    // ...
}
```

You can check roles in Twig templates too:

Listing 31-19
```
{% if is_granted('ROLE_ADMIN') %}
    <a href="/secured?_switch_user=fabien">Switch to Fabien</a>
{% endif %}
```

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

Listing 31-20
```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <a href="{{ path('logout') }}">Logout</a>
{% else %}
    <a href="{{ path('login') }}">Login</a>
{% endif %}
```

Of course you will need to define a `login` route for this to work.

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 31-21
```php
$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 31-22
```twig
{% 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 their primary account:

Listing 31-23
```twig
{% 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 31-24
```php
$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 31-25
```php
$app['security.access_rules'] = array(
    array('^/admin', 'ROLE_ADMIN', 'https'),
    array('^.*$', '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.

7. http://api.symfony.com/master/Symfony/Component/HttpFoundation/RequestMatcher.html

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

Listing 31-26
```
'users' => 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 31-27
```
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\UnsupportedUserException;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
use Doctrine\DBAL\Connection;

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)));
        }

        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 31-28
```
use Doctrine\DBAL\Schema\Table;

$schema = $app['db']->getSchemaManager();
if (!$schema->tablesExist('users')) {
    $users = new Table('users');
    $users->addColumn('id', 'integer', array('unsigned' => true, 'autoincrement' => true));
    $users->setPrimaryKey(array('id'));
    $users->addColumn('username', 'string', array('length' => 32));
    $users->addUniqueIndex(array('username'));
```

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

```
    $users->addColumn('password', 'string', array('length' => 255));
    $users->addColumn('roles', 'string', array('length' => 255));

    $schema->createTable($users);

    $app['db']->insert('users', array(
      'username' => 'fabien',
      'password' => '$2y$10$3i9/lVd8UOFIJ6PAMFt8gu3/r5g0qeCJvoSlLCsvMTythye19F77a',
      'roles' => 'ROLE_USER'
    ));

    $app['db']->insert('users', array(
      'username' => 'admin',
      'password' => '$2y$10$3i9/lVd8UOFIJ6PAMFt8gu3/r5g0qeCJvoSlLCsvMTythye19F77a',
      'roles' => '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 Encoder

By default, Silex uses the `BCrypt` algorithm to encode passwords. Additionally, the password is encoded multiple times. You can change these defaults by overriding `security.default_encoder` service to return one of the predefined encoders:

- **security.encoder.digest**: Digest password encoder.
- **security.encoder.bcrypt**: BCrypt password encoder.
- **security.encoder.pbkdf2**: Pbkdf2 password encoder.

Listing 31-29

```
$app['security.default_encoder'] = function ($app) {
    return $app['security.encoder.pbkdf2'];
};
```

Or you can define you own, fully customizable encoder:

Listing 31-30

```
use Symfony\Component\Security\Core\Encoder\PlaintextPasswordEncoder;

$app['security.default_encoder'] = function ($app) {
    // Plain text (e.g. for debugging)
    return new PlaintextPasswordEncoder();
};
```

You can change the default BCrypt encoding cost by overriding `security.encoder.bcrypt.cost`

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_listener.factory.XXX` where `XXX` is the name you want to use in your configuration:

Listing 31-31

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

    // define the authentication listener object
    $app['security.authentication_listener.'.$name.'.wsse'] = function () use ($app) {
        return new WsseListener($app['security.token_storage'], $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 31-32
```php
$app->register(new Silex\Provider\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].

The Guard component simplifies the creation of custom authentication providers. How to Create a Custom Authentication System with Guard

Stateless Authentication

By default, a session cookie is created to persist the security context of the user. However, if you use certificates, HTTP authentication, WSSE and so on, the credentials are sent for each request. In that case, you can turn off persistence by activating the `stateless` authentication flag:

Listing 31-33
```php
$app['security.firewalls'] = array(
    'default' => array(
        'stateless' => true,
        'wsse' => true,

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

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

Traits

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

- **encodePassword**: Encode a given password.

Listing 31-34
```
$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 31-35
```
$app->get('/', function () {
    // do something but only for admins
})->secure('ROLE_ADMIN');
```

The `Silex\Route\SecurityTrait` must be used with a user defined `Route` class, not the application.

