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 the`Response::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