Symfony 5 Deep Dive! The HttpKernel Request-Response Flow

With <3 from SymfonyCasts

Chapter 1: Events, Events & Events!

Hi friends! Ok: so you already know how to use Symfony... maybe you... use it every day. Heck, I love it so much, I've been known to use it on vacation! And now, you're ready to go deeper - to find out how Symfony *really* works under-the-hood. If this is you, welcome! We're in for a wild ride.

In this first deep dive tutorial, we're going to the *heart* of what happens during the request-response process in Symfony. It all centers around a class called `HttpKernel`, which is an *incredible* class. This *one* class is used as the *heart* of Symfony *and* Drupal... as well as a bunch of other projects, for example, phpBB - the famous forum system.

So how can one class be the *heart* of technologies that are seemingly *so* different? That's what we're going to find out.

Project Setup

As always, if you *truly* want to impress your friends with your *deep* knowledge of Symfony, download the course code and code along with me. After you unzip the file, you'll find a `start/` directory with the same code that you see here. Follow the `README.md` file for all the *thrilling* setup instructions.

The *last* step will be to leverage the Symfony binary to start a web server with `symfony serve`. I'm actually going to pass `-d` so it runs in the background as a daemon:

```
$ symfony serve -d
```

Now, *spin* back over to your browser and head to https://localhost:8000 to find: The SpaceBar. Some of you might recognize this from our Symfony 4 tutorials. Well, I've upgraded it to Symfony 5 and it will be our *perfect* guinea pig for diving deep into Symfony.

Request -> Controller -> Response. But what else?

Ok: we know that *everything* starts with a request: a request comes into our server, it's handled by our application, yadda, yadda, yadda, a response comes out... and profit! The goal of this tutorial is simple: find out what *really* happens in between.

For the homepage, let's find its controller: `src/Controller/ArticleController.php`. Here it is: `homepage()`, with the route above it.

The two things that we *know* happen between the start of the request and the end of the response, are that the route is matched and then *something* calls our controller... probably Fabien personally calls it... I don't know. And then our controller always, well *usually*, returns a response. That's what `$this->render()` returns.

What I want to know is: *who* executes the routing and *who* ultimately calls my controller? I want to see the code that does that!

Holder of Secrets: The Profiler Performance Tab

To start this journey, go back to your browser and, on the web debug toolbar on the bottom, right click on the milliseconds link and open it in a new tab to jump into the "Performance" section of the profiler.

This screen is *awesome*. It's *meant* to show you where your site might be slow, but its *real* superpower is that it can show you *everything* that's happening inside of Symfony. The trick is to change this "threshold" input box from 1 milliseconds down to 0... so that it doesn't hide anything.

Simply gorgeous. This is the request-response process. You can see - kind of in the middle here - is our controller: it took 36 milliseconds to execute. You can see the Twig templates being executed below it, and even little Doctrine queries happening along the way.

The biggest thing I want you to notice is that most of the other lines - both before and after the controller - contain the word `Listener`, or sometimes `Subscriber`, which is basically another word for "listener".

Because, at a high level, here's what happens inside Symfony: it boots, triggers some events, executes your controller, then dispatches some other events.

To get an even *better* view of these events, click... the Events tab! This shows all the events that were dispatched during this request. So, apparently there's an event called `kernel.request`: that was the *first* event dispatched. And here are all of the listeners - so all the "functions" - that were called when that event was triggered.

Then there's another event called `kernel.controller`... and many more. You can even see listeners for events that were *not* triggered during this request.

So... let's start messing with stuff! Next, let's create our *own* event listener and execute code *before* our controller is called.

Chapter 2: Hooking into Symfony with an Event Subscriber

Before we dive into the core code, let's hook *into* the request-response process. Let's create our own *listener* to this `kernel.request` event. To do that, in the `src/` directory, I already have an `EventListener/` directory. It doesn't matter *where* we put this class, but inside here, let's create a new class called `UserAgentSubscriber`.

All event subscribers must implement `EventSubscriberInterface`. I'll go to the Code -> Generate menu on PhpStorm - or Command + N on a Mac - and select "Implement Methods" to generate the one method this interface requires: `getSubscribedEvents()`. Inside, return an array of *all* the events we want to listen to, which will just be one.

14 lines | src/EventListener/UserAgentSubscriber.php

```php
... lines 1 - 4
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class UserAgentSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {

    }
}
```

Now... you *might* be expecting me to say `'kernel.request' => 'onKernelRequest'`. This would mean that when the `kernel.request` event happens, I want Symfony to call an `onKernelRequest()` method on this class that we will create in a minute. This *would* work, but starting in Symfony 4.3, instead of using this made-up `kernel.request` string, you can pass the event *class* name, which in this case is `RequestEvent::class`.

22 lines | src/EventListener/UserAgentSubscriber.php

```php
... lines 1 - 5
use Symfony\Component\HttpKernel\Event\RequestEvent;
... line 7
class UserAgentSubscriber implements EventSubscriberInterface
{
... lines 10 - 14
    public static function getSubscribedEvents()
    {
        return [
            RequestEvent::class => 'onKernelRequest'
        ];
    }
}
```

More and more, you'll see documentation that tells you to listen to an event *class* like this, instead of a random string.

Now, create the function: `public function onKernelRequest()`. Inside, dump and die `it's alive!!!`.

22 lines | src/EventListener/UserAgentSubscriber.php

```php
... lines 1 - 9
    public function onKernelRequest()
    {
        dd('it\'s alive!!!');
    }
... lines 14 - 22
```

Cool! With any luck, Symfony will call our event listener *very* early on and it will kill the page. Close the profiler, refresh and... it's alive! Well actually, it's *dead*, but ya know... that's what we wanted!

Logging in the Listener and Controller

To make the class more interesting, let's log something! You know the drill: add `public function __construct()` with `LoggerInterface $logger`. I'll hit Alt+Enter and go to initialize fields as a lazy way to create the property and set it down here.

30 lines | src/EventListener/UserAgentSubscriber.php

```php
... lines 1 - 4
use Psr\Log\LoggerInterface;
... lines 6 - 8
class UserAgentSubscriber implements EventSubscriberInterface
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }
... lines 17 - 28
}
```

In the method, add `$this->logger->info()` with:

> I'm logging SUPER early on the request!

30 lines | src/EventListener/UserAgentSubscriber.php

```php
... lines 1 - 17
    public function onKernelRequest()
    {
        $this->logger->info('I\'m logging SUPER early on the request!');
    }
... lines 22 - 30
```

To compare this to logging in a controller, go back to `ArticleController`. On the `homepage` action, autowire a `$logger` argument and say `$logger->info()`:

> Inside the controller!

66 lines | src/Controller/ArticleController.php

```php
... lines 1 - 8
use Psr\Log\LoggerInterface;
... lines 10 - 13
class ArticleController extends AbstractController
{
... lines 16 - 28
    public function homepage(ArticleRepository $repository, LoggerInterface $logger)
    {
        $logger->info('Inside the controller!');
... lines 32 - 36
    }
... lines 38 - 64
}
```

We *expect* that the listener will be called first because the RequestEvent, also known as `kernel.request`, happens *before* the controller is executed. Refresh the page. It works... and once again, open the profiler in a new tab, click Logs and... perfect! First our listener log and *then* the controller.

And you can *now* see our subscriber inside the performance section! Make sure you have the threshold set to 0. Let's see... there it is: `UserAgentSubscriber`. And then down... *way* after that... is the controller.

The Event Argument

One of the other "laws" of Symfony's event system is that a listener will *always* be passed a single argument: an *event* object. What *type* of object is it? This is where the new "event class names as event names" comes in handy. We're listening to `RequestEvent`, which means - surprise! - Symfony will pass us a `RequestEvent` object! Let's just `dd($event)`.

31 lines | src/EventListener/UserAgentSubscriber.php

```php
// ... lines 1 - 6
use Symfony\Component\HttpKernel\Event\RequestEvent;
// ... line 8
class UserAgentSubscriber implements EventSubscriberInterface
{
// ... lines 11 - 17
    public function onKernelRequest(RequestEvent $event)
    {
        dd($event);
// ... line 21
    }
// ... lines 23 - 29
}
```

Ok, move back over, close the profiler again, refresh and... there it is! Each event you listen to will be passed a *different* event object... and each event object will have different super-powers: giving you whatever information you might need for that particular situation, and *often*, allowing you to *change* things.

For example, this event contains the `Request` object... because if you're listening to this *very* early event in Symfony... there's a good chance that you might want to use the `Request` object to do something.

In fact, let's do exactly that. Clear out our method and say `$request = $event->getRequest()`. And then we'll grab the `$userAgent` off of the request with `$request->headers->get('User-Agent')`. *Finally*, let's log this: `$this->logger->info()` and I'll use `sprintf()` to say

> The User-Agent is %s

Pass `$userAgent` for the placeholder.

33 lines | src/EventListener/UserAgentSubscriber.php

```php
// ... lines 1 - 17
    public function onKernelRequest(RequestEvent $event)
    {
        $request = $event->getRequest();

        $userAgent = $request->headers->get('User-Agent');
        $this->logger->info(sprintf('The User-Agent is "%s"', $userAgent));
    }
// ... lines 25 - 33
```

Let's check it out! Move over, refresh, open the profiler in a new tab, go down to Logs and... we got it! We're logging the user agent *before* the controller is called.

Ok! Now that we've hooked into Symfony, let's take a step back and start tracing through *everything* that happens from the start of the request, line-by-line. We'll even see *where* the `RequestEvent` is dispatched and eventually where the controller is executed.

Let's start that journey next.

Chapter 3: index.php to HttpKernel::handle()

Let's start from the *very* beginning of the request. When we load a page, the *first* file that's executed is `public/index.php`. No matter what, this is where it all starts. So let's literally go through this file line-by-line and see what happens.

> If you start a new project in Symfony 5.3 or later, this file will look quite different thanks to the new `symfony/runtime` component. But, the same things are still happening behind the scenes.

index.php Bootstrapping

The first thing it does is require this `config/bootstrap.php` file. For our purposes, this... isn't important. It requires the Composer autoloader... and then the rest of this file is all about loading and normalizing environment variables. Sure, environment variables *are* important to Symfony, but if you want to understand the request-response flow, not so much.

Next, if we're in debug mode, it calls `Debug::enable()`. That's great to set up some debugging tools... but not relevant to us.

Hello Kernel

The first thing *we* care about is down here: `$kernel = new Kernel()`. This is actually instantiating *our* `src/Kernel.php` class, which is the *heart* of our application.

The `Kernel` is passed the environment as the first argument and a debug flag as the second. That controls a *bunch* of behavior... but isn't very important to the request-response flow.

But the next line *is* important. We always knew that there was a `Request` object inside Symfony. If you ever wondered *who* creates the `Request` and *where*, here's your answer: it's created in *our* code - not somewhere deep in the core.

The `::createFromGlobals()` method - I'll hold command or control to open that method inside Symfony - is a shortcut to create the `Request` object and populate its data with the normal superglobal variables, like `$_SERVER` and `$_POST`. This gives us a nice `Request` object that represents the current request info.

HttpKernel::handle(): Our App in One Method

The next line... oh... the next line. This is probably my *favorite* line of code in *all* of PHP: `$response = $kernel->handle($request)`. *That* runs our app. We don't know exactly what happens *inside* that method - that's what we're going to figure out - but isn't it beautiful? Our application & Symfony are *not* some weird, global monster that takes over our PHP process and eats our objects. Nope, it's a *pure* function. Input `$request`, output `$response`... which is *exactly* what our job as a developer is! Understand the incoming request, and use that to create a response.