Listing 31-36
```
use Silex\Route;

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

Listing 31-37
```
$app['route_class'] = 'MyRoute';
```

Chapter 32
Remember Me

The *RememberMeServiceProvider* adds "Remember-Me" authentication to the *SecurityServiceProvider*.

Parameters

n/a

Services

n/a

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

Registering

Before registering this service provider, you must register the *SecurityServiceProvider*:

Listing 32-1
```php
$app->register(new Silex\Provider\SecurityServiceProvider());
$app->register(new Silex\Provider\RememberMeServiceProvider());

$app['security.firewalls'] = array(
    'my-firewall' => array(
        'pattern'     => '^/secure$',
        'form'        => true,
        'logout'      => true,
        'remember_me' => array(
            'key'                => 'Choose_A_Unique_Random_Key',
            'always_remember_me' => true,
            /* Other options */
        ),
        'users' => array( /* ... */ ),
```

```
15         ),
16     );
```

Options

- **key**: A secret key to generate tokens (you should generate a random string).
- **name**: Cookie name (default: `REMEMBERME`).
- **lifetime**: Cookie lifetime (default: `31536000` ~ 1 year).
- **path**: Cookie path (default: `/`).
- **domain**: Cookie domain (default: `null` = request domain).
- **secure**: Cookie is secure (default: `false`).
- **httponly**: Cookie is HTTP only (default: `true`).
- **always_remember_me**: Enable remember me (default: `false`).
- **remember_me_parameter**: Name of the request parameter enabling remember_me on login. To add the checkbox to the login form. You can find more information in the *Symfony cookbook*[1] (default: `_remember_me`).

1. `http://symfony.com/doc/current/cookbook/security/remember_me.html`

Chapter 33
Serializer

The *SerializerServiceProvider* provides a service for serializing objects.

Parameters

None.

Services

- **serializer**: An instance of *Symfony\Component\Serializer\Serializer*[1].
- **serializer.encoders**: *Symfony\Component\Serializer\Encoder\JsonEncoder*[2] and *Symfony\Component\Serializer\Encoder\XmlEncoder*[3].
- **serializer.normalizers**: *Symfony\Component\Serializer\Normalizer\CustomNormalizer*[4] and *Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer*[5].

Registering

Listing 33-1
```
1  $app->register(new Silex\Provider\SerializerServiceProvider());
```

Add the Symfony's *Serializer Component*[6] as a dependency:

Listing 33-2
```
1  composer require symfony/serializer
```

1. http://api.symfony.com/master/Symfony/Component/Serializer/Serializer.html
2. http://api.symfony.com/master/Symfony/Component/Serializer/Encoder/JsonEncoder.html
3. http://api.symfony.com/master/Symfony/Component/Serializer/Encoder/XmlEncoder.html
4. http://api.symfony.com/master/Symfony/Component/Serializer/Normalizer/CustomNormalizer.html
5. http://api.symfony.com/master/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html
6. http://symfony.com/doc/current/components/serializer.html

Usage

The `SerializerServiceProvider` provider provides a `serializer` service:

Listing 33-3

```
use Silex\Application;
use Silex\Provider\SerializerServiceProvider;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$app = new Application();

$app->register(new SerializerServiceProvider());

// only accept content types supported by the serializer via the assert method.
$app->get("/pages/{id}.{_format}", function (Request $request, $id) use ($app) {
    // assume a page_repository service exists that returns Page objects. The
    // object returned has getters and setters exposing the state.
    $page = $app['page_repository']->find($id);
    $format = $request->getRequestFormat();

    if (!$page instanceof Page) {
        $app->abort("No page found for id: $id");
    }

