Symfony

The Quick Tour

What could be better to make up your own mind than to try out Symfony yourself? Aside from a little time, it will cost you nothing. Step by step you will explore the Symfony universe. Be careful, Symfony can become addictive from the very first encounter!

Chapter 1
The Big Picture

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

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

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

Downloading Symfony2

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

Ready? Start by downloading the "*Symfony2 Standard Edition*[1]", a Symfony *distribution* that is preconfigured for the most common use cases and also contains some code that demonstrates how to use Symfony2 (get the archive with the *vendors* included to get started even faster).

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

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

1. `http://symfony.com/download`

```
        DemoBundle/
            Controller/
            Resources/
            ...
vendor/
    symfony/
    doctrine/
    ...
web/
    app.php
    ...
```

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

```
php bin/vendors install
```

Checking the Configuration

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

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

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

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

Symfony2 should welcome and congratulate you for your hard work so far!
../_images/welcome.jpg

Understanding the Fundamentals

One of the main goals of a framework is to ensure *Separation of Concerns*[2]. This keeps your code organized and allows your application to evolve easily over time by avoiding the mixing of database calls, HTML tags, and business logic in the same script. To achieve this goal with Symfony, you'll first need to learn a few fundamental concepts and terms.

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

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

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

../_images/hello_fabien.png

2. http://en.wikipedia.org/wiki/Separation_of_concerns

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

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

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

Routing

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

```
# app/config/routing_dev.yml
_welcome:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Welcome:index }

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

# ...
```

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

The Symfony2 Standard Edition uses *YAML*[3] for its configuration files, but Symfony2 also supports XML, PHP, and annotations natively. The different formats are compatible and may be used interchangeably within an application. Also, the performance of your application does not depend on the configuration format you choose as everything is cached on the very first request.

Controllers

A controller is a fancy name for a PHP function or method that handles incoming *requests* and returns *responses* (often HTML code). Instead of using the PHP global variables and functions (like `$_GET` or `header()`) to manage these HTTP messages, Symfony uses objects: ***Request***[4] and ***Response***[5]. The simplest possible controller might create the response by hand, based on the request:

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

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

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

3. http://www.yaml.org/
4. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html
5. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html

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

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

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

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

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

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

The `WelcomeController` class extends the built-in `Controller` class, which provides useful shortcut methods, like the *`render()`*[6] method that loads and renders a template (`AcmeDemoBundle:Welcome:index.html.twig`). The returned value is a Response object populated with the rendered content. So, if the needs arise, the Response can be tweaked before it is sent to the browser:

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

    return $response;
}
```

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

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

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

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

6. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#render()

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

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

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

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

    // ...
}
```

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

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

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

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

Templates

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

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

```
{% block title "Hello " ~ name %}

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

By default, Symfony2 uses *Twig*[7] as its template engine but you can also use traditional PHP templates if you choose. The next chapter will introduce how templates work in Symfony2.

Bundles

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

Working with Environments

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

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

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

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

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

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

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

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

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

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

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

where performance is key. That's why you should always use the development environment when developing your application.

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

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

web_profiler:
    toolbar: true
    intercept_redirects: false
```

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

Final Thoughts

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

Chapter 2
The View

After reading the first part of this tutorial, you have decided that Symfony2 was worth another 10 minutes. Great choice! In this second part, you will learn more about the Symfony2 template engine, *Twig*[8]. Twig is a flexible, fast, and secure template engine for PHP. It makes your templates more readable and concise; it also makes them more friendly for web designers.

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

Getting familiar with Twig

If you want to learn Twig, we highly recommend you to read its official *documentation*[9]. This section is just a quick overview of the main concepts.

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

- `{{ ... }}`: Prints a variable or the result of an expression;
- `{% ... %}`: Controls the logic of the template; it is used to execute `for` loops and `if` statements, for example.

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

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Webpage</title>
```

8. http://twig.sensiolabs.org/
9. http://twig.sensiolabs.org/documentation

```
    </head>
    <body>
        <h1>{{ page_title }}</h1>

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

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

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

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

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

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

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

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

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

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

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

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

Decorating Templates

More often than not, templates in a project share common elements, like the well-known header and footer. In Symfony2, we like to think about this problem differently: a template can be decorated by another one. This works exactly the same as PHP classes: template inheritance allows you to build a

base "layout" template that contains all the common elements of your site and defines "blocks" that child templates can override.