One of the properties of a "pure" function like this is that you can call it as many times as you want. So yes, in theory, a single `Kernel` can handle *multiple* requests inside just *one* PHP process. In fact, let's do that!

Up above, let's say `$request1 = Request::create()` - which is another shortcut to create a `Request` object. Let's make this look like a Request for our login page. Pass `/login` as the first arg.

Now create a `$request2` variable and pretend that this is a request for `/register`.

39 lines | public/index.php

```
... lines 1 - 24
$request1 = Request::create('/login');
$request2 = Request::create('/register');
... lines 27 - 39
```

Could we run our kernel and get 2 responses for these 2 requests? Uh... totally! `$response1 = $kernel->handle($request1)` ... and then `$response2 = $kernel->handle($request2)`. Let's see what they look like: `dump($response1)`, `dump($response2)` and then `die`.

39 lines | public/index.php

```
... lines 1 - 27
$response1 = $kernel->handle($request1);
$response2 = $kernel->handle($request2);

dump($response1);
dump($response2);
die;
... lines 34 - 39
```

Let's do this! Move over, refresh and... check it out! We just handled *two* different requests on the same page! The first *does* contain the HTML for the login page, and the second... for the registration page. Amazing.

And this idea of handling multiple requests in Symfony is something that really *does* happen! It happens with sub-requests - a topic that we will cover later in this tutorial - and some people use an event loop in PHP to boot a single kernel and then handle *many*, *real*, HTTP requests.

Ok, remove all of this code. It's now obvious that if we *really* want to understand what happens inside Symfony, we need to find out what happens inside of this `$kernel->handle()` method. We're going be opening a *lot* of core files, so make sure you have an easy way to "jump to a file" by typing a filename in your editor. In PhpStorm, I can hit Shift+Shift to open a file called `HttpKernel.php`, which lives *deep* inside Symfony. If you don't see it, make sure the "Include non-project items" checkbox is checked - PhpStorm usually does that automatically if you type a specific filename.

Once inside... scroll down to the `handle()` method.

Hello HttpKernel::handle()

Ok, *technically* the `$kernel->handle()` method we saw in `index.php` is *not* the `handle()` method in this class. Symfony *first* initializes the dependency injection container - the topic of a *future* deep-dive tutorial - and *then* calls this method.

The *first* thing I want you to notice is that the *entire* function is surrounded by a try-catch block. So almost *immediately* when our app starts running, our code is surrounded by a try catch! That's not important *yet*. But later, we'll see what happens when an exception is thrown from *anywhere*.

The *real* logic of `HttpKernel` lives in this `handleRaw()` method. Scroll down a little to find it. Ah yes: `handleRaw()`. *This* is the Symfony framework. These 50 lines of code are the heart of *everything* that happens in Symfony! And not just Symfony: these *same* 50 lines of code run Drupal, phpBB and many other things!

So next: let's start our journey through this strange and wondrous method.

Chapter 4: RequestEvent & RouterListener

We've traced the code from the first file that's executed - `public/index.php` - all the way into this core `HttpKernel` class. Specifically, this `handleRaw()` method. These 50 lines of code are the *entire* framework. It somehow transforms a `Request` at the beginning into a `Response` at the end. The question is: how? What does it *do* to accomplish that?

The RequestStack

The first line uses something called a `RequestStack`: `$this->requestStack->push($request)`.

The `RequestStack` is a small object that basically holds an array of `Request` objects. It's not important now, but we *will* talk about it later. Because, yes, as *strange* as it may seem, there *is* a concept inside Symfony of handling *multiple* `Request` objects at the same time. But don't worry about it now.

Dispatching RequestEvent (kernel.request)

The first really important thing is that Symfony dispatches an event. So, almost before doing *anything* else, the first event is triggered. It's called `KernelEvents::REQUEST`, which is a constant that *really* means the string: `kernel.request`. And... surprise! What type of *event* object is passed to these listeners? A `RequestEvent`.

Go back to your browser and refresh the page: it should be working now. Click any of the web debug toolbar links to jump into the profiler... and go to the Performance section.

As we talked about earlier, our controller is somewhere in the middle... and most of the things before and after the controller are *listeners* to different events. In fact, look at this one: a gray bar called `kernel.request`. We just saw where that's dispatched!

The Listeners to RequestEvent

This shows us how *long* it took to execute all of the listeners for this event. To get a better view, go back to the Events section of the profiler. Yep, the *very* first event that was triggered was `kernel.request`. And in this app, it has about 10 different listeners.

For the purposes of understanding how Symfony works, the majority of these listeners aren't very important. They do various things. Like, for example, this `ValidateRequestListener` checks that a few important parts of the request are correctly formatted. Basically, it will cause an error if it looks like a hacker is trying to *manipulate* the request, which is cool!

At the bottom of the listener list, check it out! There is *our* `UserAgentSubscriber`. It's last because it has the lowest listener priority.

Really, out of all these listeners, there is only *one* that is *critically* important to understanding how the framework works. It's `RouterListener`. If you ever wondered *where* the routing system is actually executed - at what *point* it looks at the request and tries to find a matching route - here is your answer: `RouterListener`.

Hello RouterListener

So... let's go see what it does! I'll hit Shift+Shift and type `RouterListener.php`. I'll click the "Include non-project" items box to see it.

Open this up! The first thing I want you to notice is that this looks a *lot* like our event subscriber. It implements `EventSubscriberInterface` and... if we find the `getSubscribedEvents()` method down here, it listens to a *few* events. The only one that's important for us is `KernelEvents::Request`, which, remember, *really* means `kernel.request`.

Find the `onKernelRequest` method... here it is. Skip down a little - for me, down to lines 111 to 115. *This* is where the router is executed: `$this->matcher` is the router. It's not really important which side of the `if` statement is actually executed: either way, this runs the routing.

So... *my* question is: what is the *end result* of executing the "match" functionality on the Router? At a high level, we understand the router: it looks at the current URL - and sometimes other parts of the request - and determines which route matches.

What does Routing Return?

But... what is this $parameters variable? What does the match() method return? Let's find out! Hit enter and dd($parameters).

177 lines | vendor/symfony/http-kernel/EventListener/RouterListener.php

```
... lines 1 - 96
    public function onKernelRequest(RequestEvent $event)
    {
... lines 99 - 108
        try {
... lines 110 - 115
            dd($parameters);
... lines 117 - 129
        } catch (ResourceNotFoundException $e) {
... lines 131 - 141
        }
    }
... lines 144 - 177
```

Let's go! Find your browser and click *back* to get the homepage. I'm also going to open an article page in another tab... and then refresh.

Interesting: what we get back from the router is an *array* with two things inside: _route - that's the name of the matched route - and _controller, which is the full class and method name of the controller that's *attached* to the route.

Ok... what about the article show page? Move to that tab. Woh! We get the *same* thing - _route and _controller - but with one *new* item in the array: a slug key! As a reminder, if you go to src/Controller/ArticleController.php and find the show() method, its *route* has a wildcard called slug! So *really*, what the router returns is this: a combination of *all* the wildcard values in the route *plus* _controller and _route.

That's... *mostly* true. But come on! This is a deep-dive course! So next, let's look even *deeper* at routes and route defaults to learn the *full* story. We'll also look at what RouterListener *does* with this super-important array.

This array is the *end* result of the route-matching process. Apparently, the router returns an array with the wildcard values from the route *plus* keys for the route and controller.

But... it's a bit more interesting than that. A great way to see how, is by playing with a route in YAML. Open up `config/routes.yaml`. Uncomment the example route and change the path to `/playing`. Now, on your browser, open *another* tab and go to https://localhost:8000/playing.

4 lines | config/routes.yaml

```yaml
index:
    path: /playing
... lines 3 - 4
```

That's exactly what we expected: `_route` set to the route name and `_controller` set to the controller string for that route.

Route Defaults

But in reality, the `controller` key in a YAML route is just a shortcut. Before Symfony 4, there *was* no `controller` key. Nope, to define a controller you added a `defaults` key and put an `_controller` key below *that*.

5 lines | config/routes.yaml

```yaml
index:
... line 2
    defaults:
        _controller: App\Controller\DefaultController::index
```

Move over and refresh now. Woh! We get the *exact* same array! Yep, the `controller` key is *really* just a shortcut for setting an `_controller` *default* value on the route.

This is actually an important point, but to see why, let's go a bit further. First, add a `{id}` wildcard to the end of the path. Then, at your browser, add `/5` to the end of the URL. And... yep! The array now has an `id` key: no surprise.

5 lines | config/routes.yaml

```yaml
index:
    path: /playing/{id}
... lines 3 - 5
```

Normally, the purpose of `defaults` on a route are to give a default value for a *wildcard*. If we say `id: 10`... and then refresh, the array still contains 5 because *that's* what's in the URL. But *thanks* to the default, *now* we can *just* go to `/playing` and... the id uses the default value `10`.

6 lines | config/routes.yaml

```yaml
index:
... line 2
    defaults:
... line 4
        id: 10
```

Cool. But what if we just... *invent* a new key and put it here? Like `totally_inventing_this_default` set to `true`.

7 lines | config/routes.yaml

```yaml
index:
... line 2
    defaults:
... lines 4 - 5
        totally_inventing_this_default: true
```

This won't change how the route *matches*, but it *will* change what we get back in the array. Refresh. The `totally_inventing_this_default` key is now inside the returned array!

So here's the *full* story of what the route matching process returns: it returns an `array_merge` of the route defaults and any wildcard values in the route.... plus the `_route` key... just in case that's handy.

With route annotations, it looks a bit different, but it's exactly the same. We can add a `defaults` key and set `foo` to `bar`. Back in the browser, close the last tab and refresh the article show page. We suddenly have a `foo` key! On the route, remove that `defaults` stuff.

66 lines | src/Controller/ArticleController.php

```php
... lines 1 - 13
class ArticleController extends AbstractController
{
... lines 16 - 38
    /**
     * @Route("/news/{slug}", name="article_show", defaults={"foo": "bar"})
     */
    public function show(Article $article, SlackClient $slack)
... lines 43 - 64
}
```

Request Attributes

So why is it *so* important to understand exactly *what* the route-matching process returns? We'll find out the *full* answer soon. But first... back in `RouterListener`, what does this class *do* with the `$parameters` array?

Remove the `dd()`... and let's follow the logic. It does some logging and... here it is: `$request->attributes->add($parameters)`. *This* is important.

Let's back up for a second: the `Request` object has several public properties and *all* of them - except one! - correspond to something on the HTTP request. For example, `$request->headers` holds the HTTP request headers, `$request->cookies` holds the cookies, and there are others like `$request->query` to read the query parameters. The point is: *all* of these refer to real "parts" of an HTTP request. You could talk to a Java developer about HTTP headers and they would know what you're referring to.

The *one* exception is `$request->attributes`. This property does *not* correspond to any *real* part of the HTTP request. If you ask that *same* Java developer:

> Hey! What are the attributes on your request?

They'll think you're nuts. Nope, the Request attributes are something totally invented by Symfony. The *purpose* of the request attributes is to be a place where you can store data about the request that's specific to your application. So, storing the *controller*, for example, is a perfect fit! That's *completely* a Symfony concept.

Anyways, the array of `$parameters` from the router is added to the `$request->attributes()`. What does that... do? Absolutely nothing. Soon, something *else* will *use* this data, but at this moment, this is *just* data sitting on the request.

It also sets another attribute `_route_params`... but that's not really important.

