Table of Contents

You are browsing the **Symfony 4 documentation**, which changes significantly from Symfony 3.x. If your app doesn't use Symfony 4 yet, browse the Symfony 3.4 documentation.

Controller

4.2 version edit this page

A controller is a PHP function you create that reads information from the `Request` object and creates and returns a `Response` object. The response could be an HTML page, JSON, XML, a file download, a redirect, a 404 error or anything else. The controller executes whatever arbitrary logic *your application* needs to render the content of a page.

Tip

If you haven't already created your first working page, check out *Create your First Page in Symfony* and then come back!

A Simple Controller ¶

While a controller can be any PHP callable (a function, method on an object, or a `Closure`), a controller is usually a method inside a controller class:

```
1  // src/Controller/LuckyController.php
2  namespace App\Controller;
3
4  use Symfony\Component\HttpFoundation\Response;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class LuckyController
8  {
9      /**
10      * @Route("/lucky/number/{max}", name="app_lucky_number")
11      */
12     public function number($max)
13     {
14         $number = random_int(0, $max);
15
16         return new Response(
17             '<html><body>Lucky number: '.$number.'</body></html>'
18         );
19     }
20 }
```

The controller is the `number()` method, which lives inside a controller class `LuckyController`.

This controller is pretty straightforward:

- *line 2*: Symfony takes advantage of PHP's namespace functionality to namespace the entire controller class.
- *line 4*: Symfony again takes advantage of PHP's namespace functionality: the `use` keyword imports the `Response` class, which the controller must return.
- *line 7*: The class can technically be called anything, but it's suffixed with `Controller` by convention.
- *line 12*: The action method is allowed to have a `$max` argument thanks to the `{max}` wildcard in the route.
- *line 16*: The controller creates and returns a `Response` object.

Mapping a URL to a Controller ¶

In order to *view* the result of this controller, you need to map a URL to it via a route. This was done above with the `@Route("/lucky/number/{max}")` route annotation.

To see your page, go to this URL in your browser:

> *http://localhost:8000/lucky/number/100*

For more information on routing, see *Routing*.

The Base Controller Class & Services ¶

To make life nicer, Symfony comes with an optional base controller class called `AbstractController`. You can extend it to get access to some helper methods.

Add the `use` statement atop your controller class and then modify `LuckyController` to extend it:

```
// src/Controller/LuckyController.php
namespace App\Controller;

+ use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

- class LuckyController
+ class LuckyController extends AbstractController
{
    // ...
}
```

That's it! You now have access to methods like $this->render() and many others that you'll learn about next.

Generating URLs

The `generateUrl()` method is just a helper method that generates the URL for a given route:

```
$url = $this->generateUrl('app_lucky_number', ['max' => 10]);
```

Redirecting

If you want to redirect the user to another page, use the `redirectToRoute()` and `redirect()` methods:

```
use Symfony\Component\HttpFoundation\RedirectResponse;

// ...
public function index()
{
    // redirects to the "homepage" route
    return $this->redirectToRoute('homepage');

    // redirectToRoute is a shortcut for:
    // return new RedirectResponse($this->generateUrl('homepage'));

    // does a permanent - 301 redirect
    return $this->redirectToRoute('homepage', [], 301);

    // redirect to a route with parameters
    return $this->redirectToRoute('app_lucky_number', ['max' => 10]);

    // redirects to a route and maintains the original query string parameters
    return $this->redirectToRoute('blog_show', $request->query->all());

    // redirects externally
    return $this->redirect('http://symfony.com/doc');
}
```

Caution

The `redirect()` method does not check its destination in any way. If you redirect to a URL provided by end-users, your application may be open to the unvalidated redirects security vulnerability.

Rendering Templates ¶

If you're serving HTML, you'll want to render a template. The `render()` method renders a template **and** puts that content into a `Response` object for you:

```
// renders templates/lucky/number.html.twig
return $this->render('lucky/number.html.twig', ['number' => $number]);
```

Templating and Twig are explained more in the *Creating and Using Templates article*.

Fetching Services ¶

Symfony comes *packed* with a lot of useful objects, called *services*. These are used for rendering templates, sending emails, querying the database and any other "work" you can think of.

If you need a service in a controller, type-hint an argument with its class (or interface) name. Symfony will automatically pass you the service you need:

```
use Psr\Log\LoggerInterface;
// ...

/**
 * @Route("/lucky/number/{max}")
 */
public function number($max, LoggerInterface $logger)
{
    $logger->info('We are logging!');
    // ...
}
```

Awesome!