    return new Response($app['serializer']->serialize($page, $format), 200, array(
        "Content-Type" => $request->getMimeType($format)
    ));
})->assert("_format", "xml|json")
  ->assert("id", "\d+");
```

Chapter 34
Service Controllers

As your Silex application grows, you may wish to begin organizing your controllers in a more formal fashion. Silex can use controller classes out of the box, but with a bit of work, your controllers can be created as services, giving you the full power of dependency injection and lazy loading.

Why would I want to do this?

- Dependency Injection over Service Location

  Using this method, you can inject the actual dependencies required by your controller and gain total inversion of control, while still maintaining the lazy loading of your controllers and its dependencies. Because your dependencies are clearly defined, they are easily mocked, allowing you to test your controllers in isolation.
- Framework Independence

  Using this method, your controllers start to become more independent of the framework you are using. Carefully crafted, your controllers will become reusable with multiple frameworks. By keeping careful control of your dependencies, your controllers could easily become compatible with Silex, Symfony (full stack) and Drupal, to name just a few.

Parameters

There are currently no parameters for the `ServiceControllerServiceProvider`.

Services

There are no extra services provided, the `ServiceControllerServiceProvider` simply extends the existing **resolver** service.

Registering

Listing 34-1
```
$app->register(new Silex\Provider\ServiceControllerServiceProvider());
```

Usage

In this slightly contrived example of a blog API, we're going to change the `/posts.json` route to use a controller, that is defined as a service.

Listing 34-2
```
use Silex\Application;
use Demo\Repository\PostRepository;

$app = new Application();

$app['posts.repository'] = function() {
    return new PostRepository;
};

$app->get('/posts.json', function() use ($app) {
    return $app->json($app['posts.repository']->findAll());
});
```

Rewriting your controller as a service is pretty simple, create a Plain Ol' PHP Object with your `PostRepository` as a dependency, along with an `indexJsonAction` method to handle the request. Although not shown in the example below, you can use type hinting and parameter naming to get the parameters you need, just like with standard Silex routes.

If you are a TDD/BDD fan (and you should be), you may notice that this controller has well defined responsibilities and dependencies, and is easily tested/specced. You may also notice that the only external dependency is on `Symfony\Component\HttpFoundation\JsonResponse`, meaning this controller could easily be used in a Symfony (full stack) application, or potentially with other applications or frameworks that know how to handle a *Symfony/HttpFoundation*[1] `Response` object.

Listing 34-3
```
namespace Demo\Controller;

use Demo\Repository\PostRepository;
use Symfony\Component\HttpFoundation\JsonResponse;

class PostController
{
    protected $repo;

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

    public function indexJsonAction()
    {
        return new JsonResponse($this->repo->findAll());
    }
}
```

And lastly, define your controller as a service in the application, along with your route. The syntax in the route definition is the name of the service, followed by a single colon (:), followed by the method name.

Listing 34-4

1. http://symfony.com/doc/master/components/http_foundation/introduction.html

```php
$app['posts.controller'] = function() use ($app) {
    return new PostController($app['posts.repository']);
};

$app->get('/posts.json', "posts.controller:indexJsonAction");
```

In addition to using classes for service controllers, you can define any callable as a service in the application to be used for a route.

Listing 34-5
```php
namespace Demo\Controller;

use Demo\Repository\PostRepository;
use Symfony\Component\HttpFoundation\JsonResponse;

function postIndexJson(PostRepository $repo) {
    return function() use ($repo) {
        return new JsonResponse($repo->findAll());
    };
}
```

And when defining your route, the code would look like the following:

Listing 34-6
```php
$app['posts.controller'] = function($app) {
    return Demo\Controller\postIndexJson($app['posts.repository']);
};