After kernel.request... we have Request Attributes!

Ok! `RouterListener` done! Close that class, high-five your cat - and go back to `HttpKernel`. As we saw, there are a lot of listeners to the `kernel.request` event, but by *far*, the most important one is `RouterListener`. So what *changed* in our system before and after this `dispatch()` line? Basically... *just* the request attributes.

In fact, let's see this. Above dispatch, `dump($request->attributes->all()`. Then copy that... dump after, and `die`. Refresh the article show page. Yep! Before we dispatch the event, the attributes are empty. After? We have `_route`, `_controller`, `slug` and hey! A few *other* things were added by *other* listeners related to security. That's not important for us - but still, interesting!

286 lines | vendor/symfony/http-kernel/HttpKernel.php

```
// ... lines 1 - 39
class HttpKernel implements HttpKernelInterface, TerminableInterface
{
// ... lines 42 - 114
    private function handleRaw(Request $request, int $type = self::MASTER_REQUEST): Response
    {
// ... lines 117 - 120
        dump($request->attributes->all());
        $this->dispatcher->dispatch($event, KernelEvents::REQUEST);
        dump($request->attributes->all());
        die;
// ... lines 125 - 169
    }
// ... lines 171 - 284
}
```

Remove all that debug code.

Seeing the Dumped Route

Before we find out *how* the request attributes are used, I want to show you something kinda cool. We're going to look at a cache file: `var/cache/dev` ... and then `url_matching_routes.php`.

This file is automatically generated by Symfony and is the *end-result* of *all* of the routes in our application. This file is *insane*. After reading our routes, Symfony generates a huge list of regular expressions and which route should match which part, and dumps them to this file. This is used by the route-matching process so that it's *blazingly* fast. It's... pretty amazing.

Anyways, next! Let's see the significance of those Request attributes by continuing to go through the `handleRaw()` method.

Chapter 6: The Controller Resolver

After the kernel.request event is dispatched, it checks if $event->hasResponse() and, *if* that's true... it *returns* the response! Well, it calls $this->filterResponse() and returns *that* - but that's not too important. We'll see that method later. The point is: the function *ends*. The response is returned *immediately* before our controller is executed or anything else.

Listeners to kernel.request can Return a Response

This is kinda cool. If a listener to kernel.request *somehow* already has enough information to return a response... it can do that! It's not super common, it could be used for security or a maintenance page... but hey! Let's try it ourselves!

In src/EventListener/UserAgentSubscriber.php , we can say $event->setResponse() . Not *all* event classes have this setResponse() method - but RequestEvent does. Then say new Response() and set a very important message.

38 lines | src/EventListener/UserAgentSubscriber.php

```
... lines 1 - 6
use Symfony\Component\HttpFoundation\Response;
... lines 8 - 9
class UserAgentSubscriber implements EventSubscriberInterface
{
... lines 12 - 18
    public function onKernelRequest(RequestEvent $event)
    {
        $event->setResponse(new Response(
            'Ah, ah, ah: you didn\'t say the magic word'
        ));
... lines 24 - 28
    }
... lines 30 - 36
}
```

Now when we refresh... yay! Nedry killed *every* page!

Remove that code... and then I'll close a few files.

The Controller Resolver

Back in HttpKernel , so far we've dispatch *one* event. RouterListener listened to that event and modified the request attributes.

Let's keep following the code. This next line is interesting: inside the if we have: $controller = $this->resolver->getController() . This "resolver" thing is a "controller resolver". At a high level, it's beautiful. We know that we will eventually need to execute a controller that will create the Response. This class is *entirely* responsible for *determining* that controller: we pass it the request and - *somehow* - it gives us back a controller, which can be any *callable*.

What Class is the ControllerResolver?

How can we figure out what class $this->resolver is? Well... of course, there's always the handy dd($this->resolver) , which... tells us that this is an instance of TraceableControllerResolver . By the way, whenever you see a class called "Traceable" something, this is almost *definitely* an object that is *decorating* the *real* object in the dev environment in order to give us some debugging info. The *real* controller resolver is inside: Symfony\Bundle\FrameworkBundle\Controller\ControllerResolver .

Another way to figure this out - maybe a slightly nerdier way for a deep-dive course - is with smart use of the debug:container command. The $resolver is one of the arguments to the constructor on top: it's argument 2.

We can see the interface type-hint, but not the concrete class.

Scroll back down, then move over to your terminal and run `php bin/console debug:container http_kernel` - that's the service id of HttpKernel - `--show-arguments`.

This tells me that the second argument is `debug.controller_resolver`. Ok, let's run this command again to get more info about that:

```
$ php bin/console debug:container debug.controller_resolver --show-arguments
```

This uses Symfony's service decoration - a topic we'll see in our next deep-dive tutorial. But basically, when a service is decorated, the *original*, "inner" service's id will be `debug.controller_resolver.inner`. So... run `debug:container` with that!

```
$ php bin/console debug:container debug.controller_resolver.inner --show-arguments
```

And here it is: the *true* controller resolver class is... what we already know: it's called `ControllerResolver` and lives inside FrameworkBundle.

Opening the ControllerResolver Classes

So... let's open that up and see what it looks like! I'll hit Shift+Shift and look for `ControllerResolver.php`. Oh, there are *two* of them: the one from `FrameworkBundle` and another from `HttpKernel`. So... there's some inheritance going on: the `ControllerResolver` from `FrameworkBundle` extends a `ContainerControllerResolver`... which extends the `ControllerResolver` from HttpKernel. The class from `FrameworkBundle` doesn't contain anything important that we need to look at. So I'm actually going to open `ContainerControllerResolver` first. And... yep! Its base class is `ControllerResolver`, which lives in the same namespace. Hold Command or Ctrl and click that class to open it.

Hello ControllerResolver

Ok, time to see what's going on! `HttpKernel` called the `getController()` method. Let's go see what that looks like!

The `getController()` method is passed the `Request` and... oh! Check it out! The first thing it does is fetch `_controller` from the request attributes! So why is `_controller` the "magic" key you can use in your YAML route? It's because of this line right here: the `ControllerResolver` *looks* for `_controller`.

Ultimately, what this method needs to return is some sort of *callable*. For us, it will be a method inside an object, but it can also be a number of other things, like an anonymous function. Let's see what our `$controller` looks like at this point. `dd($controller)`, then move back to your browser and refresh.

222 lines | vendor/symfony/http-kernel/Controller/ControllerResolver.php

```
... lines 1 - 23
class ControllerResolver implements ControllerResolverInterface
{
... lines 26 - 35
    public function getController(Request $request)
    {
... lines 38 - 45
        dd($controller);
... lines 47 - 96
    }
... lines 98 - 222
```

Ah yes: for us, `$controller` is the normal string format we've been seeing: the full controller class, `::`, then the method name.

Remove the `dd()` and let's trace down on the code. This has a bunch of if statements, which are all basically trying to figure out if the controller is maybe *already* a callable. It checks if it's an array and if the 0 and 1 elements are set - because that's a callable format. We're also not an object... and our string is not a function name. Basically, our controller is *not* already a "callable".

So ultimately, we fall down to `$callable = $this->createController()`. Somehow *this* function converts our string into something that can be invoked. How? Let's find out next.

Chapter 7: Who Creates the Controller & Gives it the Container?

In a Symfony app, this `$controller` variable is the string format that comes from the router - something like `App\Controller\ArticleController::homepage`. This function - the `getController()` method of the `ControllerResolver` - has one simple job: it needs to *transform* that string into a PHP *callable*. To do that, it calls `createController()`.

Invokable Classes

Let's scroll down to find this method. Here it is: `protected function createController()` with a string `$controller` argument. The *first* thing it does is *check* to see if the controller does *not* have `::` in the middle. If it does *not* contain `::`, the controller is actually an *invokable* class. This is a strategy for controllers that some people in the Symfony world are using - it's especially popular in ApiPlatform. The idea is that each controller class has only *one* controller method - called `__invoke()`. When a class has an `__invoke()` method, objects of that class are *invokable*: you can execute the object like a function. Anyways, if you use invokable controllers, then your `$controller` string is *just* the class name: no method name is needed.

How Symfony handles invokable controllers is actually *pretty* similar to how it will handle *our* situation: we'll see this `instantiateController()` method in a moment.

Instantiating the Controller Object

Because our controller *does* have a `::` in the middle, it *explodes* the two parts: everything before the `::` is assigned to a `$class` variable and everything after is set to a `$method` variable. Then, inside the try-catch, it *puts* this into a callable syntax: an array where the `0` index is an object and `1` index is the string method name. I know, PHP is weird: but this type of syntax *is* callable.

Of course, on this line, `$class` is *still* just a string. To *instantiate* our controller, it calls - surprise - `instantiateController()`!

This method is *overridden* in the child class. Go over to `ContainerControllerResolver` and find `instantiateController()`. Awesome! It checks to see if the class is in the *container*. And if it is, it doesn't instantiate the controller itself: it *fetches* it from the container and returns it.

How your Controller is Fetched from the Container

This is what's happening in our case: our controller is a *service*. In fact, pretty much *everything* in the `src/` directory is a service... or at least, is *eligible* to be a service - we'll go deeper into that in the next deep-dive tutorial. That's thanks to the `config/services.yaml` file. This section auto-registers everything in the `src/Controller` directory as a service.

So... our controller is a service... and `ContainerControllerResolver` fetches it from the container. But this *only* works because the class name of our controller *matches* its service id. What I mean is: there is a service in the container whose `id` is *literally* `App\Controller\ArticleController`.

This is teamwork in action! The annotation route automatically set the controller string to the *class* name... and because that's *also* the id of the service in the container, we can fetch it out without *any* extra config.

So the truth is, your controller syntax isn't *really* `ClassName::methodName`. It's `ServiceId::methodName`. If your controller service had a *different* id for some reason, that's ok! In that case, you would set your controller to your *service* id `::` then method name in YAML. There's also a way to do this in annotations.

Fetching your controller from the container *also* works because controller services are *public*. Really, they're the *only* services that we routinely make public. If you look back at `services.yaml`, it's not immediately obvious *why* they're public - I don't see a `public: true` anywhere. I'll save the details for the *next* deep-dive tutorial, but the controller services are public thanks to this *tag*. *One* of the things it does is make all of the services *public* so that the `ContainerControllerResolver` can fetch them directly.

The Old Way: Direct Instantiation

If, for *some* reason, your controller is *not* registered as a service, then it calls `parent::instantiateController()`, which... could not be simpler. It says `new $class()` and passes it *no* arguments. That's basically legacy at this point: it's how controllers we created *prior* to Symfony 4.

The Final Callable Controller Result

Scroll back up in `ControllerResolver` to `getController()`. This is all a *long* way of saying that our controller string - this `App\Controller\ArticleController::homepage` - is split into two pieces, the service is fetched from the container, and it's returned from here in a callable format.

Close both of the controller resolver classes and head back to `HttpKernel`. Let's see what this final *$controller* looks like. After the `if`, `dd($controller)`.

284 lines | vendor/symfony/http-kernel/HttpKernel.php

```
... lines 1 - 11
namespace Symfony\Component\HttpKernel;
... lines 13 - 39
class HttpKernel implements HttpKernelInterface, TerminableInterface
{
... lines 42 - 114
    private function handleRaw(Request $request, int $type = self::MASTER_REQUEST): Response
    {
... lines 117 - 130
        dd($controller);
... lines 132 - 167
    }
... lines 169 - 282
}
```

Ok, move over... and refresh. That's it! The weird PHP callable syntax: an array where the `0` index is an `ArticleController` *object*, and the `1` index is the string `homepage`.

