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/