Symfony

The Quick Tour

for Symfony 2.4

generated on May 28, 2014

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!

The Quick Tour (2.4)

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

Contents at a Glance

The Big Picture 4
The View 12
The Controller 17
The Architecture 21

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.

Installing Symfony2

First, check that the PHP version installed on your computer meets the Symfony2 requirements: 5.3.3 or higher. Then, open a console and execute the following command to install the latest version of Symfony2 in the `myproject/` directory:

Listing 1-1
```
1 $ composer create-project symfony/framework-standard-edition myproject/ ~2.4
```

Composer[1] is the package manager used by modern PHP applications and the only recommended way to install Symfony2. To install Composer on your Linux or Mac system, execute the following commands:

Listing 1-2
```
1 $ curl -sS https://getcomposer.org/installer | php
2 $ sudo mv composer.phar /usr/local/bin/composer
```

To install Composer on a Windows system, download the *executable installer*[2].

Beware that the first time you install Symfony2, it may take a few minutes to download all its components. At the end of the installation process, the installer will ask you to provide some configuration options for the Symfony2 project. For this first project you can safely ignore this configuration by pressing the `<Enter>` key repeatedly.

1. `https://getcomposer.org/`
2. `http://getcomposer.org/download`

Running Symfony2

Before running Symfony2 for the first time, execute the following command to make sure that your system meets all the technical requirements:

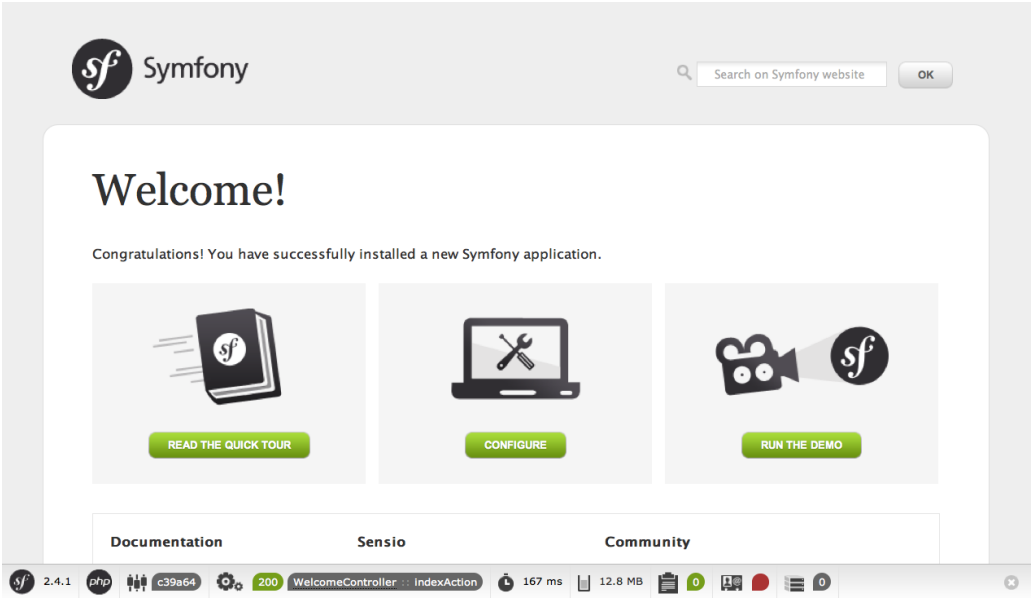
Listing 1-3
```
$ cd myproject/
$ php app/check.php
```

Fix any error reported by the command and then use the PHP built-in web server to run Symfony:

Listing 1-4
```
$ php app/console server:run
```

If you get the error *There are no commands defined in the "server" namespace.*, then you are probably using PHP 5.3. That's ok! But the built-in web server is only available for PHP 5.4.0 or higher. If you have an older version of PHP or if you prefer a traditional web server such as Apache or Nginx, read the *Configuring a web server* article.

Open your browser and access the `http://localhost:8000` URL to see the Welcome page of Symfony2:



Understanding the Fundamentals

One of the main goals of a framework is to keep your code organized and to allow 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.

