Create your PHP Framework

.. toctree::

part01
part02
part03
part04
part05
part06
part07
part08
part09
part10
part11
part12

Introduction

Symfony2_ is a reusable set of standalone, decoupled, and cohesive PHP components that solve common web development problems.

Instead of using these low-level components, you can use the ready-to-be-used Symfony2 full-stack web framework, which is based on these components... or you can create your very own framework. This book is about the latter.

.. note::

If you just want to use the Symfony2 full-stack framework, you'd better
read its official `documentation`_ instead.

Why would you like to create your own framework?

Why would you like to create your own framework in the first place? If you look around, everybody will tell you that it's a bad thing to reinvent the wheel and that you'd better choose an existing framework and forget about creating your own altogether. Most of the time, they are right but I can think of a few good reasons to start creating your own framework:

- To learn more about the low level architecture of modern web frameworks in general and about the Symfony2 full-stack framework internals in particular;
- To create a framework tailored to your very specific needs (just be sure first that your needs are really specific);
- To experiment creating a framework for fun (in a learn-and-throw-away approach);
- To refactor an old/existing application that needs a good dose of recent web development best practices;
- To prove the world that you can actually create a framework on your own (... but with little effort).

I will gently guide you through the creation of a web framework, one step at a time. At each step, you will have a fully-working framework that you can use as is or as a start for your very own. We will start with simple frameworks and more features will be added with time. Eventually, you will have a fully-featured full-stack web framework.

And of course, each step will be the occasion to learn more about some of the Symfony2 Components.

.. tip::

If you don't have time to read the whole book, or if you want to get
started fast, you can also have a look at `Silex`_, a micro-framework
based on the Symfony2 Components. The code is rather slim and it leverages
many aspects of the Symfony2 Components.

Many modern web frameworks advertize themselves as being MVC frameworks. We won't talk

about the MVC pattern as the Symfony2 Components are able to create any type of frameworks, not just the ones that follow the MVC architecture. Anyway, if you have a look at the MVC semantics, this book is about how to create the Controller part of a framework. For the Model and the View, it really depends on your personal taste and I will let you use any existing third-party libraries (Doctrine, Propel, or plain-old PDO for the Model; PHP or Twig for the View).

When creating a framework, following the MVC pattern is not the right goal. The main goal should be the **Separation of Concerns**; I actually think that this is the only design pattern that you should really care about. The fundamental principles of the Symfony2 Components are focused on the HTTP specification. As such, the frameworks that we are going to create should be more accurately labelled as HTTP frameworks or Request/Response frameworks.

Before we start

Reading about how to create a framework is not enough. You will have to follow along and actually type all the examples we will work on. For that, you need a recent version of PHP (5.3.8 or later is good enough), a web server (like Apache or NGinx), a good knowledge of PHP and an understanding of Object Oriented programming.

Ready to go? Let's start.

Bootstrapping

Before we can even think of creating our first framework, we need to talk about some conventions: where we will store our code, how we will name our classes, how we will reference external dependencies, etc.

To store our framework, create a directory somewhere on your machine:

.. code-block:: sh

```
$ mkdir framework
$ cd framework
```

Dependency Management ~~~~~~~~~~~~~~~~~~~~

To install the Symfony2 Components that we need for our framework, we are going to use Composer_, a project dependency manager for PHP. If you don't have it yet,download and install_ Composer now:

.. code-block:: sh

```
$ curl -sS https://getcomposer.org/installer | php
```

Then, generate an empty composer.json file, where Composer will store the framework dependencies:

.. code-block:: sh

```
$ php composer.phar init -n
```

Our Project

Instead of creating our framework from scratch, we are going to write the same "application" over and over again, adding one abstraction at a time. Let's start with the simplest web application we can think of in PHP::

```
<?php

// framework/index.php

$input = $_GET['name'];

printf('Hello %s', $input);
```

Use the PHP built-in server to test this great application in a browser (http://localhost:4321/index.php?name=Fabien):

.. code-block:: sh

```
$ php -S 127.0.0.1:4321
```

In the next chapter, we are going to introduce the HttpFoundation Component and see what it brings us.

.. _Symfony2: http://symfony.com/ .. _documentation: http://symfony.com/doc .. _Silex: http://silex.sensiolabs.org/ .. _Composer: http://packagist.org/about-composer .. _download and install: https://getcomposer.org/doc/01-basic-usage.md

The HttpFoundation Component

Before diving into the framework creation process, I first want to step back and take a look at why you would like to use a framework instead of keeping your plain-old PHP applications as is. Why using a framework is actually a good idea, even for the simplest snippet of code and why creating your framework on top of the Symfony2 components is better than creating a framework from scratch.

.. note::

I won't talk about the obvious and traditional benefits of using a framework when working on big applications with more than a few developers; the Internet has already plenty of good resources on that topic.

Even if the "application" we wrote in the previous chapter was simple enough, it suffers from a few problems::

```
<?php

// framework/index.php

$input = $_GET['name'];

printf('Hello %s', $input);
```

First, if the `name` query parameter is not defined in the URL query string, you will get a PHP warning; so let's fix it::

```
<?php

// framework/index.php

$input = isset($_GET['name']) ? $_GET['name'] : 'World';

printf('Hello %s', $input);
```

Then, this *application is not secure*. Can you believe it? Even this simple snippet of PHP code is vulnerable to one of the most widespread Internet security issue, XSS (Cross-Site Scripting). Here is a more secure version::

```
<?php

$input = isset($_GET['name']) ? $_GET['name'] : 'World';

header('Content-Type: text/html; charset=utf-8');

printf('Hello %s', htmlspecialchars($input, ENT_QUOTES, 'UTF-8'));
```

.. note::

As you might have noticed, securing your code with ``htmlspecialchars`` is tedious and error prone. That's one of the reasons why using a template engine like `Twig`_, where auto-escaping is enabled by default, might be a good idea (and explicit escaping is also less painful with the usage of a

simple ``e`` filter).

As you can see for yourself, the simple code we had written first is not that simple anymore if we want to avoid PHP warnings/notices and make the code more secure.

Beyond security, this code is not even easily testable. Even if there is not much to test, it strikes me that writing unit tests for the simplest possible snippet of PHP code is not natural and feels ugly. Here is a tentative PHPUnit unit test for the above code::

```
<?php

// framework/test.php

class IndexTest extends \PHPUnit_Framework_TestCase
{
    public function testHello()
    {
        $_GET['name'] = 'Fabien';

        ob_start();
        include 'index.php';
        $content = ob_get_clean();

        $this->assertEquals('Hello Fabien', $content);
    }
}
```

.. note::

If our application were just slightly bigger, we would have been able to find even more problems. If you are curious about them, read the `Symfony2 versus Flat PHP`_ chapter of the Symfony2 documentation.

At this point, if you are not convinced that security and testing are indeed two very good reasons to stop writing code the old way and adopt a framework instead (whatever adopting a framework means in this context), you can stop reading this book now and go back to whatever code you were working on before.

.. note::

Of course, using a framework should give you more than just security and testability, but the more important thing to keep in mind is that the framework you choose must allow you to write better code faster.

Going OOP with the HttpFoundation Component

Writing web code is about interacting with HTTP. So, the fundamental principles of our framework should be around the HTTP specification_.

The HTTP specification describes how a client (a browser for instance) interacts with a server (our application via a web server). The dialog between the client and the server is specified by well-defined *messages*, requests and responses: *the client sends a request to the server and based on this request, the server returns a response*.

In PHP, the request is represented by global variables ($_GET, $_POST, $_FILE, $_COOKIE,

$_SESSION...) and the response is generated by functions (echo, header, setcookie, ...).

The first step towards better code is probably to use an Object-Oriented approach; that's the main goal of the Symfony2 HttpFoundation component: replacing the default PHP global variables and functions by an Object-Oriented layer.

To use this component, add it as a dependency of the project:

.. code-block:: sh

```
$ php composer.phar require symfony/http-foundation
```

Running this command will also automatically download the Symfony HttpFoundation component and install it under the vendor/ directory.

.. sidebar:: Class Autoloading

When installing a new dependency, Composer also generates a
``vendor/autoload.php`` file that allows any class to be easily
`autoloaded`_. Without autoloading, you would need to require the file
where a class is defined before being able to use it. But thanks to
`PSR-0`_, we can just let Composer and PHP do the hard work for us.

Now, let's rewrite our application by using theRequest and the Response classes::

```
<?php

// framework/index.php

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

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

$request = Request::createFromGlobals();

$input = $request->get('name', 'World');

$response = new Response(sprintf('Hello %s', htmlspecialchars($input, ENT_QUOTES, 'UTF-8')));

$response->send();
```

The createFromGlobals() method creates a Request object based on the current PHP global variables.

The send() method sends the Response object back to the client (it first outputs the HTTP headers followed by the content).

.. tip::

Before the ``send()`` call, we should have added a call to the
``prepare()`` method (``$response->prepare($request);``) to ensure that
our Response were compliant with the HTTP specification. For instance, if
we were to call the page with the ``HEAD`` method, it would remove the
content of the Response.

The main difference with the previous code is that you have total control of the HTTP messages.

You can create whatever request you want and you are in charge of sending the response whenever you see fit.

.. note::

We haven't explicitly set the ``Content-Type`` header in the rewritten
code as the charset of the Response object defaults to ``UTF-8``.