Controllers: Boring Services

Go ahead and remove that `dd()`. So... this is *beautiful*. Our controller is a *boring* service object: there's nothing special about it at all. Need to use a service like the logger? No problem! In `ArticleController`, add another argument to the constructor: `LoggerInterface $logger`. I'll hit Alt + Enter and go to "Initialize Fields" to create that property and set it. To prove it's working, let's say `$this->logger->info('Controller instantiated!')`.

76 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
use Psr\Log\LoggerInterface;
... lines 10 - 13
class ArticleController extends AbstractController
{
... lines 16 - 19
    private $logger;
... line 21
    public function __construct(bool $isDebug, LoggerInterface $logger)
    {
... line 24
        $this->logger = $logger;

        $this->logger->info('Controller instantiated!');
    }
... lines 29 - 74
}
```

Move over, refresh, click a link to open the profiler and go to the Logs section. *Cool*. The first log is from our listener to `kernel.request`, then our controller is instantiated and *then* it's executed.

So yea! Our controller is a *boring* service. Well, it *does* have that superpower where you can autowire services into controller *methods* - but we'll learn how that works in a few minutes.

I *do* have one more question, though. The controller is full of shortcut methods like `$this->render()`. How does that work? We never injected the `twig` service... so how is our "boring, normal service" using something that we didn't inject? How is it getting the `twig` service?

Let's dig into that mystery next!

Chapter 8: How does the Controller Access the Container?

Our controller is a *beautiful*, boring service. I *love* boring things. This means that, if we need to access some *other* service from here, we need to "inject" it - either through the constructor *or* by autowiring it as an argument to the controller method - a special superpower of controllers that we'll talk about soon.

The point is: we can't just "grab" a service out of thin air that we haven't injected.... which is why *I'm* wondering: how the heck does this `render()` shortcut method work? *Certainly* that uses the `twig` service... but we have *not* injected that into our class

Let's go digging! Hold command or control and click `render()` to jump into our parent class: `AbstractController`. This method basically just calls `renderView()`, which is right above us.

Hmm: `renderView()` apparently fetches the `twig` service directly from the container. But, hold on a second. How did our controller service get access to the container? Because, it's not like we're injecting it via autowiring or any other way. So... *who* is populating the `$this->container` property?

Oh, but it's even *more* mysterious than this. Search for `parameter_bag` in `AbstractController` to find a method called `getParameter()`. This method fetches a service directly from the container called `parameter_bag`.

Let's get some info on this service. Find your terminal and run:

```
$ php bin/console debug:container parameter_bag
```

Woh. It's public *false*! This is *not* a service that you should be able to fetch out of the container directly by saying `$this->container->get('parameter_bag')`. It should give us an error! So what the heck is going on?

Service Subscriber Magic

Here's the answer: our controller extends `AbstractController`. And `AbstractController` implements a special interface called `ServiceSubscriberInterface`. This is actually something we talk about in one of our Doctrine tutorials.

When you implement `ServiceSubscriberInterface`, it forces you to have a method called `getSubscribedServices()` where you return an array that says *which* services you need inside of this class. Then, Symfony will pass you a "mini" container that *holds* all of these services.

At the top of `AbstractController`, see this `setContainer()` method with a `ContainerInterface` type-hint? That will *not* be the real container. Nope, that will be the "mini-container" that holds all the services from our `getSubscribedServices()` method. And because our controller is an autowired service... and this method has `@required` above it, Symfony knows to call `setContainer()` immediately after instantiating this object.

This is what gives our controller the ability to fetch all those services that *we* didn't inject directly. It also fetches them *lazily*: none of the services are instantiated *unless* we need to use them.

So... our controller is not *just* a boring, normal service: it has this extra superpower. But... this is actually something that... *any* service in our system can implement - it's not special to controllers. So once again, our controller *is* beautifully boring and normal.

Next: we *now* have a callable controller! Let's keep going through `HttpKernel` to see what happens next. Because... one *big* thing we're still missing is what *arguments* we should pass to the controller.

Chapter 9: The Argument Resolver

Inside HttpKernel, we now have the controller. But before we run around excitedly and try to *call* that controller... we need to figure out what *arguments* to pass to it.

To help see this clearly, in ArticleController::show(), we need to make one small change. Instead of having an argument type-hinted with Article and allowing Symfony to automatically query for it by the slug, let's temporarily do this manually. Remove that argument and replace it with $slug. Now add another arg: ArticleRepository $articleRepository so that we can make the query: $article = $articleRepository->findOneBy(['slug' => $slug]). And then, if *not* $article, throw $this->createNotFoundException().

76 lines | src/Controller/ArticleController.php

```
// ... lines 1 - 13
class ArticleController extends AbstractController
{
// ... lines 16 - 45
    public function show($slug, SlackClient $slack, ArticleRepository $articleRepository)
    {
        $article = $articleRepository->findOneBy(['slug' => $slug]);

        if (!$article) {
            throw $this->createNotFoundException();
        }
// ... lines 53 - 60
    }
// ... lines 62 - 74
}
```

Functionally, this is identical to what we had before... but it will help us with our deep-dive. By the way, *we* know that this createNotFoundException() line will result in a 404 page. If you hold Command or Ctrl and click into that method, it returns a NotFoundHttpException. So... for *some* reason, *this* specific exception maps to a 404... while most *other* exceptions will cause a 500 page. By the end of this tutorial, we'll know *exactly* why this happens.

The kernel.controller Event

Go back to HttpKernel. Now that we've figured out what the controller is, the next thing that happens is... we dispatch another event! This one is called KernelEvents::CONTROLLER, which maps to the string kernel.controller.

So let's look at *everything* we've done so far: we dispatched an event, found the controller, then dispatched another event. That's *all*.

There are no particularly important listeners to this event, from the perspective of how the framework operates. Refresh the article show page... and click to open the profiler. Go to the Events tab... and find kernel.controller.

In this app, there are 6 listeners... but nothing critical. A few of them come from FrameworkExtraBundle - a bundle that gives us a *lot* of magic shortcuts. These rely *heavily* on listeners... and we'll talk about how some of them work later.

One of the things that a listener to this event can do is *change* the controller. It's not very common, but you can see it down here: $controller = $event->getController(). Hold Command or Ctrl to open the ControllerEvent class. Here it is: a listener can call setController() to *completely* change the controller to some *other* callable.

The Argument Resolver

Ok, back in HttpKernel after the kernel.controller "hook point", this next line is the missing piece: we need to know what arguments we should pass when we call the controller. To figure that out, it uses something called the "argument resolver". And it's pretty cool... we call `getArguments()`, pass it the `$request` and `$controller` and it - *somehow* - figures out all the arguments that this controller should be passed.

Ok, you know the drill: let's open this thing up and see how it works! This time, the class is simple: I'll hit Shift+Shift and open a file called `ArgumentResolver.php`. Find the `getArguments()` method.

Okay, interesting. It *first* uses a `foreach` to loop over `$this->argumentMetadataFactory->createArgumentMetadata()` as `$metadata`. This is actually looping over each *argument* to the controller function. So for the show page, this would loop 3 times: once for each argument.

The ArgumentMetadata

Then, *inside* that loop, it does another: it does a `foreach` over something called `argumentValueResolvers`. Let's see what's going on here. Inside the first loop, `dd()` the `$metadata` variable: this should be *something* that, sort of, represents a single argument.

98 lines | vendor/symfony/http-kernel/Controller/ArgumentResolver.php

```
... lines 1 - 27
final class ArgumentResolver implements ArgumentResolverInterface
{
... lines 30 - 45
    public function getArguments(Request $request, callable $controller): array
    {
... lines 48 - 49
        foreach ($this->argumentMetadataFactory->createArgumentMetadata($controller) as $metadata) {
            dd($metadata);
... lines 52 - 81
        }
... lines 83 - 84
    }
... lines 86 - 96
}
```

Move over and refresh. Huh. Apparently this is an `ArgumentMetadata` object, which holds the name of the argument - `slug`... because that's the name of the first argument to the controller. It also holds the argument `type`, which in this case is `null`. For the second argument it would be `SlackClient`. It has some other stuff too: like if the argument has a `defaultValue` or `isNullable`.

That's... really cool! It's *all* the metadata about that one argument. The next question is: what does this function *do* with that metadata?

The Argument Value Resolvers

Clear out the `dd()`. Let's figure out what these `$argumentValueResolvers` are. This argument is actually an iterator - it has an `iterable` type... which is *not* important... except that we need to get fancy to see what's inside. `dd(iterator_to_array($this->argumentValueResolvers))`.

98 lines | vendor/symfony/http-kernel/Controller/ArgumentResolver.php

```
... lines 1 - 27
final class ArgumentResolver implements ArgumentResolverInterface
{
... lines 30 - 45
    public function getArguments(Request $request, callable $controller): array
    {
... lines 48 - 49
        foreach ($this->argumentMetadataFactory->createArgumentMetadata($controller) as $metadata) {
            dd(iterator_to_array($this->argumentValueResolvers));
... lines 52 - 81
        }
... lines 83 - 84
    }
... lines 86 - 96
}
```

Move over and... 8 items! Each object is being decorated by a `TraceableValueResolver`. But if you look inside - I'll expand a few of these - you'll see the *true* object: `RequestAttributeValueResolver`, `RequestValueResolver`, and a `SessionValueResolver`. *These* are the objects that figure out which *value* to pass to each controller argument.

Another way to see this - since this is a deep-dive tutorial - is to find your terminal and run:

```
$ php bin/console debug:container --tag=controller.argument_value_resolver
```

Because if you want to create your *own* argument value resolver - we'll do that later - you need to create a service and give it *this* tag. This gives us the same list - but it's a bit easier to see the *true* names: some `request_attribute` resolver, `request` resolver, `session` resolver, `user_value_resolver` and more.

We're going to walk through some of the most important value resolvers next.

How Argument Value Resolver Works

But before we do, let's go back and... see how this system works! We loop over each argument... and then loop again over every argument value resolver and call a `supports()` method on each. So, one-by-one, we're asking each argument value resolver:

> Hey! Do you know what value to pass for a `$slug` argument with no type-hint?

Or, on the next loop,

> Yo! Do you know what value to pass to a `$slack` argument with a `SlackClient` type-hint?

If an argument resolver returns `false` from `supports()`, then it continues onto the next one. If it returns `true`, it *then* calls `$resolver->resolve()` to get the value.

So - *hopefully* - by the end of looping through all the argument value resolvers, one of them has figured out what value to pass to the argument.

Next, let's open up the *most* important argument value resolvers and figure out what they do. This will answer a cool question: what are *all* the possible arguments that a controller is allowed to have... and why?

Chapter 10: Argument Value Resolvers

We just learned that when Symfony tries to figure out what arguments to pass to your controller, it calls this `ArgumentResolver::getArguments()` method. It loops over the arguments one-by-one and *then* loops over these things called `argumentValueResolvers`, to see which one can figure out *what* to pass to each argument.

To see a list of all of the argument value resolvers, we ran:

```
$ php bin/console debug:container --tag=controller.argument_value_resolver
```

They're all decorated inside a `TraceableValueResolver` class, but you can see, kind of by their name, what's really inside. So, I want to know: what are *all* the possible arguments that I'm allowed to have on my controller? To find out, let's look inside the *most* important of these argument value resolvers.

RequestAttributeValueResolver: Wildcards as Arguments

Remove the `dd()` from `ArgumentResolver`. I'll hit Shift + Shift to open up my *favorite* resolver: `RequestAttributeValueResolver.php`.

Perfect. Remember, as it loops over the arguments, the first thing `ArgumentResolver` does is call `supports()` on each of these to figure out if *this* value resolver can help. It passes us the `$request` and the `ArgumentMetadata`.

To see how this works, `dd()` `$request->attributes->all()` and also `$argument`.