$app->get('/posts.json', 'posts.controller');
```

Chapter 35
Var Dumper

The *VarDumperServiceProvider* provides a mechanism that allows exploring then dumping any PHP variable.

Parameters

- **var_dumper.dump_destination**: A stream URL where dumps should be written to (defaults to `null`).

Services

- n/a

Registering

Listing 35-1
```
1  $app->register(new Silex\Provider\VarDumperServiceProvider());
```

Add the Symfony VarDumper Component as a dependency:

Listing 35-2
```
1  composer require symfony/var-dumper
```

Usage

Adding the VarDumper component as a Composer dependency gives you access to the `dump()` PHP function anywhere in your code.

If you are using Twig, it also provides a `dump()` Twig function and a `dump` Twig tag.

The VarDumperServiceProvider is also useful when used with the Silex WebProfiler as the dumps are made available in the web debug toolbar and in the web profiler.

Chapter 36
Doctrine

The *DoctrineServiceProvider* provides integration with the *Doctrine DBAL*[1] for easy database access (Doctrine ORM integration 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.
  - **charset**: Only relevant for `pdo_mysql`, and `pdo_oci/oci8`, specifies the charset used when connecting to the database.
  - **path**: Only relevant for `pdo_sqlite`, specifies the path to the SQLite database.
  - **port**: Only relevant for `pdo_mysql`, `pdo_pgsql`, and `pdo_oci/oci8`, specifies the port of the database to connect to.

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

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

Registering

Listing 36-1
```php
$app->register(new Silex\Provider\DoctrineServiceProvider(), array(
    'db.options' => array(
        'driver'   => 'pdo_sqlite',
        'path'     => __DIR__.'/app.db',
    ),
));
```

Add the Doctrine DBAL as a dependency:

Listing 36-2
```
composer require "doctrine/dbal:~2.2"
```

Usage

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

Listing 36-3
```php
$app->get('/blog/{id}', function ($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 36-4
```php
$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',
            'charset'   => 'utf8mb4',
        ),
        'mysql_write' => array(
            'driver'    => 'pdo_mysql',
            'host'      => 'mysql_write.someplace.tld',
            'dbname'    => 'my_database',
            'user'      => 'my_username',
            'password'  => 'my_password',
            'charset'   => 'utf8mb4',
        ),
    ),
));
```

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 36-5

```
$app['db']->fetchAll('SELECT * FROM table');

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

Using multiple connections:

Listing 36-6
```
$app->get('/blog/{id}', function ($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 = ?";
    $app['dbs']['mysql_write']->executeUpdate($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://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/

Chapter 37
Webserver Configuration

Apache

If you are using Apache, make sure `mod_rewrite` is enabled and use the following `.htaccess` file:

Listing 37-1
```
<IfModule mod_rewrite.c>
    Options -MultiViews

    RewriteEngine On
    #RewriteBase /path/to/app
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [QSA,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] to make your .htaccess even easier:

Listing 37-2
```
FallbackResource index.php
```

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

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

nginx

The **minimum configuration** to get your application running under Nginx is:

Listing 37-3

```
server {
    server_name domain.tld www.domain.tld;
    root /var/www/project/web;

    location / {
        # try to serve file directly, fallback to front controller
        try_files $uri /index.php$is_args$args;
    }

    # If you have 2 front controllers for dev|prod use the following line instead
    # location ~ ^/(index|index_dev)\.php(/|$) {
    location ~ ^/index\.php(/|$) {
        # the ubuntu default
        fastcgi_pass   unix:/var/run/php/phpX.X-fpm.sock;
        # for running on centos
        #fastcgi_pass   unix:/var/run/php-fpm/www.sock;

        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param HTTPS off;

        # Prevents URIs that include the front controller. This will 404:
        # http://domain.tld/index.php/some-path
        # Enable the internal directive to disable URIs like this
        # internal;
    }

    #return 404 for all php files as we do have a front controller
    location ~ \.php$ {
        return 404;
    }

    error_log /var/log/nginx/project_error.log;
    access_log /var/log/nginx/project_access.log;
}
```

IIS

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

Listing 37-4

```
<?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>
```

```
21      </system.webServer>
22  </configuration>
```

Lighttpd

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

Listing 37-5
```
server.document-root = "/path/to/app"

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

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

PHP

PHP ships with a built-in webserver for development. This server allows you to run silex without any configuration. However, in order to serve static files, you'll have to make sure your front controller returns false in that case:

Listing 37-6
```
// web/index.php

$filename = __DIR__.preg_replace('#(\?.*)$#', '', $_SERVER['REQUEST_URI']);
if (php_sapi_name() === 'cli-server' && is_file($filename)) {
    return false;
}

$app = require __DIR__.'/../src/app.php';
$app->run();
```

Assuming your front controller is at `web/index.php`, you can start the server from the command-line with this command:

