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