```
42 lines | vendor/symfony/http-kernel/Controller/ArgumentResolver/RequestAttributeValueResolver.php
... lines 1 - 22
final class RequestAttributeValueResolver implements ArgumentValueResolverInterface
{
... lines 25 - 27
    public function supports(Request $request, ArgumentMetadata $argument): bool
    {
        dd($request->attributes->all(), $argument);
... line 31
    }
... lines 33 - 40
}
```

Because... check this out! This class *uses* the now-famous `$request->attributes`.

Move over and refresh the article show page. *Beautiful*. The request attributes have what we expect: the stuff from the router. And because this is the *first* time through the loop, it's asking us if we know what value to pass to the `slug` argument.

In the `supports()` logic, if you ignore the `isVariadic()` part - that's not too important - what this basically says is:

> I can provide the value for the argument *if* the name of the argument - `slug` in this case - is inside `$request->attributes`.

Down in `resolve()`... yea! It literally returns `$request->attributes->get($argument->getName())`.

This is *huge*! The *first* thing we learn about Symfony routes and controllers is that if you have a `{slug}` wildcard in your route, you're *allowed* to have a `$slug` argument in your controller. Why does that work? Now we know! It's a two step process. First, the router puts all the wildcard values into `$request->attributes`. And second, this `RequestAttributeValueResolver` looks *into* `$request->attributes` using the argument name and returns the value if it's there. This class is what gives us this *fundamentally* important functionality.

RequestValueResolver: Request Argument

But that's not the only cool argument value resolver! Remove the `dd()` and... let's go open another one! I'll hit Shift + Shift to open a class called `RequestValueResolver`. There it is!

What are some *other* things that we know we are allowed to have as arguments? Let me find an example... hmm... I'll open up `ArticleAdminController`. Here we go: one of the other things you can do is add an argument that's type-hinted with the `Request` class. If we do that, we get the request.

How does this work? It's thanks to `RequestValueResolver`. This one is dead simple. It says:

> Hey! If this argument is type-hinted with the `Request` class... or a sub-class... pass the request!

That's *precisely* what `supports()` checks for. And `resolve()` couldn't be shorter.

SessionValueResolver for SessionInterface Argument

Ok, what else is there? I'm going to go to the directory tree on top and double-click this `ArgumentResolver` folder. That moves us *into* this directory on the left... which is cool because this is *full* of other argument resolvers!

A few of these are similar to `RequestValueResolver` - like `SessionValueResolver`. You may or may not know this, but you can type-hint an argument with `SessionInterface` and you'll get the session. That works thanks to this resolver.

UserValueResolver for UserInterface Argument

Another resolver lives in a different directory - I'll hit Shift+Shift to open it: `UserValueResolver.php`. This resolver allows you to type-hint `UserInterface` on an argument to get your security User object.

The Amazing ServiceValueResolver

At this point, if we look back at `ArticleController::show`, we now know how the first argument works, but... we haven't seen a resolver yet that explains the next two. The second and third arguments are type-hinted with *services*. Where is the magic that allows us to type-hint a service in a controller method?

The answer to that is the `ServiceValueResolver`. It's *such* a cool class, that let's look at it in depth, next.

Chapter 11: How Service Autowiring Works in a Controller Method

The *one* common type of controller argument that we *haven't* explained yet are for arguments that are type-hinted with an autowireable service. How does that work? This class - `ServiceValueResolver` - is responsible. It's, honestly, a piece of genius from Nicolas Grekas - the person who wrote it. And it's one of my favorite things to look at.

Inside `supports()`, before we even look at what it's doing, let's `dd($argument)`. When we refresh... there we go. The *first* time this is called is actually for the *second* argument to our controller. Why? Because the first argument was handled by `RequestAttributeValueResolver` before this was ever called. It's not usually important, but these value resolvers can be given a priority.

95 lines | vendor/symfony/http-kernel/Controller/ArgumentResolver/ServiceValueResolver.php

```
... lines 1 - 24
final class ServiceValueResolver implements ArgumentValueResolverInterface
{
... lines 27 - 36
    public function supports(Request $request, ArgumentMetadata $argument): bool
    {
        dd($argument);
        $controller = $request->attributes->get('_controller');
... lines 41 - 56
    }
... lines 58 - 93
}
```

Anyways, the `$slack` argument is the first one that hits our `supports()` method. And the *key* thing is that this argument has a type of `App\Service\SlackClient`.

ServiceValueResolver::supports()

Let's look at the logic. Hey! It's our old friend request attributes! We just can't seem to get away from you. The first thing this method does it get the controller with `$request->attributes->get('_controller')`. For us, that's the now-familiar `ClassName::methodName` string format.

Next, it does some normalization of the format... which isn't relevant to us: it's just trying to make sure that the `$controller` variable is *ultimately* a string.

Finally, we hit an if statement that says: if not `$this->container->has($controller)`. Hmm. It seems like it's... checking to see if our controller is a service?

The Controller Argument ServiceLocator

Actually, no: it's doing something *totally* different. To see what's going on, before the return, `dd($controller)` and also `$this->container` so we can see what it looks like.

```
96 lines | vendor/symfony/http-kernel/Controller/ArgumentResolver/ServiceValueResolver.php
... lines 1 - 36
    public function supports(Request $request, ArgumentMetadata $argument): bool
    {
... lines 39 - 50
        if (!$this->container->has($controller) && false !== $i = strrpos($controller, ':')) {
            $controller = substr($controller, 0, $i).strtolower(substr($controller, $i));
        }

        dd($controller, $this->container);
... lines 56 - 57
    }
... lines 59 - 96
```

Now... refresh! Ok, the controller is no surprise: it's the `ClassName::methodName` string syntax. But check out `$this->container`. This is *not* the main Symfony container. This is - *once again* - one of those *small* containers, called a service locator.

The details about how this class works aren't too important... but you can browse the `$serviceMap` property to see what's *inside* of this container. Apparently it holds 34 services... and weird, it has one service for *every* single controller method in our system. The id is the full controller string, including the `::methodName` part.

So... this is weird. What is this thing? To make sense of it, let's also `dd()` `$this->container->get($controller)`. That's eventually what the last line of `supports()` calls.

```
96 lines | vendor/symfony/http-kernel/Controller/ArgumentResolver/ServiceValueResolver.php
... lines 1 - 36
    public function supports(Request $request, ArgumentMetadata $argument): bool
    {
... lines 39 - 54
        dd($controller, $this->container, $this->container->get($controller));
... lines 56 - 57
    }
... lines 59 - 96
```

Refresh now. This last dump for `$this->container->get($controller)` gives us... *another* mini-container! And if we look at the `$serviceMap` property, it has two things: `articleRepository` and `slack`... which *exactly* match the two argument names that are type-hinted with services in our controller!

So... in reality, the `$this->container` property is *basically* a big array of mini "containers". More technically, it's a service locator for *every* controller function in our system. And each of *those* locators contain all the services for all of the arguments that are type-hinted with a service.

Thanks to that, if we look down in `resolve()`... and skip passed some normalization, it ultimately yields `$this->container->get($controller)` - to get the mini-container - then `->get($argument->getName())` to get the specific service for the `$slack` or `articleRepository` argument.

So we have a big container, full of containers... which are full of services. How crazy is that?

How the Controller Containers are Built

The *truly* amazing part is how Symfony figures all of this out. *All* of the logic for building this container of containers is done when your cache is built: there is *zero* runtime overhead.

The key behind this working is hiding inside of *your* `config/services.yaml` file. Have you ever wondered why the `src/Controller` directory has its *own* import section? It's not *strictly* needed... because classes in this directory are *already* registered as services thanks to the import above.

The reason is this tag... which does *two* things. First, it makes these services *public*. We talked about that earlier: controller services must be public so that the controller resolver can fetch that service out of the main container at runtime.

The *second* thing it does is more interesting. When Symfony is building the container cache, it looks for *all* of the services that have this tag. It then finds all the public methods on the classes and uses autowiring to figure out all the arguments that are type-hinted with an autowireable service. It uses *that* info to create this final container of containers.

This was a *key* innovation in Symfony 3.4 that allowed us to use autowiring in our controller methods.

Head back to `ServiceValueResolver` and, back up here, let's remove the `dd()` .

Route Defaults can be Arguments

So... that's basically it for the main argument value resolvers. If your argument name matches a route wildcard, then that's *allowed* as an argument. And actually, now that we understand that these wildcard values go into the request attributes *and* that any route *defaults* are added to the same place, open `config/routes.yaml` . Thanks to this `totally_inventing_this_default` key that we added to `defaults` , this will be put into the request attributes and we *could* add an argument to our controller with this name.

One Missing Piece: Auto-querying Entity Arguments

But we still haven't explained *one* thing. Open `ArticleAdminController` and find the `edit()` action. It doesn't explain why I can type-hint an entity class and... *something* queries for it automatically and passes us the object.

We'll learn how that works later: it uses a bit of an *older* system inside of Symfony.

Next, let's go back to `HttpKernel` and continue our journey. We now have the controller *and* the arguments. Yep! It's time to actually *execute* the controller.

Chapter 12: Calling the Controller & View Event

Ok, let's remember what we've done so far... because it's really not *that* much.

The kernel.controller_arguments Event

We dispatched an event, found the controller, dispatched another event, found the arguments, and... guess what? Now we're dispatching *another* event called `KernelEvents::CONTROLLER_ARGUMENTS`, which resolves to the string `kernel.controller_arguments`.

There are no core, important listeners to this: it's just another hook point. The only difference between this event and `kernel.controller` is that this event has access to the *arguments*. So if you needed to do something that's *based* on what arguments we're *about* to pass to the controller, this is your event. Listeners to this event *also* have the ability to change the controller *or* the arguments via setter methods on the event class.

Calling the Controller

What's next? It's the moment we've *all* be waiting for. Drum roll.. we call the controller! Yes! We *knew* that the code that executed the controller lived *somewhere*... and here it is. It's delightfully simple: `$controller(...$arguments)`. I love that.

The kernel.view Event

And of course, what does our Symfony controller *always* return? A `Response`. Unless... it doesn't. It turns out, your controller does *not* actually need to return a `Response` object. If it returns something else, then you end up inside the next if statement. And... what does Symfony do? You can kinda start guessing whenever I ask that question. The answer is *pretty* much always: Symfony dispatches an event. This time it's `KernelEvents::VIEW`, which means `kernel.view`.

When Symfony dispatches this event, it's basically saying:

> Yea... so, this controller returned something that's *not* a response... and I kinda need a response that I can send back to the user. Can any listeners to this event, somehow, *transform* whatever the controller returned into a response? That would be schweet!

This is, kind of, the "view", or "V" in a true MVC framework. Normally our controller return a response directly. But instead, you *could*, for example, return... an `Article` entity! You would return a "model". Then, you could write a listener to this event that transforms that `Article` entity into HTML by rendering the template.

This event isn't used anywhere in Symfony's core, but it *is* used *extensively* inside API Platform. Internally, their controllers return *objects*. Then, they have a listener - actually a few that work together - that *transforms* that object into JSON, XML or whatever format the user is requesting.

After dispatching the event, it checks to see if the event has a response. Hold Command or Ctrl and click to open the `ViewEvent` class. If you listen to this event, you have access to what the controller *returned*. Then... I'm going to open the parent class... a listener can call `setResponse()` with whatever `Response` it created.

So if *one* of the listeners sets the response, then `$event->hasResponse()` will be true and... we're saved! We *do* now have a response. But if *none* of the listeners are able to set a response... sad trombone... then *finally*, Symfony panics! It says that controller must return a `Response` object, but it returned something different. And then, one of my *favorite* parts of this whole system: if `null === $response`, it politely adds:

> Did you forget to add a return statement somewhere in your controller?

Ah yes, I *have* forgotten that, many times. Let's forget it now! Add a `return` in our `show()` action, then spin over, refresh and... enjoy the error!

77 lines | src/Controller/ArticleController.php