Listing 37-7
```
$ php -S localhost:8080 -t web web/index.php
```

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 38

Changelog

2.1.0 (2016-XX-XX)

- added support for registering Swiftmailer plugins

2.0.4 (2016-11-06)

- fixed twig.app_variable definition
- added support for latest versions of Twig 1.x and 2.0 (Twig runtime loaders)
- added support for Symfony 2.3

2.0.3 (2016-08-22)

- fixed lazy evaluation of 'monolog.use_error_handler'
- fixed PHP7 type hint on controllers

2.0.2 (2016-06-14)

- fixed Symfony 3.1 deprecations

2.0.1 (2016-05-27)

- fixed the silex form extension registration to allow overriding default ones
- removed support for the obsolete Locale Symfony component (uses the Intl one now)
- added support for Symfony 3.1

2.0.0 (2016-05-18)

- decoupled the exception handler from HttpKernelServiceProvider
- Switched to BCrypt as the default encoder in the security provider
- added full support for RequestMatcher
- added support for Symfony Guard
- added support for callables in CallbackResolver
- added FormTrait::namedForm()
- added support for delivery_addresses, delivery_whitelist, and sender_address
- added support to register form types / form types extensions / form types guessers as services
- added support for callable in mounts (allow nested route collection to be built easily)
- added support for conditions on routes
- added support for the Symfony VarDumper Component
- added a global Twig variable (an AppVariable instance)
- [BC BREAK] CSRF has been moved to a standalone provider (`form.secret` is not available anymore)
- added support for the Symfony HttpFoundation Twig bridge extension
- added support for the Symfony Asset Component
- bumped minimum version of Symfony to 2.8
- bumped minimum version of PHP to 5.5.0
- Updated Pimple to 3.0
- Updated session listeners to extends HttpKernel ones
- [BC BREAK] Locale management has been moved to LocaleServiceProvider which must be registered if you want Silex to manage your locale (must also be registered for the translation service provider)
- [BC BREAK] Provider interfaces moved to SilexApi namespace, published as separate package via subtree split
- [BC BREAK] ServiceProviderInterface split in to EventListenerProviderInterface and BootableProviderInterface
- [BC BREAK] Service Provider support files moved under SilexProvider namespace, allowing publishing as separate package via sub-tree split
- `monolog.exception.logger_filter` option added to Monolog service provider
- [BC BREAK] `$app['request']` service removed, use `$app['request_stack']` instead

1.3.6 (2016-XX-XX)

- n/a

1.3.5 (2016-01-06)

- fixed typo in SecurityServiceProvider

1.3.4 (2015-09-15)

- fixed some new deprecations
- fixed translation registration for the validators

1.3.3 (2015-09-08)

- added support for Symfony 3.0 and Twig 2.0

- fixed some Form deprecations
- removed deprecated method call in the exception handler
- fixed Swiftmailer spool flushing when spool is not enabled

1.3.2 (2015-08-24)

- no changes

1.3.1 (2015-08-04)

- added missing support for the Expression constraint
- fixed the possibility to override translations for validator error messages
- fixed sub-mounts with same name clash
- fixed session logout handler when a firewall is stateless

1.3.0 (2015-06-05)

- added a *$app['user']* to get the current user (security provider)
- added view handlers
- added support for the OPTIONS HTTP method
- added caching for the Translator provider
- deprecated *$app['exception_handler']->disable()* in favor of *unset($app['exception_handler'])*
- made Silex compatible with Symfony 2.7 an 2.8 (and keep compatibility with Symfony 2.3, 2.5, and 2.6)
- removed deprecated TwigCoreExtension class (register the new HttpFragmentServiceProvider instead)
- bumped minimum version of PHP to 5.3.9

1.2.5 (2015-06-04)

- no code changes (last version of the 1.2 branch)

1.2.4 (2015-04-11)

- fixed the exception message when mounting a collection that doesn't return a ControllerCollection
- fixed Symfony dependencies (Silex 1.2 is not compatible with Symfony 2.7)

1.2.3 (2015-01-20)

- fixed remember me listener
- fixed translation files loading when they do not exist
- allowed global after middlewares to return responses like route specific ones