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

Listing 1-5
```
http://localhost:8000/demo/hello/Fabien
```

What's going on here? Have a look at each part of 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 matching the requested URL (i.e. the virtual path) against some configured paths. The demo paths are defined in the `app/config/routing_dev.yml` configuration file:

Listing 1-6

```yaml
# app/config/routing_dev.yml
# ...

# AcmeDemoBundle routes (to be removed)
_acme_demo:
    resource: "@AcmeDemoBundle/Resources/config/routing.yml"
```

This imports a `routing.yml` file that lives inside the AcmeDemoBundle:

Listing 1-7

```yaml
# src/Acme/DemoBundle/Resources/config/routing.yml
_welcome:
    path:     /
    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.

In addition to YAML files, routes can be configured in XML or PHP files and can even be embedded in PHP annotations. This flexibility is one of the main features of Symfony2, a framework that never imposes a particular configuration format on you.

Controllers

A controller is 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* and *Response*. The simplest possible controller might create the response by hand, based on the request:

Listing 1-8
```
use Symfony\Component\HttpFoundation\Response;

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

return new Response('Hello '.$name);
```

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:

Listing 1-9
```
// 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 using 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()* 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 need arises, the `Response` can be tweaked before it is sent to the browser:

Listing 1-10
```
public function indexAction()
{
    $response = $this->render('AcmeDemoBundle:Welcome:index.txt.twig');
    $response->headers->set('Content-Type', 'text/plain');

```

```
6       return $response;
7   }
```

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`.

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:

Listing 1-11
```
1  # src/Acme/DemoBundle/Resources/config/routing.yml
2  # ...
3  _demo:
4      resource: "@AcmeDemoBundle/Controller/DemoController.php"
5      type:     annotation
6      prefix:   /demo
```

The *logical name* of the file containing the `_demo` routes 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:

Listing 1-12
```
1  // src/Acme/DemoBundle/Controller/DemoController.php
2  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
3  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
4
5  class DemoController extends Controller
6  {
7      /**
8       * @Route("/hello/{name}", name="_demo_hello")
9       * @Template()
10      */
11     public function helloAction($name)
12     {
13         return array('name' => $name);
14     }
15
16     // ...
17 }
```

The `@Route()` annotation creates a new route matching the `/hello/{name}` path to the `helloAction()` method. Any string enclosed in curly brackets, like `{name}`, is considered a variable that can be directly retrieved as a method argument with the same name.

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 to it each variable of the returned array. 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`).

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):

Listing 1-13
```twig
{# 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 %}
```

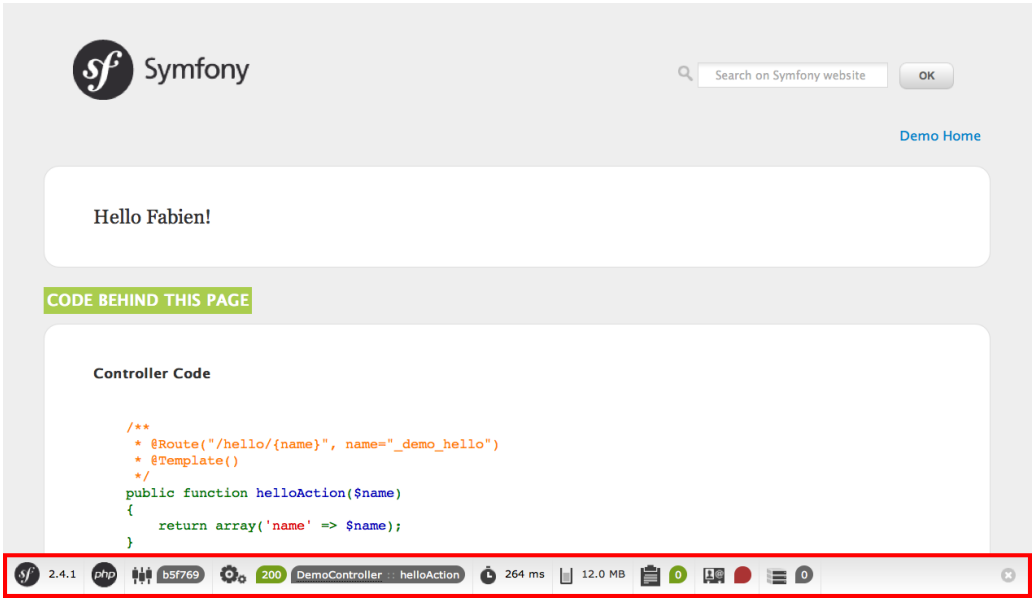
By default, Symfony2 uses *Twig*[3] as its template engine but you can also use traditional PHP templates if you choose. The *second part of this tutorial* will introduce how templates work in Symfony2.

Bundles

You might have wondered why the *Bundle* word is used in many names you 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, you have manipulated one bundle, AcmeDemoBundle. You will learn more about bundles in the *last part of this tutorial*.