```
... lines 1 - 13
class ArticleController extends AbstractController
{
... lines 16 - 45
    public function show($slug, SlackClient $slack, ArticleRepository $articleRepository)
    {
        return;
... lines 49 - 61
    }
... lines 63 - 75
}
```

Go back and remove that.

At this point, we *definitely* have a `Response` object: either our controller returned a `Response` or a listener to `kernel.view` was able to create one. So *finally*, down here... we're done! We made it! We return `$this->filterResponse($response)`. What does *that* method do? Do you remember what you're *always* supposed to answer when I ask that question? Yep, it dispatches *another* event! Let's look into that next and dig into a few *really* cool listeners on it, including one that takes us into the topic of the "request format".

Chapter 13: kernel.response Event & Request Format

No matter what, Symfony's job is to look at the incoming request and *somehow* convert that into a response. Well, that's *really* the job of `HttpKernel`, specifically the `handleRaw()` method.

At this point we have a `Response` object. Yay! Either our controller returned the response or a `kernel.view` listener created it. Then, at the bottom, our last act is to return `$this->filterResponse($response)`.

Before we look at that method, there was *one* other place during this process where we could have *already* returned a response. Scroll to the top. Here: listeners to the *very* first event - `kernel.request` - have the ability to set a `Response` immediately. If they do, this *also* calls `$this->filterResponse()`.

The kernel.response Event

The point is: no *matter* how we get the `Response`, eventually, `filterResponse()` will be called. Hold Command or Ctrl and click that method to jump to it. What a shock! This method dispatches yet *another* event! This one is called `KernelEvents::RESPONSE`, which is the string `kernel.response`.

There are *several* interesting listeners to this event. Go back to your browser, refresh the page, click to go into the profiler and then into the Events section. *Way* down... here it is: `kernel.response`.

WebDebugToolbarListener

There are many cool, subtle, listeners to this event, like the `WebDebugToolbarListener`! Listeners to this event have access to the final `Response` object. I'm going to open the original page in a new tab... and view the HTML source. Very simply, the `WebDebugToolbarListener` checks to see if this is a full HTML page and, if it is, it *modifies* the HTML in the `Response` and *adds* a bunch of JavaScript at the bottom. This JavaScript is what's responsible for rendering the web debug toolbar. That's *such* a cool example.

ResponseListener

As far as understanding the *mechanics* of the request-response flow, there are actually *no* critical listeners to this event... but there *are* still some pretty important ones, like `ResponseListener`. Let's open that one up.

I'll hit Shift+Shift to open `ResponseListener.php`: get the one from `http-kernel/`, not `security`. It says:

> `ResponseListener` fixes the `Response` headers based on the `Request`.

Let's... find out what that means. Inside `onKernelResponse()`... the most important part is that this calls `$response->prepare($event->getRequest())`. Ok! Hold Command or Ctrl to jump into *that*.

Here's the idea: once the `Response` has been created, this *checks* the `Response` to see if it has some missing pieces. For example, if the response doesn't have a `Content-Type` header yet - if that's not something that *we* set in the controller - this uses some information from the *request* to set that header *for* you.

Request "Preferred Format"

This touches on an important, but *subtle* detail inside Symfony. I'm curious how the `getPreferredFormat()` method works. Hold Command or Ctrl to jump into *that*.

I won't go into the full details of this method, but here are the basics. This method's job is to try to determine what *format* - like `html` or `json` - that the client - the "thing" that's sending the request - *prefers* for the response. Basically: does the client want an HTML response, a JSON response or something else?

To figure that out, it does two things. First, it calls `$this->getRequestFormat()`. Basically, it checks to see if something else has explicitly said:

> This is *definitely* the format that the user is requesting

This can be set in 2 different ways. First, by calling `$request->setRequestFormat()`, which you could do, for example, in a listener. At this time, there is only *one* place in *all* of core that calls this function: the `json_login` authentication system calls `$request->setRequestFormat('json')`.

The *second* way the request format can be set is by putting an `_format` key into the request attributes! That most commonly happens by adding an `_format` *default* to your route *or* by actually having an `{_format}` route wildcard.

If *neither* of those things happens - which is the "normal" way that things work - then this method *next* loops over the "acceptable content types" to find one that works. This reads the `Accept` header on the `Request`, loops over them with the highest priority format first, and, as soon as it finds one that it understands, that's returned.

The *big* point here is this: Symfony has a concept of "what response *format* does this request *prefer*?". It uses that to set the `Content-Type` header. This idea is going to be important again *later* with error handling.

By the way, this line *may* change to use `$request->getRequestFormat()` at some point in the future... which just means that if you *like* this idea of it automatically setting the `Content-Type` response header based on the `Accept` request header, make sure you do it explicitly when you create your `Response`.

Close the `Response` class and `ResponseListener`. Before we boldly push forward, there are a *few* other listeners I want to look at. Let's check those out next and then... yes... send the final response to the user! We're close.

Chapter 14: Finishing the Request

There are several other interesting listeners to `kernel.response`. Here's one: `ContextListener`... and it's from security! Open that up: Shift+Shift, `ContextListener.php`.

ContextListener: Loading the Security User from the Session

Scroll down to find the method we care about: `onKernelResponse()`. It says:

> Writes the security token into the session.

If you use a "stateful" firewall... which you probably *are*, unless your security system is a pure, API token-based system, then *this* is the class that's responsible for taking your authenticated `User` object - technically the "token" object that holds it - and saving it into the session. Here it is: `$session->set($this->sessionKey, serialize($token))`.

This class is *also* responsible for *unserializing* the token at the start of each request - that's in a different method.

DisallowRobotsIndexingListener

Close this class and look back at the event list. Let's see... there's a listener called `DisallowRobotsIndexingListener`, which adds an `X-Robots-Tag` header set to `noindex` *if* you set a `framework.disallow_search_engine_index` option to *true*. Phew! That option *defaults* to `true` in dev... which is why we see this. So if you... *accidentally*... deploy your site in dev mode, it won't be indexed.

SessionListener

Let's look at *one* more: `SessionListener`. Open that one up: Shift+Shift then `SessionListener.php`.

This class is responsible for actually *storing* the session information. It extends `AbstractSessionListener`... which holds the majority of the logic

This also listens on the `kernel.request` event... but we're interested in the `onKernelResponse()` method. It does several things... but eventually, it *calls* `$session->save()` to actually *put* your session data into storage. All these tiny invisible pieces help make your application sing.

kernel.finish_request & RequestStack

Ok, *enough* playing with these listeners. Close the two session classes and go back to `HttpKernel`. After dispatching the `kernel.response` event, this calls a `finishRequest()` method and then *finally* returns the `Response` that's on the event. Let's see what `finishRequest()` does. Ah! It dispatches one *more* event and then calls `$this->requestStack->pop()`.

Remember: this `RequestStack` object is basically a collection of request objects - something we'll talk more about soon. The `pop()` method *removes* the most-recently-added `Request` object *from* that collection. If you scroll back up to the top of `handleRaw()`, the `pop()` call does the *opposite* of `$this->requestStack->push($request)`. So... we don't know *why* this request stack thing needs to exist... but we *at least* know that the current `Request` object is *added* to the `RequestStack` at the beginning of handling the request, and then *removed* at the end.

Returning the Response to the User

So... we're done! The `filterResponse()` method returns the `Response`, then `handleRaw()` returns the same `Response`... and then `handle()` *also* returns the `Response`... all the way back to `index.php`: `$response = $kernel->handle($request)`.

We made it! But we haven't *sent* anything to the user yet: everything is still just stored in PHP memory. The next call takes care of that: `$response->send()`. I'll open that up. It's just a *fancy* way of calling the PHP `header()`

function to set all the headers and then echo'ing the content. At this point, our response is *sent*!

kernel.terminate: The Final Event

Back in `index.php`, there's *one* final line: `$kernel->terminate()`. Let's find that inside of `HttpKernel`. And... wow. I'm personally *shocked*. This dispatches one *final* event.

This event is dispatched *so* late that... if your web server is set up correctly, the response has *already* been sent to the user! This event isn't used too often... but it *is* where all the *data* for the profiler is stored, for example. In fact, that's the *only* listener to this event: `ProfilerListener`.

So *that* is Symfony's request-response process in depth. A work of art.

It may have seemed like a lot, but if you zoom out, it's delightfully simple: we dispatch an event, find the controller, dispatch an event, find the arguments, dispatch an event, call the controller, dispatch 2 more events in `filterResponse()` and `finishRequest()` and then, back in `index.php`, we send the headers, echo the content and dispatch one *last* event. It's... kind of a "find the controller, call the controller" system... with a *ton* of events mixed in as hook points.

But go back to `HttpKernel` and scroll *all* the way back up to `handle()`. Ah yes, this wraps *all* of our code in a try-catch block. So what happens if an exception is thrown from somewhere in our app? Well, quite a lot. Let's jump into that next.

Chapter 15: Exception Handling

We've now walked through the "happy" path: we know how a successful request is converted into a response. But what about the *unhappy* path. Well, our *entire* application is wrapped in this try-catch block. So, if an exception is thrown from *anywhere*, this will catch it. The `$catch` argument is `true` by default, so we're not going to hit this `finishRequest()` line. Nope, if an exception is thrown, this will call `$this->handleThrowable()`.

And, this an interesting situation. Because no matter *what* went wrong in your app - like your database is being eaten by robots, or your PHP code has become self-aware and is rewriting itself, *ultimately*, you need to return a *response* to the user... even if that's just a photo of those robots cooking your servers over a slow fire. Somehow, *something* needs to convert the `Exception` into a `Response`. *That* is the job of `handleThrowable()`.

Hello kernel.exception Event

Hold Command or Ctrl and click to jump down to this method. What does `handleThrowable()` do? I know this may come as a shock to you... but it dispatches *another* event: `KernelEvents::EXCEPTION`, or `kernel.exception`.

Move over and refresh the page. Click into the profiler and then into Events. Now, *this* page did *not* throw an exception. So this event was *not* dispatched. But we can click on "Not Called Listeners" to find it. Let's see... I went *right* past it: here it is. The `kernel.exception` event has about 5 listeners.

How the "Welcome to Symfony" Page is Rendered

The most important one is `ErrorListener`, which we're going to look at in a few minutes. But check this out: `RouterListener` - the *same* class we looked at earlier - *also* listens to this event. Why? This powers a *minor* feature in Symfony... but the *way* that it accomplishes it is a *super* fun example of how different parts of Symfony work together. Let's dive in!

I'll hit Shift+Shift to open up our old friend: `RouterListener.php`. Let's see... I'm looking for `getSubscribedEvents()`. There it is: on `KernelEvents::EXCEPTION`, call `onKernelException()` with a priority of `-64`... which means that it will be called fairly late compared to other listeners.

Find the `onKernelException()` method. The *purpose* of this method is to render a nice "Welcome to Symfony" page when you start a *brand* new project that has no homepage. That's what this `createWelcomeResponse()` does: it renders a PHP template. Let's see that: Shift + Shift to open `welcome.html.php`. Here it is: "Welcome to Symfony!" with links to the docs and other things. If you start a *brand* new Symfony 5 project, this is what you would see.

I *love* this page... because it's really cute. But for our purposes, I want to know how this works. Back in `RouterListener`... actually, look back at `HttpKernel`. Listeners to this event are passed an `ExceptionEvent` object... and the actual exception that was *thrown* - that's the `$e` variable - is passed *into* that object! That makes sense: listeners to this event will *probably* need to know *which* exception was thrown.

In `RouterListener`, it checks to see if the exception that was thrown is an instance of `NotFoundHttpException`. Which, by the way, is the *exact* exception class that *we* throw in a controller whenever we want a 404 page. It's inside the `createNotFoundException()` method: `return new NotFoundHttpException`. That's not important for understanding how this welcome page is rendered... but it *is* interesting that this exception class keeps popping up.

Anyways, if the exception is *not* a `NotFoundHttpException`, this listener does nothing. But if it *is*, it then checks to see if the *previous* exception is an instance of `NoConfigurationException`. If so, it renders the welcome page.

So... then... what's going on exactly? Here's the story: when the Router is executed, if no route is matched and the URL is `/`, it throws a `NoConfigurationException`.

If you scroll up to `onKernelRequest()`, remember, *this* is what executes the router. Specifically, the `matchRequest()` method throws that exception. The `NoConfigurationException` *extends* `ResourceNotFoundException`. That's important because this *entire* block is wrapped in a try-catch: it's catching `ResourceNotFoundException`.

So in general, if the router can't match a route, it throws a `ResourceNotFoundException`. But in this *one* special case - when you're accessing the homepage and no route was found - it throws a *subclass* called `NoConfigurationException`.

So it catches the exception and *throws* a new `NotFoundHttpException` but set the exception from the *router* as the *previous* exception. Ultimately, this `NotFoundHttpException` is thrown from this method, *caught* by the try-catch in HttpKernel and put into the `ExceptionEvent` object.

Then, `RouterListener` listens to the `kernel.exception` event and if the exception is a `NotFoundHttpException` whose *previous* exception is `NoConfigurationException`... it renders the welcome page.

Go team!

Manually Rendering the Welcome Page

For fun, let's see if we can trigger the welcome page manually. Go to `ArticleController`. We're throwing a `NotFoundHttpException` if the `slug` wasn't found in the database. Pass `null` as the first argument to `createNotFoundException()` - that's just the message, not important for us. The second argument is a way to set the *previous* exception. Let's fake what the router does: `new NoConfigurationException()`.

77 lines | src/Controller/ArticleController.php

