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