What other services can you type-hint? To see them, use the `debug:autowiring` console command:

```
$ php bin/console debug:autowiring
```

If you need control over the *exact* value of an argument, you can bind the argument by its name:

YAML XML PHP

```
# config/services.yaml
services:
    # ...

```

```
    # explicitly configure the service
    App\Controller\LuckyController:
        public: true
        bind:
            # for any $logger argument, pass this specific service
            $logger: '@monolog.logger.doctrine'
            # for any $projectDir argument, pass this parameter value
            $projectDir: '%kernel.project_dir%'
```

Like with all services, you can also use regular constructor injection in your controllers.

For more information about services, see the *Service Container* article.

Generating Controllers ¶

To save time, you can install Symfony Maker and tell Symfony to generate a new controller class:

```
$ php bin/console make:controller BrandNewController

created: src/Controller/BrandNewController.php
```

If you want to generate an entire CRUD from a Doctrine *entity*, use:

```
$ php bin/console make:crud Product
```

New in version 1.2: The `make:crud` command was introduced in MakerBundle 1.2.

Managing Errors and 404 Pages ¶

When things are not found, you should return a 404 response. To do this, throw a special type of exception:

```
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

// ...
public function index()
{
    // retrieve the object from database
    $product = ...;
    if (!$product) {
        throw $this->createNotFoundException('The product does not exist');

        // the above is just a shortcut for:
        // throw new NotFoundHttpException('The product does not exist');
    }

```

```
    return $this->render(...);
}
```

The `createNotFoundException()` method is just a shortcut to create a special `NotFoundHttpException` object, which ultimately triggers a 404 HTTP response inside Symfony.

If you throw an exception that extends or is an instance of `HttpException`, Symfony will use the appropriate HTTP status code. Otherwise, the response will have a 500 HTTP status code:

```
// this exception ultimately generates a 500 status error
throw new \Exception('Something went wrong!');
```

In every case, an error page is shown to the end user and a full debug error page is shown to the developer (i.e. when you're in "Debug" mode - see The parameters Key: Parameters (Variables)).

To customize the error page that's shown to the user, see the *How to Customize Error Pages* article.

The Request object as a Controller Argument ¶

What if you need to read query parameters, grab a request header or get access to an uploaded file? All of that information is stored in Symfony's `Request` object. To get it in your controller, add it as an argument and **type-hint it with the Request class**:

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

public function index(Request $request, $firstName, $lastName)
{
    $page = $request->query->get('page', 1);

    // ...
}
```

Keep reading for more information about using the Request object.

Managing the Session ¶

Symfony provides a session service that you can use to store information about the user between requests. Session is enabled by default, but will only be started if you read or write from it.

Session storage and other configuration can be controlled under the framework.session configuration in `config/packages/framework.yaml`.

To get the session, add an argument and type-hint it with `SessionInterface`:

```
use Symfony\Component\HttpFoundation\Session\SessionInterface;

public function index(SessionInterface $session)
{
    // stores an attribute for reuse during a later user request
    $session->set('foo', 'bar');
```

```

    // gets the attribute set by another controller in another request
    $foobar = $session->get('foobar');

    // uses a default value if the attribute doesn't exist
    $filters = $session->get('filters', []);
}
```

Stored attributes remain in the session for the remainder of that user's session.

For more info, see *Sessions*.

Flash Messages ¶

You can also store special messages, called "flash" messages, on the user's session. By design, flash messages are meant to be used exactly once: they vanish from the session automatically as soon as you retrieve them. This feature makes "flash" messages particularly great for storing user notifications.

For example, imagine you're processing a *form* submission:

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

public function update(Request $request)
{
    // ...

    if ($form->isSubmitted() && $form->isValid()) {
        // do some sort of processing

        $this->addFlash(
            'notice',
            'Your changes were saved!'
        );
        // $this->addFlash() is equivalent to $request->getSession()->getFlashBag()->add()

        return $this->redirectToRoute(...);
    }

    return $this->render(...);
}
```

After processing the request, the controller sets a flash message in the session and then redirects. The message key (`notice` in this example) can be anything: you'll use this key to retrieve the message.

In the template of the next page (or even better, in your base layout template), read any flash messages from the session using `app.flashes()`:

```
{# templates/base.html.twig #}

{# read and display just one flash message type #}
{% for message in app.flashes('notice') %}
```

```
    <div class="flash-notice">
        {{ message }}
    </div>
{% endfor %}

{# read and display several types of flash messages #}
{% for label, messages in app.flashes(['success', 'warning']) %}
    {% for message in messages %}
        <div class="flash-{{ label }}">
            {{ message }}
        </div>
    {% endfor %}
{% endfor %}

{# read and display all flash messages #}
{% for label, messages in app.flashes %}
    {% for message in messages %}
        <div class="flash-{{ label }}">
            {{ message }}
        </div>
    {% endfor %}
{% endfor %}
```

It's common to use `notice`, `warning` and `error` as the keys of the different types of flash messages, but you can use any key that fits your needs.

