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/