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

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

{% block title "Hello " ~ name %}

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

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

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

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

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

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

Using Tags, Filters, and Functions

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

Including other Templates

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

Create an `embedded.html.twig` template:

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

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

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

{# override the body block from embedded.html.twig #}
{% block content %}
    {% include "AcmeDemoBundle:Demo:embedded.html.twig" %}
{% endblock %}
```

Embedding other Controllers

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

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

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

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

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

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

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

    // ...
}
```

Creating Links between Pages

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

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

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

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

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

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

Including Assets: images, JavaScripts, and stylesheets

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

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

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

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

Escaping Variables

Twig is configured to automatically escapes all output by default. Read Twig *documentation*[10] to learn more about output escaping and the Escaper extension.

Final Thoughts

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

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

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

10. `http://twig.sensiolabs.org/documentation`

Chapter 3
The Controller

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

Using Formats

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

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

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

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

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

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

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

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

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

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

Redirecting and Forwarding

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

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

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

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

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

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

Getting information from the Request

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

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

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

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

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

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

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

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

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

Persisting Data in the Session

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

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

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

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

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

// set the user locale
$session->setLocale('fr');
```

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

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

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

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

Securing Resources

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

```
# app/config/security.yml
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

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

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

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

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

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

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

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

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

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

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

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

Now, log in as `user` (who does *not* have the `ROLE_ADMIN` role) and from the secured hello page, click on the "Hello resource secured" link. Symfony2 should return a 403 HTTP status code, indicating that the user is "forbidden" from accessing that resource.

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

Caching Resources

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

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

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

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

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

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

Final Thoughts

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

Chapter 4
The Architecture

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

Understanding the Directory Structure

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

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

The `web/` Directory

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

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

use Symfony\Component\HttpFoundation\Request;

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

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

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

The app/ Directory

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

This class must implement two methods:

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

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

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

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

// ...

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

The *`UniversalClassLoader`*[11] is used to autoload files that respect either the technical interoperability *standards*[12] for PHP 5.3 namespaces or the PEAR naming *convention*[13] for classes. As you can see here, all dependencies are stored under the `vendor/` directory, but this is just a convention. You can store them wherever you want, globally on your server or locally in your projects.

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

Understanding the Bundle System

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

11. http://api.symfony.com/2.0/Symfony/Component/ClassLoader/UniversalClassLoader.html
12. http://groups.google.com/group/php-standards/web/psr-0-final-proposal
13. http://pear.php.net/

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

Registering a Bundle

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

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

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

    return $bundles;
}
```

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

Configuring a Bundle

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

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

framework:
    secret:          %secret%
    charset:         UTF-8
    router:          { resource: "%kernel.root_dir%/config/routing.yml" }
    form:            true
    csrf_protection: true
    validation:      { enable_annotations: true }
    templating:      { engines: ['twig'] } #assets_version: SomeVersionScheme
```

```
    session:
        default_locale: %locale%
        auto_start:     true

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

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

# Doctrine Configuration
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
        dbname:   %database_name%
        user:     %database_user%
        password: %database_password%
        charset:  UTF8

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

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

jms_security_extra:
    secure_controllers:  true
    secure_all_services: false
```

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

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

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

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

web_profiler:
```

```
    toolbar: true
    intercept_redirects: false

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

assetic:
    use_controller: true
```

Extending a Bundle

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

Logical File Names

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

Logical Controller Names

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

Logical Template Names

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

Extending Bundles

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

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

Using Vendors

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

Understanding the Cache and Logs

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

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

Using the Command Line Interface

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

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

```
php app/console
```

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

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

Final Thoughts

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

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

Symfony2, The Quick Tour

If you find typos or errors, feel free to report them by creating a ticket on the Symfony ticketing system (*http://trac.symfony-project.org/register*). Based on tickets and users feedback, this book is continuously updated.

You can contact the author about this book, Symfony and Open-Source at *fabien@symfony.com* or for training, consulting, application development, or business related questions at *fabien.potencier@sensio.com*.