With the Request class, you have all the request information at your fingertips thanks to a nice and simple API::

```
<?php

// the URI being requested (e.g. /about) minus any query parameters
$request->getPathInfo();

// retrieve GET and POST variables respectively
$request->query->get('foo');
$request->request->get('bar', 'default value if bar does not exist');

// retrieve SERVER variables
$request->server->get('HTTP_HOST');

// retrieves an instance of UploadedFile identified by foo
$request->files->get('foo');

// retrieve a COOKIE value
$request->cookies->get('PHPSESSID');

// retrieve an HTTP request header, with normalized, lowercase keys
$request->headers->get('host');
$request->headers->get('content_type');

$request->getMethod();    // GET, POST, PUT, DELETE, HEAD
$request->getLanguages(); // an array of languages the client accepts
```

You can also simulate a request::

```
$request = Request::create('/index.php?name=Fabien');
```

With the Response class, you can easily tweak the response::

```
<?php

$response = new Response();

$response->setContent('Hello world!');
$response->setStatusCode(200);
$response->headers->set('Content-Type', 'text/html');

// configure the HTTP cache headers
$response->setMaxAge(10);
```

.. tip::

To debug a Response, cast it to a string; it will return the HTTP
representation of the response (headers and content).

Last but not the least, these classes, like every other class in the Symfony code, have been audited_ for security issues by an independent company. And being an Open-Source project also means that many other developers around the world have read the code and have already fixed potential security problems. When was the last you ordered a professional security audit for your home-made framework?

Even something as simple as getting the client IP address can be insecure::

```
<?php

if ($myIp == $_SERVER['REMOTE_ADDR']) {
    // the client is a known one, so give it some more privilege
}
```

It works perfectly fine until you add a reverse proxy in front of the production servers; at this point, you will have to change your code to make it work on both your development machine (where you don't have a proxy) and your servers::

```
<?php

if ($myIp == $_SERVER['HTTP_X_FORWARDED_FOR'] || $myIp == $_SERVER['REMOTE_ADDR']) {
    // the client is a known one, so give it some more privilege
}
```

Using the Request::getClientIp() method would have given you the right behavior from day one (and it would have covered the case where you have chained proxies)::

```
<?php

$request = Request::createFromGlobals();

if ($myIp == $request->getClientIp()) {
    // the client is a known one, so give it some more privilege
}
```

And there is an added benefit: it is*secure* by default. What do I mean by secure? The $_SERVER['HTTP_X_FORWARDED_FOR'] value cannot be trusted as it can be manipulated by the end user when there is no proxy. So, if you are using this code in production without a proxy, it becomes trivially easy to abuse your system. That's not the case with the getClientIp() method as you must explicitly trust your reverse proxies by calling setTrustedProxies()::

```
<?php

Request::setTrustedProxies(array('10.0.0.1'));

if ($myIp == $request->getClientIp(true)) {
    // the client is a known one, so give it some more privilege
}
```

So, the getClientIp() method works securely in all circumstances. You can use it in all your projects, whatever the configuration is, it will behave correctly and safely. That's one of the goal of using a framework. If you were to write a framework from scratch, you would have to think about all these cases by yourself. Why not using a technology that already works?

.. note::

If you want to learn more about the HttpFoundation component, you can have a look at the `API`_ or read its dedicated `documentation`_ on the Symfony website.

Believe or not but we have our first framework. You can stop now if you want. Using just the Symfony2 HttpFoundation component already allows you to write better and more testable code. It also allows you to write code faster as many day-to-day problems have already been solved for you.

As a matter of fact, projects like Drupal have adopted the HttpFoundation component; if it works for them, it will probably work for you. Don't reinvent the wheel.

I've almost forgot to talk about one added benefit: using the HttpFoundation component is the start of better interoperability between all frameworks and applications using it (like Symfony2_, Drupal 8_, phpBB 4_, ezPublish 5_, Laravel_, Silex_, and more_).

.. _Twig: http://twig.sensiolabs.com/ .. _Symfony2 versus Flat PHP: http://symfony.com/doc/current/book/from_flat_php_to_symfony2.html .. _HTTP specification: http://tools.ietf.org/wg/httpbis/ .. _API: http://api.symfony.com/2.0/Symfony/Component/HttpFoundation.html .. _documentation: http://symfony.com/doc/current/components/http_foundation.html .. _audited: http://symfony.com/blog/symfony2-security-audit .. _Symfony2: http://symfony.com/ .. _Drupal 8: http://drupal.org/ .. _phpBB 4: http://www.phpbb.com/ .. _ezPublish 5: http://ez.no/ .. _Laravel: http://laravel.com/ .. _Silex: http://silex.sensiolabs.org/ .. _Midgard CMS: http://www.midgard-project.org/ .. _Zikula: http://zikula.org/ .. _autoloaded: http://php.net/autoload .. _PSR-0: https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md .. _more: http://symfony.com/components/HttpFoundation

The Front Controller

Up until now, our application is simplistic as there is only one page. To spice things up a little bit, let's go crazy and add another page that says goodbye::

```
<?php

// framework/bye.php

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

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

$request = Request::createFromGlobals();

$response = new Response('Goodbye!');
$response->send();
```

As you can see for yourself, much of the code is exactly the same as the one we have written for the first page. Let's extract the common code that we can share between all our pages. Code sharing sounds like a good plan to create our first "real" framework!

The PHP way of doing the refactoring would probably be the creation of an include file::

```
<?php

// framework/init.php

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

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

$request = Request::createFromGlobals();
$response = new Response();
```

Let's see it in action::

```
<?php

// framework/index.php

require_once __DIR__.'/init.php';

$input = $request->get('name', 'World');

$response->setContent(sprintf('Hello %s', htmlspecialchars($input, ENT_QUOTES, 'UTF-8')));
$response->send();
```

And for the "Goodbye" page::

```
<?php

// framework/bye.php
```

```
require_once __DIR__.'/init.php';

$response->setContent('Goodbye!');
$response->send();
```

We have indeed moved most of the shared code into a central place, but it does not feel like a good abstraction, does it? We still have the send() method for all pages, our pages do not look like templates, and we are still not able to test this code properly.

Moreover, adding a new page means that we need to create a new PHP script, which name is exposed to the end user via the URL (http://127.0.0.1:4321/bye.php): there is a direct mapping between the PHP script name and the client URL. This is because the dispatching of the request is done by the web server directly. It might be a good idea to move this dispatching to our code for better flexibility. This can be easily achieved by routing all client requests to a single PHP script.

.. tip::

Exposing a single PHP script to the end user is a design pattern called
the "`front controller`_".

Such a script might look like the following::

```
<?php

// framework/front.php

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

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

$request = Request::createFromGlobals();
$response = new Response();

$map = array(
    '/hello' => __DIR__.'/hello.php',
    '/bye'   => __DIR__.'/bye.php',
);

$path = $request->getPathInfo();
if (isset($map[$path])) {
    require $map[$path];
} else {
    $response->setStatusCode(404);
    $response->setContent('Not Found');
}

$response->send();
```

And here is for instance the new hello.php script::

```
<?php

// framework/hello.php

$input = $request->get('name', 'World');
```

```
$response->setContent(sprintf('Hello %s', htmlspecialchars($input, ENT_QUOTES, 'UTF-8')));
```

In the `front.php` script, `$map` associates URL paths with their corresponding PHP script paths.

As a bonus, if the client asks for a path that is not defined in the URL map, we return a custom 404 page; you are now in control of your website.

To access a page, you must now use the `front.php` script:

- http://127.0.0.1:4321/front.php/hello?name=Fabien
- http://127.0.0.1:4321/front.php/bye

`/hello` and `/bye` are the page paths.

.. tip::

Most web servers like Apache or nginx are able to rewrite the incoming URLs
and remove the front controller script so that your users will be able to
type ``http://127.0.0.1:4321/hello?name=Fabien``, which looks much better.

The trick is the usage of the `Request::getPathInfo()` method which returns the path of the Request by removing the front controller script name including its sub-directories (only if needed -- see above tip).

.. tip::

You don't even need to setup a web server to test the code. Instead,
replace the ``$request = Request::createFromGlobals();`` call to something
like ``$request = Request::create('/hello?name=Fabien');`` where the
argument is the URL path you want to simulate.

Now that the web server always access the same script (`front.php`) for all pages, we can secure the code further by moving all other PHP files outside the web root directory:

.. code-block:: text

```
example.com
â”œâ”€â”€ composer.json
â”‚   src
â”‚   â”œâ”€â”€ pages
â”‚       â”œâ”€â”€ hello.php
â”‚       â””â”€â”€ bye.php
â”œâ”€â”€ vendor
â””â”€â”€ web
    â””â”€â”€ front.php
```

Now, configure your web server root directory to point to `web/` and all other files won't be accessible from the client anymore.

To test your changes in a browser (`http://localhost:4321/?name=Fabien`), run the PHP built-in server:

.. code-block:: sh

```
$ php -S 127.0.0.1:4321 -t web/ web/front.php
```

.. note::

For this new structure to work, you will have to adjust some paths in
various PHP files; the changes are left as an exercise for the reader.