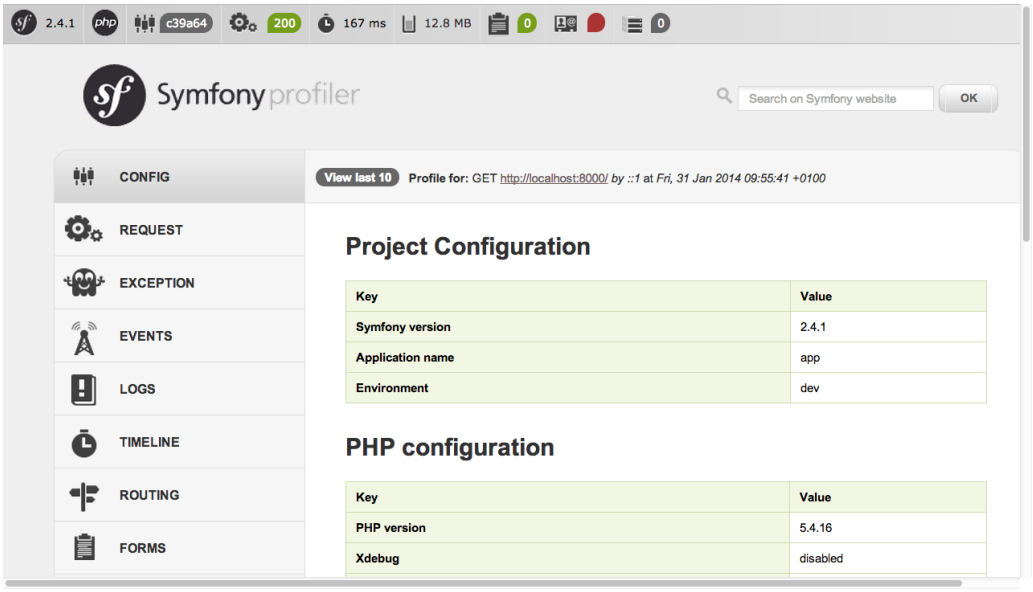
Working with Environments

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 the "Web Debug Toolbar", and it is a Symfony2 developer's best friend!



But what you see initially is only the tip of the iceberg; click on any of the bar sections to open the profiler and get much more detailed information about the request, the query parameters, security details, and database queries:

3. http://twig.sensiolabs.org/



Of course, it would be unwise to have this tool enabled when you deploy your application, so by default, the profiler is not enabled in the `prod` environment.

What is an Environment?

An *Environment* represents a group of configurations that's used to run your application. Symfony2 defines two environments by default: `dev` (suited for when developing the application locally) and `prod` (optimized for when executing the application on production).

Typically, the environments share a large amount of configuration options. For that reason, you put your common configuration in `config.yml` and override the specific configuration file for each environment where necessary:

Listing 1-14
```yaml
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

web_profiler:
    toolbar: true
    intercept_redirects: false
```

In this example, the `dev` environment loads the `config_dev.yml` configuration file, which itself imports the common `config.yml` file and then modifies it by enabling the web debug toolbar.

When you visit the `app_dev.php` file in your browser, you're executing your Symfony application in the `dev` environment. To visit your application in the `prod` environment, visit the `app.php` file instead.

The demo routes in our application are only available in the `dev` environment. Therefore, if you try to access the `http://localhost/app.php/demo/hello/Fabien` URL, you'll get a 404 error.

If instead of using PHP's built-in webserver, you use Apache with `mod_rewrite` enabled and take advantage of the `.htaccess` file Symfony2 provides in `web/`, you can even omit the `app.php` part of the URL. The default `.htaccess` points all requests to the `app.php` front controller:

Listing 1-15
```
http://localhost/demo/hello/Fabien
```

For more details on environments, see "*Environments & Front Controllers*" article.

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. In this second part, you will learn more about *Twig*[1], the fast, flexible, and secure template engine for PHP. Twig makes your templates more readable and concise; it also makes them more friendly for web designers.

Getting familiar with Twig

The official *Twig documentation*[2] is the best resource to learn everything about this new template engine. This section just gives you a quick overview of its main concepts.

A Twig template is a text file that can generate any type of content (HTML, CSS, JavaScript, XML, CSV, LaTeX, ...). Twig elements are separated from the rest of the template contents using any of these delimiters:

- `{{ ... }}`: prints the content of a variable or the result of an expression;
- `{% ... %}`: controls the logic of the template; it is used for example to execute `for` loops and `if` statements;
- `{# ... #}`: allows including comments inside templates.