```
// ... lines 1 - 14
class ArticleController extends AbstractController
{
// ... lines 17 - 46
    public function show($slug, SlackClient $slack, ArticleRepository $articleRepository)
    {
// ... lines 49 - 50
        if (!$article) {
            throw $this->createNotFoundException('', new NoConfigurationException());
        }
// ... lines 54 - 61
    }
// ... lines 63 - 75
}
```

Testing time! Move over, click back to the real article show page... then change the slug to `foo`. Boom! Welcome to Symfony! I know, this is a silly example... but can you feel the power?

Back in the controller, remove that code.

Setting Response, Stop Propagation

Head over to `HttpKernel`. Symfony ultimately wants a `Response` object: it wants somebody to set the response on this event, which it fetches with `$response = $event->getResponse()`.

Hold Command or Ctrl and click to open the `ExceptionEvent` class. This class is similar to the `RequestEvent` that we saw earlier. If you find `handleRaw()`... here it is: `RequestEvent` is used for the *earliest* event in Symfony. Listeners to *that* event are able to set a `Response` if they want.

The same happens down in `handleThrowable()`: listeners can set a `Response` on the event. In fact, in `ExceptionEvent`, check out the base class! It's `RequestEvent`! It's the *exact* class the other event uses, and *it* holds the `setResponse()` method.

We already saw that method used in `RouterListener`: `$event->setResponse()` with `$this->createWelcomeResponse()`.

But what I *really* want to show you, back in `RequestEvent`, is this: if something calls `setResponse()`, the event class calls a `stopPropagation()` method. If you hold Command or Ctrl to jump to *that*, it opens in *another* base class. This method sets a flag called `propagationStopped` to true.

This is important: if you have multiple listeners to `kernel.exception` and one of them sets the `Response`, the other listeners will *not* be called. Yep, the EventDispatcher *looks* for this flag and, if it's true, it immediately stops calling the other listeners. This means that the *first* listener to set a response wins. It's a good thing to keep in mind, and it explains some of the *priorities* that the listeners have.

Next: let's look at the *critical* listeners to the `kernel.exception` event.

Chapter 16: The Critical kernel.exception Event Listeners

Back at the browser, refresh to get the normal not found page, click to open the profiler... and go into Events. Because this was a 404 page, the `kernel.exception` event *was* dispatched. The most important listener - and the one that *eventually* will render this page - is `ErrorListener`.

Let's see how it works! Hit Shift + Shift and open `ErrorListener.php`: get the one from `http-kernel/`, not `console/`. Look down here for the `getSubscribedEvents()` method. *Interesting*: it listens to `KernelEvents::CONTROLLER_ARGUMENTS` and it listens to `KernelEvents::EXCEPTION` twice. We won't look at the `CONTROLLER_ARGUMENTS` listener method - but if you want to look back at it after finishing the entire tutorial, it should make sense. What it does is minor, but interesting.

When the `kernel.exception` event is dispatched, `logKernelException()` will be called first and then, later, `onKernelException()`, because it has a `-128` priority.

How Exceptions are Logged

Find `logKernelException()` up on top. Its job is simple: log that an exception was thrown. If you follow the `logException()` logic, you'll see that it logs at a different *level* based on the status code. We're going to talk more soon about how different exceptions *get* different status codes. But the important piece here is that all 500 status code exceptions log at the `critical()` level, and 400 status code exceptions log at `error()`. If you're like us, you've probably *used* this fact before in your Monolog config to send 500 error logs to somewhere where you can be notified, like a Slack channel.

The Error Controller

The *other* listener method is `onKernelException()`. *This* is what's responsible for rendering the error page: both the nice development error page *and* the boring, production error page. It has a priority of `-128` because *it* will eventually set the `Response` on the event, which will *stop* event propagation. The low priority makes it easy to register *other* listeners before this happens. Heck, you could easily create a listener that *replaces* this one, by setting the Response itself... though, there are better ways to customize the error process.

Go find this method. Hmm. The first thing it does is reference some `$this->controller` property. Let's find out what that is. `dd($this->controller)`, then spin over to your browser, make sure you're on a 404 page and refresh.

151 lines | vendor/symfony/http-kernel/EventListener/ErrorListener.php

```
... lines 1 - 29
class ErrorListener implements EventSubscriberInterface
{
... lines 32 - 49
    public function onKernelException(ExceptionEvent $event, string $eventName = null, EventDispatcherInterface $event
    {
... lines 52 - 55
        dd($this->controller);
... lines 57 - 89
    }
... lines 91 - 149
}
```

Interesting: it's a *string*: `error_controller`. Find your terminal and run:

```
$ php bin/console debug:container error_controller
```

Surprise! `error_controller` is the id of a service! And its job apparently is to:

> Render error or exception pages from a given FlattenException

Ok, we don't know what a `FlattenException` is yet, but *apparently* this is a controller that's good at rendering error pages. Let's see what it looks like!

Hit Shift + Shift to open `ErrorController.php`. Ooooo. It has an `__invoke()` method! *This* is an *invokable* controller! We talked about those earlier when we were inside the controller resolver. *Usually* a controller will have the format `ClassName::methodName`. Well, we learned that this is *really* `ServiceId::methodName`.

Anyways, for an *invokable* controller - a controller class that has an `__invoke()` method - the syntax is simpler: just, `ServiceId`: no `::` stuff. *That* is what's happening here.

How the ErrorController is Called

Ok cool, so Symfony is going to execute this `error_controller` as a controller... and it will render the page. But... how? You can't normally just *call* a controller directly... or at least, you *shouldn't* do this.

Back in `ErrorListener`, take out the `dd()`. The logic here is *fascinating*. It says `$request = $this->duplicateRequest()` and passes the `$exception` and `$request` objects. Let's jump down to that method. Apparently, the `Request` class has a `duplicate()` method on it, which does exactly what you think - it effectively *clones* the object.

But, it passes this `$attributes` value to the third argument. This says:

> Please create an exact copy of this Request. When you do that, keep the same query parameters as the original, the same POST parameters as the original, but *replace* the original request *attributes* with this new array.

So... it's a *clone*, but with different request attributes. Most *importantly*, the new attributes have an `_controller` key set to that `error_controller` string.

Move back up to the `onKernelException()` method. We have a `Request` object that has an `_controller` request attribute. Here's the magic: `$response = $event->getKernel()->handle($request)`.

Yea! It's calling the `HttpKernel::handle()` method! The *same* one that we use in `index.php` and the *same* one we've been studying. *Inside* of handling the original request, it's handling a *second* request and getting back the response. And notice that it mentions something called a "sub request". We'll talk more about that soon.

For now, this is just a *super* fancy way of calling the `error_controller`. Instead of executing it directly, it creates a `Request` with an `_controller` attribute and tells `HttpKernel` to handle it. Neato!

Next, let's jump into `error_controller` itself and find out exactly *how* Symfony renders an error page. Because, it's a smart process: it renders the exception page in dev, the error page in prod and *even* changes format - like rendering JSON - when requested.

Chapter 17: FlattenException & Error Status Codes

The job of this ErrorController is to turn the Exception that was thrown into a Response . By the way, the error_controller is actually configurable. So if you want to control the error response on your site, you have two options so far. First, register a *listener* to kernel.exception . Or second, override the error controller via the framework.error_controller config.

But... if you did that, you would be responsible for rendering both the normal exception pages and production error pages. If you want to *change* how an error page looks, there are better ways. We'll see.

Inside the __invoke() method, the ErrorController ... is lazy! It *immediately* offloads the work to someone else - something called the errorRenderer . That returns some sort of exception... which apparently has getAsString() , getStatusCode() and getHeaders() methods. It uses these to create & return the Response .

The SerializerErrorRenderer

Let's... find out what this errorRenderer thing is: dd($this->errorRenderer) .