The last thing that is repeated in each page is the call to `setContent()`. We can convert all pages to "templates" by just echoing the content and calling the `setContent()` directly from the front controller script::

```
<?php

// example.com/web/front.php

// ...

$path = $request->getPathInfo();
if (isset($map[$path])) {
    ob_start();
    include $map[$path];
    $response->setContent(ob_get_clean());
} else {
    $response->setStatusCode(404);
    $response->setContent('Not Found');
}

// ...
```

And the `hello.php` script can now be converted to a template::

```
<!-- example.com/src/pages/hello.php -->

<?php $name = $request->get('name', 'World') ?>

Hello <?php echo htmlspecialchars($name, ENT_QUOTES, 'UTF-8') ?>
```

We have the first version of our framework::

```
<?php

// example.com/web/front.php

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

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

$request = Request::createFromGlobals();
$response = new Response();

$map = array(
    '/hello' => __DIR__.'/../src/pages/hello.php',
    '/bye'   => __DIR__.'/../src/pages/bye.php',
);

$path = $request->getPathInfo();
if (isset($map[$path])) {
    ob_start();
    include $map[$path];
    $response->setContent(ob_get_clean());
```

```
} else {
    $response->setStatusCode(404);
    $response->setContent('Not Found');
}

$response->send();
```

Adding a new page is a two step process: add an entry in the map and create a PHP template in `src/pages/`. From a template, get the Request data via the `$request` variable and tweak the Response headers via the `$response` variable.

.. note::

If you decide to stop here, you can probably enhance your framework by
extracting the URL map to a configuration file.

.. _front controller: http://symfony.com/doc/current/book/from_flat_php_to_symfony2.html#a-front-controller-to-the-rescue

The Routing Component

Before we start diving into the Routing component, let's refactor our current framework just a little to make templates even more readable::

```
<?php

// example.com/web/front.php

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

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

$request = Request::createFromGlobals();

$map = array(
    '/hello' => 'hello',
    '/bye'   => 'bye',
);

$path = $request->getPathInfo();
if (isset($map[$path])) {
    ob_start();
    extract($request->query->all(), EXTR_SKIP);
    include sprintf(__DIR__.'/../src/pages/%s.php', $map[$path]);
    $response = new Response(ob_get_clean());
} else {
    $response = new Response('Not Found', 404);
}

$response->send();
```

As we now extract the request query parameters, simplify the`hello.php` template as follows::

```
<!-- example.com/src/pages/hello.php -->

Hello <?php echo htmlspecialchars($name, ENT_QUOTES, 'UTF-8') ?>
```

Now, we are in good shape to add new features.

One very important aspect of any website is the form of its URLs. Thanks to the URL map, we have decoupled the URL from the code that generates the associated response, but it is not yet flexible enough. For instance, we might want to support dynamic paths to allow embedding data directly into the URL instead of relying on a query string:

```
# Before
/hello?name=Fabien

# After
/hello/Fabien
```

To support this feature, add the Symfony2 Routing component as a dependency:

.. code-block:: sh

```
$ php composer.phar require symfony/routing
```

Instead of an array for the URL map, the Routing component relies on a `RouteCollection` instance::

```
use Symfony\Component\Routing\RouteCollection;

$routes = new RouteCollection();
```

Let's add a route that describe the `/hello/SOMETHING` URL and add another one for the simple `/bye` one::

```
use Symfony\Component\Routing\Route;

$routes->add('hello', new Route('/hello/{name}', array('name' => 'World')));
$routes->add('bye', new Route('/bye'));
```

Each entry in the collection is defined by a name (`hello`) and a `Route` instance, which is defined by a route pattern (`/hello/{name}`) and an array of default values for route attributes (`array('name' => 'World')`).

.. note::

```
Read the official `documentation`_ for the Routing component to learn more
about its many features like URL generation, attribute requirements, HTTP
method enforcements, loaders for YAML or XML files, dumpers to PHP or
Apache rewrite rules for enhanced performance, and much more.
```

Based on the information stored in the `RouteCollection` instance, a `UrlMatcher` instance can match URL paths::

```
use Symfony\Component\Routing\RequestContext;
use Symfony\Component\Routing\Matcher\UrlMatcher;

$context = new RequestContext();
$context->fromRequest($request);
$matcher = new UrlMatcher($routes, $context);

$attributes = $matcher->match($request->getPathInfo());
```

The `match()` method takes a request path and returns an array of attributes (notice that the matched route is automatically stored under the special `_route` attribute)::

```
print_r($matcher->match('/bye'));
array (
  '_route' => 'bye',
);

print_r($matcher->match('/hello/Fabien'));
array (
  'name' => 'Fabien',
  '_route' => 'hello',
);

print_r($matcher->match('/hello'));
array (
  'name' => 'World',
  '_route' => 'hello',
);
```

.. note::

Even if we don't strictly need the request context in our examples, it is
used in real-world applications to enforce method requirements and more.

The URL matcher throws an exception when none of the routes match::

```
$matcher->match('/not-found');

// throws a Symfony\Component\Routing\Exception\ResourceNotFoundException
```

With this knowledge in mind, let's write the new version of our framework::

```
<?php

// example.com/web/front.php

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

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

$request = Request::createFromGlobals();
$routes = include __DIR__.'/../src/app.php';

$context = new Routing\RequestContext();
$context->fromRequest($request);
$matcher = new Routing\Matcher\UrlMatcher($routes, $context);

try {
    extract($matcher->match($request->getPathInfo()), EXTR_SKIP);
    ob_start();
    include sprintf(__DIR__.'/../src/pages/%s.php', $_route);

    $response = new Response(ob_get_clean());
} catch (Routing\Exception\ResourceNotFoundException $e) {
    $response = new Response('Not Found', 404);
} catch (Exception $e) {
    $response = new Response('An error occurred', 500);
}

$response->send();
```

There are a few new things in the code::

- Route names are used for template names;
- 500 errors are now managed correctly;
- Request attributes are extracted to keep our templates simple::

  Hello

- Route configuration has been moved to its own file:

  .. code-block:: php

```
<?php

// example.com/src/app.php

use Symfony\Component\Routing;

$routes = new Routing\RouteCollection(); $routes->add('hello', new
Routing\Route('/hello/{name}', array('name' => 'World'))); $routes->add('bye', new
Routing\Route('/bye'));

return $routes;
```

We now have a clear separation between the configuration (everything specific to our application in app.php) and the framework (the generic code that powers our application in front.php).

With less than 30 lines of code, we have a new framework, more powerful and more flexible than the previous one. Enjoy!

Using the Routing component has one big additional benefit: the ability to generate URLs based on Route definitions. When using both URL matching and URL generation in your code, changing the URL patterns should have no other impact. Want to know how to use the generator? Insanely easy::

```
use Symfony\Component\Routing;

$generator = new Routing\Generator\UrlGenerator($routes, $context);

echo $generator->generate('hello', array('name' => 'Fabien'));
// outputs /hello/Fabien
```

The code should be self-explanatory; and thanks to the context, you can even generate absolute URLs::

```
echo $generator->generate('hello', array('name' => 'Fabien'), true);
// outputs something like http://example.com/somewhere/hello/Fabien
```

.. tip::

Concerned about performance? Based on your route definitions, create a highly optimized URL matcher class that can replace the default ``UrlMatcher``::

```
    $dumper = new Routing\Matcher\Dumper\PhpMatcherDumper($routes);

    echo $dumper->dump();
```

.. _documentation: http://symfony.com/doc/current/components/routing.html

Templating

The astute reader has noticed that our framework hardcodes the way specific "code" (the templates) is run. For simple pages like the ones we have created so far, that's not a problem, but if you want to add more logic, you would be forced to put the logic into the template itself, which is probably not a good idea, especially if you still have the separation of concerns principle in mind.

Let's separate the template code from the logic by adding a new layer: the controller:*The controller's mission is to generate a Response based on the information conveyed by the client's Request.*

Change the template rendering part of the framework to read as follows::

```
<?php

// example.com/web/front.php

// ...

try {
    $request->attributes->add($matcher->match($request->getPathInfo()));
    $response = call_user_func('render_template', $request);
} catch (Routing\Exception\ResourceNotFoundException $e) {
    $response = new Response('Not Found', 404);
} catch (Exception $e) {
    $response = new Response('An error occurred', 500);
}
```

As the rendering is now done by an external function (`render_template()` here), we need to pass to it the attributes extracted from the URL. We could have passed them as an additional argument to `render_template()`, but instead, let's use another feature of the`Request` class called *attributes*: Request attributes is a way to attach additional information about the Request that is not directly related to the HTTP Request data.