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

Listing 2-1

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>{{ page_title }}</title>
5      </head>
6      <body>
7          <h1>{{ page_title }}</h1>
8
9          <ul id="navigation">
```

1. http://twig.sensiolabs.org/
2. http://twig.sensiolabs.org/documentation

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

To render a template in Symfony, use the `render` method from within a controller and pass the variables needed as an array using the optional second argument:

Listing 2-2
```
$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. The following code listing shows how to display the content of a variable depending on the type of the variable passed by the controller:

Listing 2-3
```
{# 1. Simple variables #}
{# array('name' => 'Fabien') #}
{{ name }}

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

{# alternative syntax for arrays #}
{{ user['name'] }}

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

{# alternative syntax for objects #}
{{ user.name() }}
{{ user.getName() }}
```

Decorating Templates

More often than not, templates in a project share common elements, like the well-known header and footer. Twig solves this problem elegantly with a concept called "template inheritance". This feature 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 uses the `extends` tag to indicate that it inherits from the common `layout.html.twig` template:

Listing 2-4
```
{# src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig #}
{% extends "AcmeDemoBundle::layout.html.twig" %}

{% block title "Hello " ~ name %}

```

```
6  {% block content %}
7      <h1>Hello {{ name }}!</h1>
8  {% 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 of the bundle.

Now, simplify the `layout.html.twig` template:

Listing 2-5
```
1  {# src/Acme/DemoBundle/Resources/views/layout.html.twig #}
2  <div>
3      {% block content %}
4      {% endblock %}
5  </div>
```

The `{% block %}` tags 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>` element.

Using Tags, Filters, and Functions

One of the best feature of Twig is its extensibility via tags, filters, and functions. Take a look at the following sample template that uses filters extensively to modify the information before displaying it to the user:

Listing 2-6
```
1  <h1>{{ article.title|trim|capitalize }}</h1>
2
3  <p>{{ article.content|striptags|slice(0, 1024) }}</p>
4
5  <p>Tags: {{ article.tags|sort|join(", ") }}</p>
6
7  <p>Next article will be published on {{ 'next Monday'|date('M j, Y')}}</p>
```

Don't forget to check out the official *Twig documentation*[3] to learn everything about filters, functions and tags.

Including other Templates

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

First, create an `embedded.html.twig` template:

Listing 2-7
```
1  {# src/Acme/DemoBundle/Resources/views/Demo/embedded.html.twig #}
2  Hello {{ name }}
```

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

Listing 2-8
```
1  {# src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig #}
2  {% extends "AcmeDemoBundle::layout.html.twig" %}
```

3. http://twig.sensiolabs.org/documentation

```
3
4  {# override the body block from embedded.html.twig #}
5  {% block content %}
6      {{ include("AcmeDemoBundle:Demo:embedded.html.twig") }}
7  {% 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 `topArticlesAction` controller method to display the most popular articles of your website. If you want to "render" the result of that method (e.g. `HTML`) inside the `index` template, use the `render` function:

Listing 2-9
```
1  {# src/Acme/DemoBundle/Resources/views/Demo/index.html.twig #}
2  {{ render(controller("AcmeDemoBundle:Demo:topArticles", {'num': 10})) }}
```

Here, the `AcmeDemoBundle:Demo:topArticles` string refers to the `topArticlesAction` action of the `Demo` controller, and the `num` argument is made available to the controller:

Listing 2-10
```
1  // src/Acme/DemoBundle/Controller/DemoController.php
2
3  class DemoController extends Controller
4  {
5      public function topArticlesAction($num)
6      {
7          // look for the $num most popular articles in the database
8          $articles = ...;
9
10         return $this->render('AcmeDemoBundle:Demo:topArticles.html.twig', array(
11             'articles' => $articles,
12         ));
13     }
14
15     // ...
16 }
```

Creating Links between Pages

Creating links between pages is a must for web applications. 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:

Listing 2-11
```
1  <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 key under which routes are defined and the parameters are the values of the variables defined in the route pattern:

Listing 2-12
```
1  // src/Acme/DemoBundle/Controller/DemoController.php
2  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
```

```php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

// ...

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

The `url` function is very similar to the `path` function, but generates *absolute* URLs, which is very handy when rendering emails and RSS files: `{{ 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:

Listing 2-13
```twig
<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.

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?

Chapter 3
The Controller

Still here after the first two parts? You are already becoming a Symfony2 addict! Without further ado, 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:

Listing 3-1
```php
// 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 special `_format` variable), Symfony2 automatically selects the right template, here `hello.xml.twig`:

Listing 3-2
```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 path instead:

Listing 3-3
```php
// 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 match URLs like `/demo/hello/Fabien.xml` or `/demo/hello/Fabien.json`.