1.2.2 (2014-09-26)

- fixed Translator locale management

- added support for the $app argument in application middlewares (to make it consistent with route middlewares)
- added form.types to the Form provider

1.2.1 (2014-07-01)

- added support permissions in the Monolog provider
- fixed Switfmailer spool where the event dispatcher is different from the other ones
- fixed locale when changing it on the translator itself

1.2.0 (2014-03-29)

- Allowed disabling the boot logic of MonologServiceProvider
- Reverted "convert attributes on the request that actually exist"
- [BC BREAK] Routes are now always added in the order of their registration (even for mounted routes)
- Added run() on Route to be able to define the controller code
- Deprecated TwigCoreExtension (register the new HttpFragmentServiceProvider instead)
- Added HttpFragmentServiceProvider
- Allowed a callback to be a method call on a service (before, after, finish, error, on Application; convert, before, after on Controller)

1.1.3 (2013-XX-XX)

- Fixed translator locale management

1.1.2 (2013-10-30)

- Added missing "security.hide_user_not_found" support in SecurityServiceProvider
- Fixed event listeners that are registered after the boot via the on() method

1.0.2 (2013-10-30)

- Fixed SecurityServiceProvider to use null as a fake controller so that routes can be dumped

1.1.1 (2013-10-11)

- Removed or replaced deprecated Symfony code
- Updated code to take advantages of 2.3 new features
- Only convert attributes on the request that actually exist.

1.1.0 (2013-07-04)

- Support for any `Psr\Log\LoggerInterface` as opposed to the monolog-bridge one.
- Made dispatcher proxy methods `on`, `before`, `after` and `error` lazy, so that they will not instantiate the dispatcher early.

- Dropped support for 2.1 and 2.2 versions of Symfony.

1.0.1 (2013-07-04)

- Fixed RedirectableUrlMatcher::redirect() when Silex is configured to use a logger
- Make `DoctrineServiceProvider` multi-db support lazy.

1.0.0 (2013-05-03)

- **2013-04-12**: Added support for validators as services.
- **2013-04-01**: Added support for host matching with symfony 2.2:

Listing 38-1
```php
$app->match('/', function() {
    // app-specific action
})->host('example.com');

$app->match('/', function ($user) {
    // user-specific action
})->host('{user}.example.com');
```

- **2013-03-08**: Added support for form type extensions and guessers as services.
- **2013-03-08**: Added support for remember-me via the `RememberMeServiceProvider`.
- **2013-02-07**: Added `Application::sendFile()` to ease sending `BinaryFileResponse`.
- **2012-11-05**: Filters have been renamed to application middlewares in the documentation.
- **2012-11-05**: The `before()`, `after()`, `error()`, and `finish()` listener priorities now set the priority of the underlying Symfony event instead of a custom one before.
- **2012-11-05**: Removing the default exception handler should now be done via its `disable()` method:

> Before:
>
> ```
> unset($app['exception_handler']);
> ```
>
> After:
>
> ```
> $app['exception_handler']->disable();
> ```

- **2012-07-15**: removed the `monolog.configure` service. Use the `extend` method instead:

> Before:

Listing 38-2
```php
$app['monolog.configure'] = $app->protect(function ($monolog) use ($app) {
    // do something
});
```

After:

Listing 38-3
```
$app['monolog'] = $app->share($app->extend('monolog', function($monolog, $app) {
    // do something

    return $monolog;
}));
```

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

Before:

Listing 38-4
```
$controllers = new ControllerCollection();
```

After:

Listing 38-5
```
$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 38-6
```
$app['twig.configure'] = $app->protect(function ($twig) use ($app) {
    // do something
});
```

After:

Listing 38-7
```
$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 38-8
```
$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 38-9
```
$app = new Application();
$app->get('/bar', function() { return 'foo'; });

return $app;
```

After:

Listing 38-10
```
$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:

Listing 38-11
```
$app->match('/', function () { echo 'foo'; }, 'GET|POST');
```

After:

Listing 38-12
```
$app->match('/', function () { echo 'foo'; })->method('GET|POST');
```