You can now create the`render_template()` function, a generic controller that renders a template when there is no specific logic. To keep the same template as before, request attributes are extracted before the template is rendered::

```
function render_template($request)
{
    extract($request->attributes->all(), EXTR_SKIP);
    ob_start();
    include sprintf(__DIR__.'/../src/pages/%s.php', $_route);

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

As `render_template` is used as an argument to the PHP`call_user_func()` function, we can replace it with any valid PHP `callbacks`_. This allows us to use a function, an anonymous function, or a method of a class as a controller... your choice.

As a convention, for each route, the associated controller is configured via the`_controller` route

attribute::

```
$routes->add('hello', new Routing\Route('/hello/{name}', array(
    'name' => 'World',
    '_controller' => 'render_template',
)));

try {
    $request->attributes->add($matcher->match($request->getPathInfo()));
    $response = call_user_func($request->attributes->get('_controller'), $request);
} catch (Routing\Exception\ResourceNotFoundException $e) {
    $response = new Response('Not Found', 404);
} catch (Exception $e) {
    $response = new Response('An error occurred', 500);
}
```

A route can now be associated with any controller and of course, within a controller, you can still use the render_template() to render a template::

```
$routes->add('hello', new Routing\Route('/hello/{name}', array(
    'name' => 'World',
    '_controller' => function ($request) {
        return render_template($request);
    }
)));
```

This is rather flexible as you can change the Response object afterwards and you can even pass additional arguments to the template::

```
$routes->add('hello', new Routing\Route('/hello/{name}', array(
    'name' => 'World',
    '_controller' => function ($request) {
        // $foo will be available in the template
        $request->attributes->set('foo', 'bar');

        $response = render_template($request);

        // change some header
        $response->headers->set('Content-Type', 'text/plain');

        return $response;
    }
)));
```

Here is the updated and improved version of our framework::

```
<?php

// example.com/web/front.php

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

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

function render_template($request)
{
    extract($request->attributes->all(), EXTR_SKIP);
```

```
    ob_start();
    include sprintf(__DIR__.'/../src/pages/%s.php', $_route);

    return new Response(ob_get_clean());
}

$request = Request::createFromGlobals();
$routes = include __DIR__.'/../src/app.php';

$context = new Routing\RequestContext();
$context->fromRequest($request);
$matcher = new Routing\Matcher\UrlMatcher($routes, $context);

try {
    $request->attributes->add($matcher->match($request->getPathInfo()));
    $response = call_user_func($request->attributes->get('_controller'), $request);
} catch (Routing\Exception\ResourceNotFoundException $e) {
    $response = new Response('Not Found', 404);
} catch (Exception $e) {
    $response = new Response('An error occurred', 500);
}

$response->send();
```

To celebrate the birth of our new framework, let's create a brand new application that needs some simple logic. Our application has one page that says whether a given year is a leap year or not. When calling `/is_leap_year`, you get the answer for the current year, but you can also specify a year like in `/is_leap_year/2009`. Being generic, the framework does not need to be modified in any way, just create a new `app.php` file::

```
<?php

// example.com/src/app.php

use Symfony\Component\Routing;
use Symfony\Component\HttpFoundation\Response;

function is_leap_year($year = null) {
    if (null === $year) {
        $year = date('Y');
    }

    return 0 == $year % 400 || (0 == $year % 4 && 0 != $year % 100);
}

$routes = new Routing\RouteCollection();
$routes->add('leap_year', new Routing\Route('/is_leap_year/{year}', array(
    'year' => null,
    '_controller' => function ($request) {
        if (is_leap_year($request->attributes->get('year'))) {
            return new Response('Yep, this is a leap year!');
        }

        return new Response('Nope, this is not a leap year.');
    }
)));

return $routes;
```

The is_leap_year() function returns true when the given year is a leap year,false otherwise. If the year is null, the current year is tested. The controller is simple: it gets the year from the request attributes, pass it to the is_leap_year() function, and according to the return value it creates a new Response object.

As always, you can decide to stop here and use the framework as is; it's probably all you need to create simple websites like those fancy one-page websites_ and hopefully a few others.

.. _callbacks: http://php.net/callback#language.types.callback .. _websites: http://kottke.org/08/02/single-serving-sites

The HttpKernel Component: the Controller Resolver

You might think that our framework is already pretty solid and you are probably right. But let's see how we can improve it nonetheless.

Right now, all our examples use procedural code, but remember that controllers can be any valid PHP callbacks. Let's convert our controller to a proper class::

```
class LeapYearController
{
    public function indexAction($request)
    {
        if (is_leap_year($request->attributes->get('year'))) {
            return new Response('Yep, this is a leap year!');
        }

        return new Response('Nope, this is not a leap year.');
    }
}
```

Update the route definition accordingly::

```
$routes->add('leap_year', new Routing\Route('/is_leap_year/{year}', array(
    'year' => null,
    '_controller' => array(new LeapYearController(), 'indexAction'),
)));
```

The move is pretty straightforward and makes a lot of sense as soon as you create more pages but you might have noticed a non-desirable side-effect... The `LeapYearController` class is *always* instantiated, even if the requested URL does not match the `leap_year` route. This is bad for one main reason: performance wise, all controllers for all routes must now be instantiated for every request. It would be better if controllers were lazy-loaded so that only the controller associated with the matched route is instantiated.

To solve this issue, and a bunch more, let's install and use the HttpKernel component:

.. code-block:: sh

```
$ php composer.phar require symfony/http-kernel
```

The HttpKernel component has many interesting features, but the one we need right now is the *controller resolver*. A controller resolver knows how to determine the controller to execute and the arguments to pass to it, based on a Request object. All controller resolvers implement the following interface::

```
namespace Symfony\Component\HttpKernel\Controller;

interface ControllerResolverInterface
{
    function getController(Request $request);

    function getArguments(Request $request, $controller);
}
```

The getController() method relies on the same convention as the one we have defined earlier: the _controller request attribute must contain the controller associated with the Request. Besides the built-in PHP callbacks, getController() also supports strings composed of a class name followed by two colons and a method name as a valid callback, like 'class::method'::

```
$routes->add('leap_year', new Routing\Route('/is_leap_year/{year}', array(
    'year' => null,
    '_controller' => 'LeapYearController::indexAction',
)));
```

To make this code work, modify the framework code to use the controller resolver from HttpKernel::

```
use Symfony\Component\HttpKernel;

$resolver = new HttpKernel\Controller\ControllerResolver();

$controller = $resolver->getController($request);
$arguments = $resolver->getArguments($request, $controller);

$response = call_user_func_array($controller, $arguments);
```

.. note::

```
As an added bonus, the controller resolver properly handles the error
management for you: when you forget to define a ``_controller`` attribute
for a Route for instance.
```

Now, let's see how the controller arguments are guessed. getArguments() introspects the controller signature to determine which arguments to pass to it by using the native PHP reflection_.

The indexAction() method needs the Request object as an argument. getArguments() knows when to inject it properly if it is type-hinted correctly::

```
public function indexAction(Request $request)

// won't work
public function indexAction($request)
```

More interesting, getArguments() is also able to inject any Request attribute; the argument just needs to have the same name as the corresponding attribute::

```
public function indexAction($year)
```

You can also inject the Request and some attributes at the same time (as the matching is done on the argument name or a type hint, the arguments order does not matter)::

```
public function indexAction(Request $request, $year)

public function indexAction($year, Request $request)
```

Finally, you can also define default values for any argument that matches an optional attribute of the Request::

```
public function indexAction($year = 2012)
```

Let's just inject the $year request attribute for our controller::

```
class LeapYearController
{
    public function indexAction($year)
    {
        if (is_leap_year($year)) {
            return new Response('Yep, this is a leap year!');
        }

        return new Response('Nope, this is not a leap year.');
    }
}
```

The controller resolver also takes care of validating the controller callable and its arguments. In case of a problem, it throws an exception with a nice message explaining the problem (the controller class does not exist, the method is not defined, an argument has no matching attribute, ...).

.. note::

With the great flexibility of the default controller resolver, you might
wonder why someone would want to create another one (why would there be an
interface if not?). Two examples: in Symfony2, ``getController()`` is
enhanced to support `controllers as services`_; and in
`FrameworkExtraBundle`_, ``getArguments()`` is enhanced to support
parameter converters, where request attributes are converted to objects
automatically.

Let's conclude with the new version of our framework::

```
<?php

// example.com/web/front.php

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

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing;
use Symfony\Component\HttpKernel;

function render_template(Request $request)
{
    extract($request->attributes->all());
    ob_start();
    include sprintf(__DIR__.'/../src/pages/%s.php', $_route);

    return new Response(ob_get_clean());
}

$request = Request::createFromGlobals();
$routes = include __DIR__.'/../src/app.php';

$context = new Routing\RequestContext();
$context->fromRequest($request);
$matcher = new Routing\Matcher\UrlMatcher($routes, $context);
$resolver = new HttpKernel\Controller\ControllerResolver();

try {
    $request->attributes->add($matcher->match($request->getPathInfo()));
```

```
    $controller = $resolver->getController($request);
    $arguments = $resolver->getArguments($request, $controller);

    $response = call_user_func_array($controller, $arguments);
} catch (Routing\Exception\ResourceNotFoundException $e) {
    $response = new Response('Not Found', 404);
} catch (Exception $e) {
    $response = new Response('An error occurred', 500);
}

$response->send();
```

Think about it once more: our framework is more robust and more flexible than ever and it still has less than 40 lines of code.

.. _reflection: http://php.net/reflection .. _FrameworkExtraBundle: http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html .. _controllers as services: http://symfony.com/doc/current/cookbook/controller/service.html

The Separation of Concerns

One down-side of our framework right now is that we need to copy and paste the code in `front.php` each time we create a new website. 40 lines of code is not that much, but it would be nice if we could wrap this code into a proper class. It would bring us better *reusability* and easier testing to name just a few benefits.

If you have a closer look at the code, `front.php` has one input, the Request, and one output, the Response. Our framework class will follow this simple principle: the logic is about creating the Response associated with a Request.

Let's create our very own namespace for our framework: `Simplex`. Move the request handling logic into its own `Simplex\\Framework` class::

```
<?php

// example.com/src/Simplex/Framework.php

namespace Simplex;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\Exception\ResourceNotFoundException;
use Symfony\Component\HttpKernel\Controller\ControllerResolver;

class Framework
{
    protected $matcher;
    protected $resolver;

    public function __construct(UrlMatcher $matcher, ControllerResolver $resolver)
    {
        $this->matcher = $matcher;
        $this->resolver = $resolver;
    }

    public function handle(Request $request)
    {
        $this->matcher->getContext()->fromRequest($request);

        try {
            $request->attributes->add($this->matcher->match($request->getPathInfo()));

            $controller = $this->resolver->getController($request);
            $arguments = $this->resolver->getArguments($request, $controller);

            return call_user_func_array($controller, $arguments);
        } catch (ResourceNotFoundException $e) {
            return new Response('Not Found', 404);
        } catch (\Exception $e) {
            return new Response('An error occurred', 500);
        }
    }
}
```

And update example.com/web/front.php accordingly::

```
<?php

// example.com/web/front.php

// ...