The `requirements` entry defines regular expressions that variables 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:

Listing 3-4
```php
return $this->redirect($this->generateUrl('_demo_hello', array('name' => 'Lucas')));
```

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

You can also internally forward the action to another using the `forward()` method:

Listing 3-5
```php
return $this->forward('AcmeDemoBundle:Hello:fancy', array(
    'name'  => $name,
    'color' => 'green'
));
```

Displaying Error Pages

Errors will inevitably happen during the execution of every web application. In the case of `404` errors, Symfony includes a handy shortcut that you can use in your controllers:

Listing 3-6
```php
throw $this->createNotFoundException();
```

For 500 errors, just throw a regular PHP exception inside the controller and Symfony will transform it into a proper 500 error page:

Listing 3-7
```php
throw new \Exception('Something went wrong!');
```

Getting Information from the Request

Symfony automatically injects the `Request` object when the controller has an argument that's type hinted with `Symfony\Component\HttpFoundation\Request`:

Listing 3-8
```php
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $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:

Listing 3-9
```
{{ 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:

Listing 3-10
```php
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $session = $this->request->getSession();

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

    // get the value of a session attribute
    $foo = $session->get('foo');

    // use a default value if the attribute doesn't exist
    $foo = $session->get('foo', 'default_value');
}
```

You can also store "flash messages" that will auto-delete after the next request. They are useful when you need to set a success message before redirecting the user to another page (which will then show the message):

Listing 3-11
```php
// store a message for the very next request (in a controller)
$session->getFlashBag()->add('notice', 'Congratulations, your action succeeded!');
```

Listing 3-12
```twig
{# display the flash message in the template #}
<div>{{ app.session.flashbag.get('notice') }}</div>
```

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:

Listing 3-13
```php
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 (`86400` seconds). Resource caching is managed by Symfony2 itself. But because caching is managed using standard HTTP cache headers, you can use Varnish or Squid without having to modify a single line of code in your application.

Final Thoughts

That's all there is to it, and I'm not even sure you'll have spent the full 10 minutes. You were briefly introduced to bundles in the first part, and all the features you'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 recommended structure is as follows:

- `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:

Listing 4-1
```php
// 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 controller first bootstraps the application using a kernel class (`AppKernel` in this case). Then, it creates the `Request` object using the PHP's global variables and passes it to the kernel. The last step is to send the response contents returned by the kernel back to the user.

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).

Autoloading is handled automatically via *Composer*[1], which means that you can use any PHP class without doing anything at all! 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.

Understanding the Bundle System

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

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:

Listing 4-2

```
// 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(),
    );

    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;
}
```

1. http://getcomposer.org

In addition to the AcmeDemoBundle that was 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 Symfony configuration:

Listing 4-3

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

framework:
    #esi:             ~
    #translator:      { fallback: "%locale%" }
    secret:          "%secret%"
    router:
        resource: "%kernel.root_dir%/config/routing.yml"
        strict_requirements: "%kernel.debug%"
    form:            true
    csrf_protection: true
    validation:      { enable_annotations: true }
    templating:      { engines: ['twig'] } #assets_version: SomeVersionScheme
    default_locale:  "%locale%"
    trusted_proxies: ~
    session:         ~

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

# Assetic Configuration
assetic:
    debug:          "%kernel.debug%"
    use_controller: false
    bundles:        [ ]
    #java: /usr/bin/java
    filters:
        cssrewrite: ~
        #closure:
        #    jar: "%kernel.root_dir%/Resources/java/compiler.jar"
        #yui_css:
        #    jar: "%kernel.root_dir%/Resources/java/yuicompressor-2.4.7.jar"

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

    orm:
```

```
        auto_generate_proxy_classes: "%kernel.debug%"
        auto_mapping: true

# Swift Mailer Configuration
swiftmailer:
    transport: "%mailer_transport%"
    host:      "%mailer_host%"
    username:  "%mailer_user%"
    password:  "%mailer_password%"
    spool:     { type: memory }
```

Each first level entry like `framework`, `twig` or `doctrine` 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:

Listing 4-4

```
# 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 it overrides AcmeDemoBundle. When Symfony loads the `AcmeDemoBundle:Welcome:index` controller, it will first look for the `WelcomeController` class in AcmeNewBundle and, if it doesn't exist, 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, to speed things up, 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:

Listing 4-5
```
1 $ php app/console
```

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

Listing 4-6
```
1 $ 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 *The Book* and pick any topic you want.