The Separation of Concerns

One down-side of our framework right now is that we need to copy and paste the code in `front.php` each time we create a new website. 40 lines of code is not that much, but it would be nice if we could wrap this code into a proper class. It would bring us better *reusability* and easier testing to name just a few benefits.

If you have a closer look at the code, `front.php` has one input, the Request, and one output, the Response. Our framework class will follow this simple principle: the logic is about creating the Response associated with a Request.

Let's create our very own namespace for our framework: `Simplex`. Move the request handling logic into its own `Simplex\\Framework` class::

```
<?php

// example.com/src/Simplex/Framework.php

namespace Simplex;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Matcher\UrlMatcher;
use Symfony\Component\Routing\Exception\ResourceNotFoundException;
use Symfony\Component\HttpKernel\Controller\ControllerResolver;

class Framework
{
    protected $matcher;
    protected $resolver;

    public function __construct(UrlMatcher $matcher, ControllerResolver $resolver)
    {
        $this->matcher = $matcher;
        $this->resolver = $resolver;
    }

    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);

            return call_user_func_array($controller, $arguments);
        } catch (ResourceNotFoundException $e) {
            return new Response('Not Found', 404);
        } catch (\Exception $e) {
            return new Response('An error occurred', 500);
        }
    }
}
```

And update example.com/web/front.php accordingly::

```
<?php

// example.com/web/front.php

// ...

$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();

$framework = new Simplex\Framework($matcher, $resolver);
$response = $framework->handle($request);

$response->send();
```

To wrap up the refactoring, let's move everything but routes definition fromexample.com/src/app.php into yet another namespace: Calendar.

For the classes defined under theSimplex and Calendar namespaces to be autoloaded, update the composer.json file:

.. code-block:: javascript

```
{
    "require": {
        "symfony/http-foundation": "2.5.*",
        "symfony/routing": "2.5.*",
        "symfony/http-kernel": "2.5.*"
    },
    "autoload": {
        "psr-0": { "Simplex\\": "src/", "Calendar\\": "src/" }
    }
}
```

.. note::

For the Composer autoloader to be updated, run ``php composer.phar update``.

Move the controller to Calendar\\Controller\\LeapYearController::

```
<?php

// example.com/src/Calendar/Controller/LeapYearController.php

namespace Calendar\Controller;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Calendar\Model\LeapYear;

class LeapYearController
{
    public function indexAction(Request $request, $year)
```

```
    {
        $leapyear = new LeapYear();
        if ($leapyear->isLeapYear($year)) {
            return new Response('Yep, this is a leap year!');
        }

        return new Response('Nope, this is not a leap year.');
    }
}
```

And move the `is_leap_year()` function to its own class too::

```
<?php

// example.com/src/Calendar/Model/LeapYear.php

namespace Calendar\Model;

class LeapYear
{
    public function isLeapYear($year = null)
    {
        if (null === $year) {
            $year = date('Y');
        }

        return 0 == $year % 400 || (0 == $year % 4 && 0 != $year % 100);
    }
}
```

Don't forget to update the `example.com/src/app.php` file accordingly::

```
$routes->add('leap_year', new Routing\Route('/is_leap_year/{year}', array(
    'year' => null,
    '_controller' => 'Calendar\\Controller\\LeapYearController::indexAction',
)));
```

To sum up, here is the new file layout:

.. code-block:: text

```
example.com
â”œâ”€â”€ composer.json
â”‚   src
â”‚   â”œâ”€â”€ app.php
â”‚   â”‚â”€â”€ Simplex
â”‚   â”‚       â”‚â”€â”€ Framework.php
â”‚   â”‚â”€â”€ Calendar
â”‚   â”‚       â”‚â”€â”€ Controller
â”‚   â”‚       â”‚   â”‚â”€â”€ LeapYearController.php
â”‚   â”‚       â”‚â”€â”€ Model
â”‚   â”‚           â”‚â”€â”€ LeapYear.php
â”œâ”€â”€ vendor
â”‚â”€â”€ web
    â”‚â”€â”€ front.php
```

That's it! Our application has now four different layers and each of them has a well defined goal:

- `web/front.php`: The front controller; the only exposed PHP code that makes the interface with

the client (it gets the Request and sends the Response) and provides the boiler-plate code to initialize the framework and our application;

- src/Simplex: The reusable framework code that abstracts the handling of incoming Requests (by the way, it makes your controllers/templates easily testable -- more about that later on);
- src/Calendar: Our application specific code (the controllers and the model);
- src/app.php: The application configuration/framework customization.