$request = Request::createFromGlobals();
$routes = include __DIR__.'/../src/app.php';

$context = new Routing\RequestContext();
$matcher = new Routing\Matcher\UrlMatcher($routes, $context);
$resolver = new HttpKernel\Controller\ControllerResolver();

$framework = new Simplex\Framework($matcher, $resolver);
$response = $framework->handle($request);

$response->send();
```

To wrap up the refactoring, let's move everything but routes definition fromexample.com/src/app.php into yet another namespace: Calendar.

For the classes defined under theSimplex and Calendar namespaces to be autoloaded, update the composer.json file:

.. code-block:: javascript

```
{
    "require": {
        "symfony/http-foundation": "2.5.*",
        "symfony/routing": "2.5.*",
        "symfony/http-kernel": "2.5.*"
    },
    "autoload": {
        "psr-0": { "Simplex\\": "src/", "Calendar\\": "src/" }
    }
}
```

.. note::

For the Composer autoloader to be updated, run ``php composer.phar update``.

Move the controller to Calendar\\Controller\\LeapYearController::

```
<?php

// example.com/src/Calendar/Controller/LeapYearController.php

namespace Calendar\Controller;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Calendar\Model\LeapYear;

class LeapYearController
{
    public function indexAction(Request $request, $year)
```

```
    {
        $leapyear = new LeapYear();
        if ($leapyear->isLeapYear($year)) {
            return new Response('Yep, this is a leap year!');
        }

        return new Response('Nope, this is not a leap year.');
    }
}
```

And move the `is_leap_year()` function to its own class too::

```
<?php

// example.com/src/Calendar/Model/LeapYear.php

namespace Calendar\Model;

class LeapYear
{
    public function isLeapYear($year = null)
    {
        if (null === $year) {
            $year = date('Y');
        }

        return 0 == $year % 400 || (0 == $year % 4 && 0 != $year % 100);
    }
}
```

Don't forget to update the `example.com/src/app.php` file accordingly::

```
$routes->add('leap_year', new Routing\Route('/is_leap_year/{year}', array(
    'year' => null,
    '_controller' => 'Calendar\\Controller\\LeapYearController::indexAction',
)));
```

To sum up, here is the new file layout:

.. code-block:: text

```
example.com
â”œâ”€â”€ composer.json
â”‚   src
â”‚   â”œâ”€â”€ app.php
â”‚   â”‚â”€â”€ Simplex
â”‚   â”‚       â”‚â”€â”€ Framework.php
â”‚   â”‚â”€â”€ Calendar
â”‚   â”‚       â”‚â”€â”€ Controller
â”‚   â”‚       â”‚   â”‚â”€â”€ LeapYearController.php
â”‚   â”‚       â”‚â”€â”€ Model
â”‚   â”‚           â”‚â”€â”€ LeapYear.php
â”œâ”€â”€ vendor
â”‚â”€â”€ web
    â”‚â”€â”€ front.php
```

That's it! Our application has now four different layers and each of them has a well defined goal:

- `web/front.php`: The front controller; the only exposed PHP code that makes the interface with

the client (it gets the Request and sends the Response) and provides the boiler-plate code to initialize the framework and our application;

- src/Simplex: The reusable framework code that abstracts the handling of incoming Requests (by the way, it makes your controllers/templates easily testable -- more about that later on);
- src/Calendar: Our application specific code (the controllers and the model);
- src/app.php: The application configuration/framework customization.

Unit Testing

You might have noticed some subtle but nonetheless important bugs in the framework we built in the previous chapter. When creating a framework, you must be sure that it behaves as advertised. If not, all the applications based on it will exhibit the same bugs. The good news is that whenever you fix a bug, you are fixing a bunch of applications too.

Today's mission is to write unit tests for the framework we have created by using PHPUnit_. Create a PHPUnit configuration file in example.com/phpunit.xml.dist:

.. code-block:: xml

```
<?xml version="1.0" encoding="UTF-8"?>

<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false"
         syntaxCheck="false"
         bootstrap="vendor/autoload.php"
>
    <testsuites>
        <testsuite name="Test Suite">
            <directory>./tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

This configuration defines sensible defaults for most PHPUnit settings; more interesting, the autoloader is used to bootstrap the tests, and tests will be stored under the example.com/tests/ directory.

Now, let's write a test for "not found" resources. To avoid the creation of all dependencies when writing tests and to really just unit-test what we want, we are going to use test doubles_. Test doubles are easier to create when we rely on interfaces instead of concrete classes. Fortunately, Symfony2 provides such interfaces for core objects like the URL matcher and the controller resolver. Modify the framework to make use of them::

```
<?php

// example.com/src/Simplex/Framework.php

namespace Simplex;

// ...

use Symfony\Component\Routing\Matcher\UrlMatcherInterface;
use Symfony\Component\HttpKernel\Controller\ControllerResolverInterface;

class Framework
{
```

```
    protected $matcher;
    protected $resolver;

    public function __construct(UrlMatcherInterface $matcher, ControllerResolverInterface $resolver)
    {
        $this->matcher = $matcher;
        $this->resolver = $resolver;
    }

    // ...
}
```

We are now ready to write our first test::

```
<?php

// example.com/tests/Simplex/Tests/FrameworkTest.php

namespace Simplex\Tests;

use Simplex\Framework;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Exception\ResourceNotFoundException;

class FrameworkTest extends \PHPUnit_Framework_TestCase
{
    public function testNotFoundHandling()
    {
        $framework = $this->getFrameworkForException(new ResourceNotFoundException());

        $response = $framework->handle(new Request());

        $this->assertEquals(404, $response->getStatusCode());
    }

    protected function getFrameworkForException($exception)
    {
        $matcher = $this->getMock('Symfony\Component\Routing\Matcher\UrlMatcherInterface');
        $matcher
            ->expects($this->once())
            ->method('match')
            ->will($this->throwException($exception))
        ;
        $resolver = $this->getMock('Symfony\Component\HttpKernel\Controller\ControllerResolverInterface');

        return new Framework($matcher, $resolver);
    }
}
```

This test simulates a request that does not match any route. As such, the `match()` method returns a `ResourceNotFoundException` exception and we are testing that our framework converts this exception to a 404 response.

Executing this test is as simple as running `phpunit` from the `example.com` directory:

.. code-block:: bash

```
$ phpunit
```

.. note::

I do not explain how the code works in details as this is not the goal of
this book, but if you don't understand what the hell is going on, I highly
recommend you to read PHPUnit documentation on `test doubles`_.

After the test ran, you should see a green bar. If not, you have a bug either in the test or in the framework code!

Adding a unit test for any exception thrown in a controller is just as easy::

```
public function testErrorHandling()
{
    $framework = $this->getFrameworkForException(new \RuntimeException());

    $response = $framework->handle(new Request());

    $this->assertEquals(500, $response->getStatusCode());
}
```

Last, but not the least, let's write a test for when we actually have a proper Response::

```
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Controller\ControllerResolver;

public function testControllerResponse()
{
    $matcher = $this->getMock('Symfony\Component\Routing\Matcher\UrlMatcherInterface');
    $matcher
        ->expects($this->once())
        ->method('match')
        ->will($this->returnValue(array(
            '_route' => 'foo',
            'name' => 'Fabien',
            '_controller' => function ($name) {
                return new Response('Hello '.$name);
            }
        )))
    ;
    $resolver = new ControllerResolver();

    $framework = new Framework($matcher, $resolver);

    $response = $framework->handle(new Request());

    $this->assertEquals(200, $response->getStatusCode());
    $this->assertContains('Hello Fabien', $response->getContent());
}
```

In this test, we simulate a route that matches and returns a simple controller. We check that the response status is 200 and that its content is the one we have set in the controller.

To check that we have covered all possible use cases, run the PHPUnit test coverage feature (you need to enable XDebug_ first):

.. code-block:: bash

```
$ phpunit --coverage-html=cov/
```

Open example.com/cov/src_Simplex_Framework.php.html in a browser and check that all the lines for the Framework class are green (it means that they have been visited when the tests were executed).

Thanks to the simple object-oriented code that we have written so far, we have been able to write unit-tests to cover all possible use cases of our framework; test doubles ensured that we were actually testing our code and not Symfony2 code.

Now that we are confident (again) about the code we have written, we can safely think about the next batch of features we want to add to our framework.

.. _PHPUnit: https://phpunit.de/manual/current/en/ .. _test doubles: https://phpunit.de/manual/current/en/test-doubles.html .. _XDebug: http://xdebug.org/

The EventDispatcher Component

Our framework is still missing a major characteristic of any good framework: *extensibility*. Being extensible means that the developer should be able to easily hook into the framework life cycle to modify the way the request is handled.