> **Tip**
>
> You can use the `peek()` method instead to retrieve the message while keeping it in the bag.

The Request and Response Object ¶

As mentioned earlier, Symfony will pass the `Request` object to any controller argument that is type-hinted with the `Request` class:

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

public function index(Request $request)
{
    $request->isXmlHttpRequest(); // is it an Ajax request?

    $request->getPreferredLanguage(['en', 'fr']);

    // retrieves GET and POST variables respectively
    $request->query->get('page');
    $request->request->get('page');

    // retrieves SERVER variables
    $request->server->get('HTTP_HOST');

```

```
    // retrieves an instance of UploadedFile identified by foo
    $request->files->get('foo');

    // retrieves a COOKIE value
    $request->cookies->get('PHPSESSID');

    // retrieves an HTTP request header, with normalized, lowercase keys
    $request->headers->get('host');
    $request->headers->get('content-type');
}
```

The `Request` class has several public properties and methods that return any information you need about the request.

Like the `Request`, the `Response` object has also a public `headers` property. This is a `ResponseHeaderBag` that has some nice methods for getting and setting response headers. The header names are normalized so that using `Content-Type` is equivalent to `content-type` or even `content_type`.

The only requirement for a controller is to return a `Response` object:

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

// creates a simple Response with a 200 status code (the default)
$response = new Response('Hello '.$name, Response::HTTP_OK);

// creates a CSS-response with a 200 status code
$response = new Response('<style> ... </style>');
$response->headers->set('Content-Type', 'text/css');
```

There are special classes that make certain kinds of responses easier. Some of these are mentioned below. To learn more about the `Request` and `Response` (and special `Response` classes), see the HttpFoundation component documentation.

Returning JSON Response ¶

To return JSON from a controller, use the `json()` helper method. This returns a special `JsonResponse` object that encodes the data automatically:

```
// ...
public function index()
{
    // returns '{"username":"jane.doe"}' and sets the proper Content-Type header
    return $this->json(['username' => 'jane.doe']);

    // the shortcut defines three optional arguments
    // return $this->json($data, $status = 200, $headers = [], $context = []);
}
```

If the *serializer service* is enabled in your application, it will be used to serialize the data to JSON. Otherwise, the `json_encode` function is used.

Streaming File Responses ¶

You can use the `file()` helper to serve a file from inside a controller:

```
public function download()
{
    // send the file contents and force the browser to download it
    return $this->file('/path/to/some_file.pdf');
}
```

The `file()` helper provides some arguments to configure its behavior:

```
use Symfony\Component\HttpFoundation\File\File;
use Symfony\Component\HttpFoundation\ResponseHeaderBag;

public function download()
{
    // load the file from the filesystem
    $file = new File('/path/to/some_file.pdf');

    return $this->file($file);

    // rename the downloaded file
    return $this->file($file, 'custom_name.pdf');

    // display the file contents in the browser instead of downloading it
    return $this->file('invoice_3241.pdf', 'my_invoice.pdf', ResponseHeaderBag::DISPOSITION_
}
```

Final Thoughts ¶

Whenever you create a page, you'll ultimately need to write some code that contains the logic for that page. In Symfony, this is called a controller, and it's a PHP function where you can do anything in order to return the final `Response` object that will be returned to the user.

To make life easier, you'll probably extend the base `AbstractController` class because this gives access to shortcut methods (like `render()` and `redirectToRoute()`).

In other articles, you'll learn how to use specific services from inside your controller that will help you persist and fetch objects from a database, process form submissions, handle caching and more.

Keep Going! ¶

Next, learn all about *rendering templates with Twig*.

Learn more about Controllers ¶

- Extending Action Argument Resolving
- How to Customize Error Pages

- How to Forward Requests to another Controller
- How to Define Controllers as Services
- How to Create a SOAP Web Service in a Symfony Controller
- How to Upload Files

Latest from the Symfony Blog

Symfony 4.2.4 released
March 3, 2019

Symfony 3.4.23 released
March 3, 2019

They Help Us Make Symfony

Thanks **Jordan Hoff** for being a Symfony contributor.
1 commit · **2** lines

Get Involved in the Community

A passionate group of over 600,000 developers from more than 120 countries, all committed to helping PHP surpass the impossible.

Getting involved →