FrameWork2 using Symfony Components

Written by Hong Le, Last updated 07.2025
Code: https://github.com/hong1234/lightFrame2.git

We'll see how to build a minimal PHP framework using these components

The HttpFoundation component

composer require symfony/http-foundation 2.5.*

Request allows us to deal with the HTTP request information such as the requested URI or the client headers, abstracting default PHP globals ($_GET, $_POST, etc.).

```
// request
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
$path = $request->getPathInfo();

switch($path) {
   case '/':
      …
   case '/about':
      …
   default:
      …
}
```

Response is used to send back response HTTP headers and data to the client, instead of using header or echo as we would in "classic" PHP.

```
// response
use Symfony\Component\HttpFoundation\Response;

$response = new Response();
$response->setContent('Not found !');
// $response->setStatusCode(Response::HTTP_NOT_FOUND);
$response->send();
```

The Routing component

composer require symfony/routing 2.5.*

We need a flexible and powerful system for routing in application

```
// route
use Symfony\Component\Routing\Route;
$route = new Route($path, array('controller' => $controller));

// routes init
use Symfony\Component\Routing\RouteCollection;
$routes = new RouteCollection();
$routes->add($path, $route);
```

```
// request context
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\RequestContext;

$request = Request::createFromGlobals();
$context = new RequestContext();
$context->fromRequest($request);

// set matcher
use Symfony\Component\Routing\Matcher\UrlMatcher;
$matcher = new UrlMatcher($routes, $context);

// let matching
use Symfony\Component\Routing\Exception\ResourceNotFoundException;

try {
   $attributes = $matcher->match($request->getPathInfo());
} catch (ResourceNotFoundException $e) {
   …
}

// get function to call
$controller = $attributes['controller'];
// unset($attributes['controller']);
```

The EventDispatcher component

composer require symfony/event-dispatcher 2.5

The way to hook into the request lifecycle and to change it. A good example is the security layer intercepting a request which attempts to load an URL between a firewall.

At the core of it, there is the *EventDispatcher* class, which registers *listeners of a particular event*. *A listener* can be any valid PHP callable function or method. When the dispatcher is notified of an event, all known listeners of this event are called.

```
// event object
use Symfony\Component\EventDispatcher\Event;
class RequestEvent extends Event { ... }
$eventObject = new RequestEvent();

// event-handler (listener)
function (Event $eventObject) { ... }

// event Dispatcher
use Symfony\Component\EventDispatcher\EventDispatcher;
$dispatcher = new EventDispatcher();

// register event-handler(listener to the event) with dispatcher
$dispatcher->addListener(
   $eventName,
   $event-handler // callback function
);

// trigge event at a point in request handling pipeline
$dispatcher->dispatch($eventName, $eventObject);
```

The HttpKernel component

composer require symfony/http-kernel 2.5.*

It would be better to wrap the framework logic into another class, which would become the “core” of our framework.

It is intended to convert the Request instance to a Response one, and provides several classes to achieve this. The only one we will use, for the moment, is the HttpKernelInterface interface. This interface defines only one method: handle.

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

class Core implements HttpKernelInterface {
  ...
  public function handle(Request $request, ...) {
    ...
    // let matching
    $attributes = $matcher->match($request->getPathInfo());
    $controller = $attributes['controller']; // function
    // call function
    $response = call_user_func_array($controller, $attributes);
    return $response;
 }
```

All together scene

---routing---

Adding routing system in Core class

```
// lib/Framework/Core.php
<?php
namespace Framework;
use Symfony\Component\Routing\RouteCollection;

class Core implements HttpKernelInterface {
  …
  protected $routes = array();

  public function __construct() {
    $this->routes = new RouteCollection();
  }

  // Associates an URL with a callback function
  public function map(string $path, callable $controller) {
    $this->routes->add($path, new Route( $path, array('controller' => $controller)));
  }
  …
}
```

Now application routes can be set in the front controller:

```
// index.php
<?php
...
```

```
$app = new Core();

$app->map( '/',  function () { return new Response('This is the home page'); } );
$app->map( '/about',  function () { return new Response('This is the about page'); } );
$app->map( '/hello/{name}',  function ($name) { return new Response('Hello '.$name); } );
```

Note that this is very similar to what the Symfony full-stack framework is doing: we inject URL parameters into the right controller.

Using the Routing component allows us to load Route objects into a UrlMatcher that will map the requested URI to a matching route. This Route object can contain any attributes that can help us execute the right part of the application. In our case, such an object will contain the PHP callback to execute if the route matches. Also, any dynamic parameters contained in the URL will be present in the route attributes.

We need to do the following changes

Create a UrlMatcher instance and tell it how to match its routes against the requested URI by providing a context to it, using a RequestContext instance.