What kind of hooks are we talking about? Authentication or caching for instance. To be flexible, hooks must be plug-and-play; the ones you "register" for an application are different from the next one depending on your specific needs. Many software have a similar concept like Drupal or Wordpress. In some languages, there is even a standard like WSGI_ in Python or Rack_ in Ruby.

As there is no standard for PHP, we are going to use a well-known design pattern, the *Observer*, to allow any kind of behaviors to be attached to our framework; the Symfony2 EventDispatcher Component implements a lightweight version of this pattern:

.. code-block:: sh

```
$ php composer.phar require symfony/event-dispatcher
```

How does it work? The *dispatcher*, the central object of the event dispatcher system, notifies *listeners* of an *event* dispatched to it. Put another way: your code dispatches an event to the dispatcher, the dispatcher notifies all registered listeners for the event, and each listener do whatever it wants with the event.

As an example, let's create a listener that transparently adds the Google Analytics code to all responses.

To make it work, the framework must dispatch an event just before returning the Response instance::

```
<?php

// example.com/src/Simplex/Framework.php

namespace Simplex;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Matcher\UrlMatcherInterface;
use Symfony\Component\Routing\Exception\ResourceNotFoundException;
use Symfony\Component\HttpKernel\Controller\ControllerResolverInterface;
use Symfony\Component\EventDispatcher\EventDispatcher;

class Framework
{
    protected $matcher;
    protected $resolver;
    protected $dispatcher;

    public function __construct(EventDispatcher $dispatcher, UrlMatcherInterface $matcher, ControllerResolverInterface $resolver)
    {
        $this->matcher = $matcher;
        $this->resolver = $resolver;
        $this->dispatcher = $dispatcher;
    }

    public function handle(Request $request)
    {
        $this->matcher->getContext()->fromRequest($request);
```

```
        try {
            $request->attributes->add($this->matcher->match($request->getPathInfo()));

            $controller = $this->resolver->getController($request);
            $arguments = $this->resolver->getArguments($request, $controller);

            $response = call_user_func_array($controller, $arguments);
        } catch (ResourceNotFoundException $e) {
            $response = new Response('Not Found', 404);
        } catch (\Exception $e) {
            $response = new Response('An error occurred', 500);
        }

        // dispatch a response event
        $this->dispatcher->dispatch('response', new ResponseEvent($response, $request));

        return $response;
    }
}
```

Each time the framework handles a Request, a`ResponseEvent` event is now dispatched::

```
<?php

// example.com/src/Simplex/ResponseEvent.php

namespace Simplex;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\EventDispatcher\Event;

class ResponseEvent extends Event
{
    private $request;
    private $response;

    public function __construct(Response $response, Request $request)
    {
        $this->response = $response;
        $this->request = $request;
    }

    public function getResponse()
    {
        return $this->response;
    }

    public function getRequest()
    {
        return $this->request;
    }
}
```

The last step is the creation of the dispatcher in the front controller and the registration of a listener for the `response` event::

```
<?php

// example.com/web/front.php

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

// ...
```

```
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();
$dispatcher->addListener('response', function (Simplex\ResponseEvent $event) {
    $response = $event->getResponse();

    if ($response->isRedirection()
        || ($response->headers->has('Content-Type') && false === strpos($response->headers->get('Content-Type'), 'html'))
        || 'html' !== $event->getRequest()->getRequestFormat()
    ) {
        return;
    }

    $response->setContent($response->getContent().'GA CODE');
});

$framework = new Simplex\Framework($dispatcher, $matcher, $resolver);
$response = $framework->handle($request);

$response->send();
```

.. note::

The listener is just a proof of concept and you should add the Google Analytics code just before the body tag.

As you can see, `addListener()` associates a valid PHP callback to a named event (`response`); the event name must be the same as the one used in the `dispatch()` call.

In the listener, we add the Google Analytics code only if the response is not a redirection, if the requested format is HTML, and if the response content type is HTML (these conditions demonstrate the ease of manipulating the Request and Response data from your code).

So far so good, but let's add another listener on the same event. Let's say that I want to set the `Content-Length` of the Response if it is not already set::

```
$dispatcher->addListener('response', function (Simplex\ResponseEvent $event) {
    $response = $event->getResponse();
    $headers = $response->headers;

    if (!$headers->has('Content-Length') && !$headers->has('Transfer-Encoding')) {
        $headers->set('Content-Length', strlen($response->getContent()));
    }
});
```

Depending on whether you have added this piece of code before the previous listener registration or after it, you will have the wrong or the right value for the `Content-Length` header. Sometimes, the order of the listeners matter but by default, all listeners are registered with the same priority, 0. To tell the dispatcher to run a listener early, change the priority to a positive number; negative numbers can be used for low priority listeners. Here, we want the `Content-Length` listener to be executed last, so change the priority to `-255`::

```
$dispatcher->addListener('response', function (Simplex\ResponseEvent $event) {
    $response = $event->getResponse();
    $headers = $response->headers;

    if (!$headers->has('Content-Length') && !$headers->has('Transfer-Encoding')) {
        $headers->set('Content-Length', strlen($response->getContent()));
    }
}, -255);
```

.. tip::

When creating your framework, think about priorities (reserve some numbers
for internal listeners for instance) and document them thoroughly.

Let's refactor the code a bit by moving the Google listener to its own class::

```
<?php

// example.com/src/Simplex/GoogleListener.php

namespace Simplex;

class GoogleListener
{
    public function onResponse(ResponseEvent $event)
    {
        $response = $event->getResponse();

        if ($response->isRedirection()
            || ($response->headers->has('Content-Type') && false === strpos($response->headers->get('Content-Type'), 'html'))
            || 'html' !== $event->getRequest()->getRequestFormat()
        ) {
            return;
        }

        $response->setContent($response->getContent().'GA CODE');
    }
}
```

And do the same with the other listener::

```
<?php

// example.com/src/Simplex/ContentLengthListener.php

namespace Simplex;

class ContentLengthListener
{
    public function onResponse(ResponseEvent $event)
    {
        $response = $event->getResponse();
        $headers = $response->headers;

        if (!$headers->has('Content-Length') && !$headers->has('Transfer-Encoding')) {
            $headers->set('Content-Length', strlen($response->getContent()));
        }
    }
}
```

Our front controller should now look like the following::

```
$dispatcher = new EventDispatcher();
$dispatcher->addListener('response', array(new Simplex\ContentLengthListener(), 'onResponse'), -255);
$dispatcher->addListener('response', array(new Simplex\GoogleListener(), 'onResponse'));
```

Even if the code is now nicely wrapped in classes, there is still a slight issue: the knowledge of the priorities is "hardcoded" in the front controller, instead of being in the listeners themselves. For each application, you have to remember to set the appropriate priorities. Moreover, the listener method names are also exposed here, which means that refactoring our listeners would mean changing all the applications that rely on those listeners. Of course, there is a solution: use subscribers instead of

listeners::

```
$dispatcher = new EventDispatcher();
$dispatcher->addSubscriber(new Simplex\ContentLengthListener());
$dispatcher->addSubscriber(new Simplex\GoogleListener());
```

A subscriber knows about all the events it is interested in and pass this information to the dispatcher via the getSubscribedEvents() method. Have a look at the new version of the GoogleListener::

```
<?php

// example.com/src/Simplex/GoogleListener.php

namespace Simplex;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class GoogleListener implements EventSubscriberInterface
{
    // ...

    public static function getSubscribedEvents()
    {
        return array('response' => 'onResponse');
    }
}
```

And here is the new version of ContentLengthListener::

```
<?php

// example.com/src/Simplex/ContentLengthListener.php

namespace Simplex;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class ContentLengthListener implements EventSubscriberInterface
{
    // ...

    public static function getSubscribedEvents()
    {
        return array('response' => array('onResponse', -255));
    }
}
```

.. tip::

A single subscriber can host as many listeners as you want on as many events as needed.

To make your framework truly flexible, don't hesitate to add more events; and to make it more awesome out of the box, add more listeners. Again, this book is not about creating a generic framework, but one that is tailored to your needs. Stop whenever you see fit, and further evolve the code from there.

.. _WSGI: http://www.python.org/dev/peps/pep-0333/#middleware-components-that-play-both-sides .. _Rack: http://rack.rubyforge.org/

The HttpKernel Component: HttpKernelInterface

In the conclusion of the second chapter of this book, I've talked about one great benefit of using the Symfony2 components: the *interoperability* between all frameworks and applications using them. Let's do a big step towards this goal by making our framework implement HttpKernelInterface::

```
namespace Symfony\Component\HttpKernel;

interface HttpKernelInterface
{
    /**
     * @return Response A Response instance
     */
    function handle(Request $request, $type = self::MASTER_REQUEST, $catch = true);
}
```

HttpKernelInterface is probably the most important piece of code in the HttpKernel component, no kidding. Frameworks and applications that implement this interface are fully interoperable. Moreover, a lot of great features will come with it for free.

Update your framework so that it implements this interface::

```
<?php

// example.com/src/Framework.php

// ...

use Symfony\Component\HttpKernel\HttpKernelInterface;

class Framework implements HttpKernelInterface
{
    // ...

    public function handle(Request $request, $type = HttpKernelInterface::MASTER_REQUEST, $catch = true)
    {
        // ...
    }
}
```

Even if this change looks trivial, it brings us a lot! Let's talk about one of the most impressive one: transparent HTTP caching_ support.

The HttpCache class implements a fully-featured reverse proxy, written in PHP; it implements HttpKernelInterface and wraps another HttpKernelInterface instance::

```
// example.com/web/front.php

$framework = new Simplex\Framework($dispatcher, $matcher, $resolver);
$framework = new HttpKernel\HttpCache\HttpCache($framework, new HttpKernel\HttpCache\Store(__DIR__.'/../cache'));

$framework->handle($request)->send();
```

That's all it takes to add HTTP caching support to our framework. Isn't it amazing?

Configuring the cache needs to be done via HTTP cache headers. For instance, to cache a response for 10 seconds, use the Response::setTtl() method::

```
// example.com/src/Calendar/Controller/LeapYearController.php

public function indexAction(Request $request, $year)
{
    $leapyear = new LeapYear();
    if ($leapyear->isLeapYear($year)) {
        $response = new Response('Yep, this is a leap year!');
    } else {
        $response = new Response('Nope, this is not a leap year.');
    }

    $response->setTtl(10);

    return $response;
}
```

.. tip::

If, like me, you are running your framework from the command line by simulating requests (``Request::create('/is_leap_year/2012')``), you can easily debug Response instances by dumping their string representation (``echo $response;``) as it displays all headers as well as the response content.

To validate that it works correctly, add a random number to the response content and check that the number only changes every 10 seconds::

```
$response = new Response('Yep, this is a leap year! '.rand());
```

.. note::

When deploying to your production environment, keep using the Symfony2 reverse proxy (great for shared hosting) or even better, switch to a more efficient reverse proxy like `Varnish`_.

Using HTTP cache headers to manage your application cache is very powerful and allows you to tune finely your caching strategy as you can use both the expiration and the validation models of the HTTP specification. If you are not comfortable with these concepts, I highly recommend you to read the HTTP caching_ chapter of the Symfony2 documentation.

The Response class contains many other methods that let you configure the HTTP cache very easily. One of the most powerful is setCache() as it abstracts the most frequently used caching strategies into one simple array::

```
$date = date_create_from_format('Y-m-d H:i:s', '2005-10-15 10:00:00');

$response->setCache(array(
    'public'        => true,
    'etag'          => 'abcde',
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
```

```
));

// it is equivalent to the following code
$response->setPublic();
$response->setEtag('abcde');
$response->setLastModified($date);
$response->setMaxAge(10);
$response->setSharedMaxAge(10);
```

When using the validation model, the `isNotModified()` method allows you to easily cut on the response time by short-circuiting the response generation as early as possible::

```
$response->setETag('whatever_you_compute_as_an_etag');

if ($response->isNotModified($request)) {
    return $response;
}
$response->setContent('The computed content of the response');

return $response;
```

Using HTTP caching is great, but what if you cannot cache the whole page? What if you can cache everything but some sidebar that is more dynamic that the rest of the content? Edge Side Includes (`ESI`_) to the rescue! Instead of generating the whole content in one go, ESI allows you to mark a region of a page as being the content of a sub-request call::

```
This is the content of your page

Is 2012 a leap year? <esi:include src="/leapyear/2012" />

Some other content
```

For ESI tags to be supported by HttpCache, you need to pass it an instance of the `ESI` class. The `ESI` class automatically parses ESI tags and makes sub-requests to convert them to their proper content::

```
$framework = new HttpKernel\HttpCache\HttpCache(
    $framework,
    new HttpKernel\HttpCache\Store(__DIR__.'/../cache'),
    new HttpKernel\HttpCache\ESI()
);
```

.. note::

```
For ESI to work, you need to use a reverse proxy that supports it like the
Symfony2 implementation. `Varnish`_ is the best alternative and it is
Open-Source.
```

When using complex HTTP caching strategies and/or many ESI include tags, it can be hard to understand why and when a resource should be cached or not. To ease debugging, you can enable the debug mode::

```
$framework = new HttpCache($framework, new Store(__DIR__.'/../cache'), new ESI(), array('debug' => true));
```

The debug mode adds a `X-Symfony-Cache` header to each response that describes what the cache layer did:

.. code-block:: text

X-Symfony-Cache: GET /is_leap_year/2012: stale, invalid, store

X-Symfony-Cache: GET /is_leap_year/2012: fresh

HttpCache has many features like support for the stale-while-revalidate and stale-if-error HTTP Cache-Control extensions as defined in RFC 5861.

With the addition of a single interface, our framework can now benefit from the many features built into the HttpKernel component; HTTP caching being just one of them but an important one as it can make your applications fly!

.. _HTTP caching: http://symfony.com/doc/current/book/http_cache.html .. _ESI: http://en.wikipedia.org/wiki/Edge_Side_Includes .. _Varnish: https://www.varnish-cache.org/

The HttpKernel Component: The HttpKernel Class

If you were to use our framework right now, you would probably have to add support for custom error messages. We do have 404 and 500 error support but the responses are hardcoded in the framework itself. Making them customizable is easy enough though: dispatch a new event and listen to it. Doing it right means that the listener has to call a regular controller. But what if the error controller throws an exception? You will end up in an infinite loop. There should be an easier way, right?

Enter the HttpKernel class. Instead of solving the same problem over and over again and instead of reinventing the wheel each time, the HttpKernel class is a generic, extensible, and flexible implementation of HttpKernelInterface.

This class is very similar to the framework class we have written so far: it dispatches events at some strategic points during the handling of the request, it uses a controller resolver to choose the controller to dispatch the request to, and as an added bonus, it takes care of edge cases and provides great feedback when a problem arises.

Here is the new framework code::

```
<?php

// example.com/src/Simplex/Framework.php

namespace Simplex;

use Symfony\Component\HttpKernel\HttpKernel;

class Framework extends HttpKernel
{
}
```

And the new front controller::

```
<?php

// example.com/web/front.php

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

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing;
use Symfony\Component\HttpKernel;
use Symfony\Component\EventDispatcher\EventDispatcher;

$request = Request::createFromGlobals();
$routes = include __DIR__.'/../src/app.php';

$context = new Routing\RequestContext();
$matcher = new Routing\Matcher\UrlMatcher($routes, $context);
$resolver = new HttpKernel\Controller\ControllerResolver();
```

```
$dispatcher = new EventDispatcher();
$dispatcher->addSubscriber(new HttpKernel\EventListener\RouterListener($matcher));

$framework = new Simplex\Framework($dispatcher, $resolver);

$response = $framework->handle($request);
$response->send();
```

RouterListener is an implementation of the same logic we had in our framework: it matches the incoming request and populates the request attributes with route parameters.

Our code is now much more concise and surprisingly more robust and more powerful than ever. For instance, use the built-in ExceptionListener to make your error management configurable::

```
$errorHandler = function (HttpKernel\Exception\FlattenException $exception) {
    $msg = 'Something went wrong! ('.$exception->getMessage().')';

    return new Response($msg, $exception->getStatusCode());
};
$dispatcher->addSubscriber(new HttpKernel\EventListener\ExceptionListener($errorHandler));
```

ExceptionListener gives you a FlattenException instance instead of the thrown Exception instance to ease exception manipulation and display. It can take any valid controller as an exception handler, so you can create an ErrorController class instead of using a Closure::

```
$listener = new HttpKernel\EventListener\ExceptionListener('Calendar\\Controller\\ErrorController::exceptionAction');
$dispatcher->addSubscriber($listener);
```

The error controller reads as follows::

```
<?php

// example.com/src/Calendar/Controller/ErrorController.php

namespace Calendar\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Exception\FlattenException;

class ErrorController
{
    public function exceptionAction(FlattenException $exception)
    {
        $msg = 'Something went wrong! ('.$exception->getMessage().')';

        return new Response($msg, $exception->getStatusCode());
    }
}
```

VoilÃ ! Clean and customizable error management without efforts. And of course, if your controller throws an exception, HttpKernel will handle it nicely.

In chapter two, we talked about theResponse::prepare() method, which ensures that a Response is compliant with the HTTP specification. It is probably a good idea to always call it just before sending the Response to the client; that's what the ResponseListener does::

```
$dispatcher->addSubscriber(new HttpKernel\EventListener\ResponseListener('UTF-8'));
```

This one was easy too! Let's take another one: do you want out of the box support for streamed responses? Just subscribe to StreamedResponseListener::

```
$dispatcher->addSubscriber(new HttpKernel\EventListener\StreamedResponseListener());
```

And in your controller, return a StreamedResponse instance instead of a Response instance.

.. tip::

```
Read the `Internals`_ chapter of the Symfony2 documentation to learn more
about the events dispatched by HttpKernel and how they allow you to change
the flow of a request.
```

Now, let's create a listener, one that allows a controller to return a string instead of a full Response object::

```
class LeapYearController
{
    public function indexAction(Request $request, $year)
    {
        $leapyear = new LeapYear();
        if ($leapyear->isLeapYear($year)) {
            return 'Yep, this is a leap year! ';
        }

        return 'Nope, this is not a leap year.';
    }
}
```

To implement this feature, we are going to listen to the kernel.view event, which is triggered just after the controller has been called. Its goal is to convert the controller return value to a proper Response instance, but only if needed::

```
<?php

// example.com/src/Simplex/StringResponseListener.php

namespace Simplex;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
use Symfony\Component\HttpFoundation\Response;

class StringResponseListener implements EventSubscriberInterface
{
    public function onView(GetResponseForControllerResultEvent $event)
    {
        $response = $event->getControllerResult();

        if (is_string($response)) {
            $event->setResponse(new Response($response));
        }
    }

    public static function getSubscribedEvents()
    {
```

```
        return array('kernel.view' => 'onView');
    }
}
```

The code is simple because the `kernel.view` event is only triggered when the controller return value is not a Response and because setting the response on the event stops the event propagation (our listener cannot interfere with other view listeners).