```
// lib/Framework/Core.php
<?php
namespace Framework;
…
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\RequestContext;

class Core implements HttpKernelInterface {
  …
  public function handle(Request $request, …) {

    // create a context using the current request
    $context = new RequestContext();
    $context->fromRequest($request);
    $matcher = new UrlMatcher($this->routes, $context);
```

The match method tries to match the URL against a known route pattern, and returns the corresponding route attributes in case of success. Otherwise it throws a ResourceNotFoundException that we can catch to display a 404 page

```
try {
    $attributes = $matcher->match($request->getPathInfo());
    …
} catch (ResourceNotFoundException $e) {
    $response = new Response('Not found!', Response::HTTP_NOT_FOUND);
}
```

We can now take advantage of the Routing component to retrieve any URL parameters. After getting rid of the controller attribute, we can call our callback function by passing other parameters as arguments (using the call_user_func_array function):

```
try {
    …
    $controller = $attributes['controller'];
    $response = call_user_func_array($controller, array_values($attributes)) ;
}
```

---dispatcher---

We can implement this in our framework by adding a property dispatcher that will hold an EventDispatcher instance, and an on method, to bind an event to a PHP callback. We'll use the dispatcher to register the callback, and to fire the event later in the framework.

```
// lib/Framework/Core.php
<?php
namespace Framework;
…
use Symfony\Component\EventDispatcher\EventDispatcher;
use Framework\Event\RequestEvent;

class Core implements HttpKernelInterface {
   …
   protected $dispatcher;

   public function __construct() {
      $this->dispatcher = new EventDispatcher();
   }
   // register a listener (callback function) to event
   public function on(string $event, callable $callback) {
      $this->dispatcher->addListener($event, $callback);
   }
```

Add listener to event (function) to Dispatcher

```
// index.php
…
$app = new Core();
...

$app->on('request', function (RequestEvent $event) {
   // let's assume a proper check here
   if ('/admin' == $event->getRequest()->getPathInfo()) {
      echo 'Access Denied!';
      exit;
   }
});
```

Every event inherits from the generic Event class, and is used to hold any information related to it.
Let's write a RequestEvent class, which will be immediately fired when a request is handled by the framework. Of course, this event must have access to the current request, using an attribute holding a Request instance.

```
class RequestEvent extends Event {…}
```

We can now update the code in the handle method to fire a RequestEvent event to the dispatcher every time a request is received.

```
public function handle(Request $request, $type = HttpKernelInterface::MASTER_REQUEST, …) {

    $event = new RequestEvent();
    $event->setRequest($request);
    $this->dispatcher->dispatch('request', $event);
    …
```

---kernel---
// Note : The front controller == index.php file

Let's create the class Core of our framework that implements the HttpKernelInterface. Now create the Core.php file under the lib/Framework directory:

```
// lib/Framework/Core.php
<?php
namespace Framework;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\HttpKernelInterface;

class Core implements HttpKernelInterface {
   public function handle(Request $request, $type = HttpKernelInterface::MASTER_REQUEST, ...) {

      $event = new RequestEvent();
      $event->setRequest($request);
      // trigge handler for event 'request'
      $this->dispatcher->dispatch('request', $event);

      // create a context of the current request
      $context = new RequestContext();
      $context->fromRequest($request);

      // set matcher
      $matcher = new UrlMatcher($this->routes, $context);

      try {
         // let matching
         $attributes = $matcher->match($request->getPathInfo());

         $controller = $attributes['controller'];
         unset($attributes['controller']);
         // call handler function with parameters
         $response = call_user_func_array($controller, array_values($attributes)) ;
      } catch (ResourceNotFoundException $e) {
         $response = new Response('Not found!', Response::HTTP_NOT_FOUND);
      }
      return $response;

   }
}
```

The only thing we did here is move the existing code into the handle method. Now we can get rid of this code in index.php and use our freshly created class instead:

```
// index.php
<?php
$loader = require 'vendor/autoload.php';
use Framework\Core;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();
$app = new Core();
$response = $app->handle($request);
$response->send();
```