Don't forget to register it in the front controller::

```
$dispatcher->addSubscriber(new Simplex\StringResponseListener());
```

.. note::

If you forget to register the subscriber, HttpKernel will throw an exception with a nice message: ``The controller must return a response (Nope, this is not a leap year. given).``.

At this point, our whole framework code is as compact as possible and it is mainly composed of an assembly of existing libraries. Extending is a matter of registering event listeners/subscribers.

Hopefully, you now have a better understanding of why the simple looking `HttpKernelInterface` is so powerful. Its default implementation, `HttpKernel`, gives you access to a lot of cool features, ready to be used out of the box, with no efforts. And because HttpKernel is actually the code that powers the Symfony2 and Silex frameworks, you have the best of both worlds: a custom framework, tailored to your needs, but based on a rock-solid and well maintained low-level architecture that has been proven to work for many websites; a code that has been audited for security issues and that has proven to scale well.

.. _`Internals`: http://symfony.com/doc/current/book/internals.html#events

The DependencyInjection Component

In the previous chapter, we emptied the Simplex\Framework class by extending the HttpKernel class from the eponymous component. Seeing this empty class, you might be tempted to move some code from the front controller to it::

```
<?php

// example.com/src/Simplex/Framework.php

namespace Simplex;

use Symfony\Component\Routing;
use Symfony\Component\HttpKernel;
use Symfony\Component\EventDispatcher\EventDispatcher;

class Framework extends HttpKernel\HttpKernel
{
    public function __construct($routes)
    {
        $context = new Routing\RequestContext();
        $matcher = new Routing\Matcher\UrlMatcher($routes, $context);
        $resolver = new HttpKernel\Controller\ControllerResolver();

        $dispatcher = new EventDispatcher();
        $dispatcher->addSubscriber(new HttpKernel\EventListener\RouterListener($matcher));
        $dispatcher->addSubscriber(new HttpKernel\EventListener\ResponseListener('UTF-8'));

        parent::__construct($dispatcher, $resolver);
    }
}
```

The front controller code would become more concise::

```
<?php

// example.com/web/front.php

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

use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
$routes = include __DIR__.'/../src/app.php';

$framework = new Simplex\Framework($routes);

$framework->handle($request)->send();
```

Having a concise front controller allows you to have several front controllers for a single application. Why would it be useful? To allow having different configuration for the development environment and the production one for instance. In the development environment, you might want to have error reporting turned on and errors displayed in the browser to ease debugging::

```
ini_set('display_errors', 1);
error_reporting(-1);
```

... but you certainly won't want that same configuration on the production environment. Having two different front controllers gives you the opportunity to have a slightly different configuration for each of them.

So, moving code from the front controller to the framework class makes our framework more configurable, but at the same time, it introduces a lot of issues:

- We are not able to register custom listeners anymore as the dispatcher is not available outside the Framework class (an easy workaround could be the adding of a `Framework::getEventDispatcher()` method);
- We have lost the flexibility we had before; you cannot change the implementation of the `UrlMatcher` or of the `ControllerResolver` anymore;
- Related to the previous point, we cannot test our framework easily anymore as it's impossible to mock internal objects;
- We cannot change the charset passed to `ResponseListener` anymore (a workaround could be to pass it as a constructor argument).

The previous code did not exhibit the same issues because we used dependency injection; all dependencies of our objects were injected into their constructors (for instance, the event dispatchers were injected into the framework so that we had total control of its creation and configuration).

Does it mean that we have to make a choice between flexibility, customization, ease of testing and not to copy and paste the same code into each application front controller? As you might expect, there is a solution. We can solve all these issues and some more by using the Symfony2 dependency injection container:

.. code-block:: sh

```
$ php composer.phar require symfony/dependency-injection
```

Create a new file to host the dependency injection container configuration::

```
<?php

// example.com/src/container.php

use Symfony\Component\DependencyInjection;
use Symfony\Component\DependencyInjection\Reference;

$sc = new DependencyInjection\ContainerBuilder();
$sc->register('context', 'Symfony\Component\Routing\RequestContext');
$sc->register('matcher', 'Symfony\Component\Routing\Matcher\UrlMatcher')
    ->setArguments(array($routes, new Reference('context')))
;
$sc->register('resolver', 'Symfony\Component\HttpKernel\Controller\ControllerResolver');

$sc->register('listener.router', 'Symfony\Component\HttpKernel\EventListener\RouterListener')
    ->setArguments(array(new Reference('matcher')))
;
$sc->register('listener.response', 'Symfony\Component\HttpKernel\EventListener\ResponseListener')
    ->setArguments(array('UTF-8'))
```

```
;
$sc->register('listener.exception', 'Symfony\Component\HttpKernel\EventListener\ExceptionListener')
    ->setArguments(array('Calendar\\Controller\\ErrorController::exceptionAction'))
;
$sc->register('dispatcher', 'Symfony\Component\EventDispatcher\EventDispatcher')
    ->addMethodCall('addSubscriber', array(new Reference('listener.router')))
    ->addMethodCall('addSubscriber', array(new Reference('listener.response')))
    ->addMethodCall('addSubscriber', array(new Reference('listener.exception')))
;
$sc->register('framework', 'Simplex\Framework')
    ->setArguments(array(new Reference('dispatcher'), new Reference('resolver')))
;

return $sc;
```

The goal of this file is to configure your objects and their dependencies. Nothing is instantiated during this configuration step. This is purely a static description of the objects you need to manipulate and how to create them. Objects will be created on-demand when you access them from the container or when the container needs them to create other objects.

For instance, to create the router listener, we tell Symfony that its class name is `Symfony\Component\HttpKernel\EventListener\RouterListener`, and that its constructor takes a matcher object (`new Reference('matcher')`). As you can see, each object is referenced by a name, a string that uniquely identifies each object. The name allows us to get an object and to reference it in other object definitions.

.. note::

By default, every time you get an object from the container, it returns
the exact same instance. That's because a container manages your "global"
objects.

The front controller is now only about wiring everything together::

```
<?php

// example.com/web/front.php

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

use Symfony\Component\HttpFoundation\Request;

$routes = include __DIR__.'/../src/app.php';
$sc = include __DIR__.'/../src/container.php';

$request = Request::createFromGlobals();

$response = $sc->get('framework')->handle($request);

$response->send();
```

As all the objects are now created in the dependency injection container, the framework code should be the previous simple version::

```
<?php

// example.com/src/Simplex/Framework.php
```

```
namespace Simplex;

use Symfony\Component\HttpKernel\HttpKernel;

class Framework extends HttpKernel
{
}
```

.. note::

If you want a light alternative for your container, consider `Pimple`_, a
simple dependency injection container in about 60 lines of PHP code.

Now, here is how you can register a custom listener in the front controller::

```
$sc->register('listener.string_response', 'Simplex\StringResponseListener');
$sc->getDefinition('dispatcher')
    ->addMethodCall('addSubscriber', array(new Reference('listener.string_response')))
;
```

Beside describing your objects, the dependency injection container can also be configured via parameters. Let's create one that defines if we are in debug mode or not::

```
$sc->setParameter('debug', true);

echo $sc->getParameter('debug');
```

These parameters can be used when defining object definitions. Let's make the charset configurable::

```
$sc->register('listener.response', 'Symfony\Component\HttpKernel\EventListener\ResponseListener')
    ->setArguments(array('%charset%'))
;
```

After this change, you must set the charset before using the response listener object::

```
$sc->setParameter('charset', 'UTF-8');
```

Instead of relying on the convention that the routes are defined by the `$routes` variables, let's use a parameter again::

```
$sc->register('matcher', 'Symfony\Component\Routing\Matcher\UrlMatcher')
    ->setArguments(array('%routes%', new Reference('context')))
;
```

And the related change in the front controller::

```
$sc->setParameter('routes', include __DIR__.'/../src/app.php');
```

We have obviously barely scratched the surface of what you can do with the container: from class names as parameters, to overriding existing object definitions, from scope support to dumping a container to a plain PHP class, and much more. The Symfony dependency injection container is really powerful and is able to manage any kind of PHP class.

Don't yell at me if you don't want to use a dependency injection container in your framework. If you don't like it, don't use it. It's your framework, not mine.

This is (already) the last chapter of this book on creating a framework on top of the Symfony2 components. I'm aware that many topics have not been covered in great details, but hopefully it gives you enough information to get started on your own and to better understand how the Symfony2 framework works internally.

If you want to learn more, I highly recommend you to read the source code of the Silex_ micro-framework, and especially its Application_ class.

Have fun!

.. _Pimple: https://github.com/fabpot/Pimple .. _Silex: http://silex.sensiolabs.org/ .. _Application: https://github.com/fabpot/Silex/blob/master/src/Silex/Application.php