```
64 lines | vendor/symfony/http-kernel/Controller/ErrorController.php
... lines 1 - 25
class ErrorController
{
... lines 28 - 38
    public function __invoke(\Throwable $exception): Response
    {
        dd($this->errorRenderer);
... lines 42 - 44
    }
... lines 46 - 62
}
```

Move over and refresh. Ok cool: it's something called SerializerErrorRenderer . And actually, it only uses this class because this project has the *serializer* component installed. If you did *not*, this would be a different class - one that we'll see in a few minutes. And, by the way, this *whole* "error renderer" thing is part of a Symfony component called error-handler that's new in Symfony 4.4.

Let dig in! I'll close a class, then hit Shift + Shift to open SerializerErrorRenderer.php . Perfect!

The All-Important FlattenException

ErrorController calls this render() method, which *immediately* calls FlattenException::createFromThrowable . A FlattenException is basically a visual representation of an exception. And notice: the render() method *returns* a FlattenException .

Hold Command or Ctrl to jump into this class. Yea, see: it's not *actually* an exception - it doesn't extend Exception or implement Throwable . But it *does* contain a lot of the same info, like the exception $message , $code , $previous and the stack trace.

The FlattenException::createFromThrowable - if we jump to that - is a way to easily create this "visual representation" based on a real exception. And *this* contains some pretty important stuff. For example, if $exception is an instance of HttpExceptionInterface , then it calls $exception->getStatusCode() to get the status code and $exception->getHeaders() to get the headers. Both the status code and headers are *ultimately* stored on this FlattenException object and *used* by ErrorController when it creates the final Response .

Why do Some Exceptions Cause Different Status Codes?

So... what *is* this HttpExceptionInterface thing? We've actually seen it. Go back to ArticleController . We know that $this->createNotFoundException() is a shortcut to instantiate a new NotFoundHttpException . Click to open that class... and click again to open its *base* class `HttpException`. Here it is: `HttpException` implements `HttpExceptionInterface`.

This is a *long* way of showing you that certain exception classes in Symfony - like `NotFoundHttpException` - map to a *specific* status code. This works because they implement `HttpExceptionInterface` and because `FlattenException` uses this.

Why does `NotFoundHttpException` *specifically* map to a 404. It calls `parent::__construct()` with 404... that is set to a `$statusCode` property... and then returned from `getStatusCode()`. You can *also* pass custom `$headers` to the exception.

And there are a bunch of other exception classes like this. I'll double-click on the `Exception` directory at the top of PhpStorm. Wow! There are more than 15 in this directory alone, like `BadRequestsHttpException`, which will give you a 400 status code, `PreconditionFailedHttpException`, which will be a 412 and many more. Hmm, where's the `IAmATeaPotHttpException`?

If you throw any of these exceptions from *anywhere* in your app, they will trigger an error page with the correct status code. This is a *powerful* thing to understand.

Back in `FlattenException`, there is also another type of exception interface called `RequestExceptionInterface`. It's not as important and it always maps to a 400 status code.

If the exception doesn't implement either of these interfaces, the status code will be 500.

These are the *most* important parts of the `FlattenException`. Close it and go back to `SerializerErrorRenderer`. The job of this method is to create a `FlattenException` object *from* the exception and make sure it contains three things that the `ErrorController` needs: the status code, headers and a *string representation* of the error, which will become the *body* of the response. We've got the status code & headers... but we *still* need to somehow generate a "string" representation of this exception. Let's see how that's done next.

Chapter 18: Serializer Error Renderer: JSON/XML Errors

This method is called by `ErrorController` and its job is to return a `FlattenException` that contains the status code, headers and *body* that should be set on the final error `Response`. The `FlattenException::createFromThrowable` intelligently sets the status code and headers. But we *still* need to, somehow, figure out what *content* to send back, like a JSON error, or an HTML page that says: "Please send help!".

Determining the Preferred Format

To do that, `SerializerErrorRenderer` *first* tries to figure out what *format* the user wants - like HTML or JSON. The `$this->format` property is actually a callback that points down here to this `getPreferredFormat()` method. This is a fancy way of getting the request and calling `$request->getPreferredFormat()`. And... hey! We know that method! I'll hit Shift + Shift and open `Response.php` from `HttpFoundation`. Search for `prepare()`. This method is called by a listener to the `kernel.response` event. It normalizes a few things... *including* setting the `Content-Type` header if it hasn't *already* been set. To help with that, it calls `$request->getPreferredFormat()` to try to figure out if the user wants HTML, JSON or something else. One of the ways it figures this out is by looking at the `Accept` header on the request.

Back in `SerializerErrorRenderer`, we're once again using `$request->getPreferredFormat()`, which will return a simple string like `html` or `json`.

Serializing the Error to JSON, XML, etc

Up in `render()`, this is pretty cool: it says:

> Hey serializer! Can you try to serialize the `FlattenException` object into this format?

If the `format` is `html`, this will fail with a `NotEncodableValueException`: the serializer doesn't handle HTML. We'll talk about that case in a minute. But if the format is `json`, `xml` or some other format that the serializer *does* support, this will convert the exception to *that* format!

We can see this. If we refresh the page... we see the big HTML exception - and we'll see the code that makes this soon. Copy the URL, find your terminal and use `curl` to fetch that URL. But *also* pass a `-H` flag to add a header: `"Accept: application/json"`:

```
$ curl https://localhost:8000/news/foo -H "Accept: application/json"
```

This will change the "preferred format" on the request to `json`. And... check it out! It's a 404 status code but in a JSON format! We can even use `text/xml` to see this in XML.

```
$ curl https://localhost:8000/news/foo -H "Accept: text/xml"
```

How is a FlattenException Serialized?

How does this work? One of the normalizers in the serializer is called `ProblemNormalizer`. I'll hit Shift + Shift to open it: `ProblemNormalizer.php`.

If you don't know much about the serializer component, the important thing to know is that a normalizer is responsible for taking an object and *converting* it into an *array* of data. Thanks to the `supportsNormalization()` method, *this* class is used when you try to normalize a `FlattenException` object.

This normalizer creates a response format that follows an HTTP specification: it helps us return an official, standardized error response. It's pretty simple: it sets keys for `type`, `title`, `status` and `detail`. In `$debug` mode, it *also* adds `class` and `trace`. Also, the `detail` key in debug mode will be the exception message... but in production, it will be the "status text", which is a generic "Not Found" message... or something similar, based on the status code. That's done so that your exception messages don't "leak" to the public.

The `normalize()` method is passed the `$exception`, which is the `FlattenException`. But if you look back at `SerializerErrorRenderer`, it *also* passes the *original* exception as an `exception` key on the `$context` - that's the 3rd argument to `normalize()`.

So this gives us a really nice error response body, without any work. If you wanted to *change* this data, you could do that by adding your *own* custom normalizer. We actually talk about this in our API Platform Security Tutorial. You could *decorate* the `ProblemNormalizer`... and maybe just add or tweak some data *or* you could create an entirely *new* normalizer. Heck, you could use the `$context` in `supports` - you need to implement `ContextAwareNormalizerInterface` to make that work - and make that *new* normalizer responsible for *only* normalizing `FlattenException` classes for a *specific*, *original* exception. If you want to try that and have problems, let us know.

Ok, close that class up. Next, let's find out what happens if the format is *not* something that the serializer can handle. Like, HTML.

Chapter 19: How the HTML Error Page is Rendered

When you use a browser, the format will be `html` . That's also the *default* format if no request format was set and if the request doesn't contain an `Accept` header. In the case of `html` , the serializer will fail by throwing a `NotEncodableValueException` . When that happens, this offloads the work to *another* error render: `$this->fallbackErrorRenderer` .

If you dumped this object, you'd find out that it's an instance of `TwigErrorRenderer` . Ok! Let's open that up: Shift + Shift `TwigErrorRender.php` .

It... interesting! It immediately calls *another* `fallbackErrorRenderer` . This one is an instance of `HtmlErrorRender` . Open it up: `HtmlErrorRenderer.php` .

Error Renderer Decoration

Then... stop. Let me explain *why* and *how* we have *three* different error renderer classes. This `HtmlErrorRenderer` is the, sort of, "core" error renderer and it *always* exists. But *if* you have Twig installed, the `TwigErrorRenderer` suddenly "takes over" the error rendering process. It does that via service *decoration*: `TwigErrorRenderer` *decorates* `HtmlErrorRenderer` .

And then... if you have the serializer component installed, suddenly there is a *third* error renderer added to the system: `SerializerErrorRenderer` , which decorates `TwigErrorRenderer` .

This is a *slight* over-simplification, but there is basically only ever *one* official "error renderer" service registered in the container. It's `id` is `error_renderer` . But through service decoration, multiple error renderers are ultimately used.

HtmlErrorRenderer: Default Exception & Error Templates

Let's look at the flow. `TwigErrorRender` calls `render()` on `HtmlErrorRenderer` . Remember: the `render()` method on *all* of these classes has the same job: to return a `FlattenException` that contains the status code, headers and the "body" that will be used for the Response.

So, it's no surprise that this *once* again starts by creating a `FlattenException` object. To get the "body" of the response, it calls `$this->renderException()` . Jump to that.

This is what builds the error or exception page. The `$debugTemplate` argument defaults to `views/exception_full.html.php` . Yea, this method render PHP templates! This template will be used in `debug` mode. If we're *not* in debug mode, then it "includes" - basically, renders - `error.html.php` . So `exception_full.html.php` in debug mode, `error.html.php` on production. The `include()` function is as simple as it gets.

Let's go *see* the debug template. Using the directory tree on top... click the `error-handler/` directory, then navigate to open `Resources/views/exception_full.html.php`

This is what we're seeing in our browser right now. To prove it, in the middle, let's add:

> I'm inside your exception page!

45 lines | vendor/symfony/error-handler/Resources/views/exception_full.html.php

```
... lines 1 - 2
<html lang="en">
... lines 4 - 12
    <body>
... lines 14 - 35
        I'm inside your exception page!
... lines 37 - 41
    </body>
</html>
... lines 44 - 45
```

Back on the browser, refresh the 404 page. There's our text! Go... take that out.

So this template and `error.html.php` are responsible for rendering the debug and production HTML error pages out-of-the-box.

Close `exception_full.html.php` ... and also `HtmlErrorRenderer.php` .

TwigErrorRenderer: Twig Overrides

Back in `TwigErrorRenderer` , this starts by getting the `FlattenException` from `HtmlErrorRenderer` . So then... if we *already* have the finished `FlattenException` , what's the point of *this* class?

This *entire* class exists to give *you* - the application developer - the ability to *override* what the error template looks like. `$this->findTemplate()` is used to check if you have a Twig *override* template. If you don't, the `FlattenException` from `HtmlErrorRenderer` is used. But if you *do* have an override template, it renders that and uses *its* HTML.

Twig Namespaces & Override Templates

Scroll down to the `findTemplate()` method. Cool! It first looks for a template called `@Twig/Exception/error%s.html.twig` , where the `%s` part is the *status* code. The `@Twig` thing is a Twig *namespace*. Every bundle in your app automatically has one. Want to render a template from `FooBarBundle` ? You could do that by saying `@FooBar` then the path to the template from within that bundle.

This is *normally* used as a way for a bundle to render a template *inside* itself. But Symfony *also* registers an *override* path for every bundle namespace. When you say `@Twig/Exception/error404.html.twig` , Twig *first* looks for the template at `templates/bundles/TwigBundle/Exception/error404.html.twig` .

Anyways, if this template exists because you created it, it will be used. Otherwise, it looks for a generic `error.html.twig` that handles all status codes. *This* is how the Twig error template overrides work.

And... phew! That's it! `SerializerErrorRenderer` renders XML & JSON pages, or, really, anything format that the serializer supports. `HtmlErrorRenderer` renders the HTML pages and `TwigErrorRenderer` allows you to override that with carefully-placed Twig templates.

Finishing the Process

Close both of the error renderers. We *now* know that there are *many* ways to hook into the exception-handling process. You can override `ErrorController` , listen to the `kernel.exception` event, customize the `ProblemNormalizer` for JSON or XML exceptions *or* add a Twig template override for custom HTML.

No matter what, `ErrorListener` sets this `Response` onto the `ExceptionEvent` . In `HttpKernel` , if the event has a response, there's a bit of final status code normalization, but it eventually passes the `Response` to `filterResponse()` . So yes, even an error page will trigger that event, which is why a 404 page *has* the web debug toolbar.

Ok team, we're now *truly* done walking through the HttpKernel process: both the happy and unhappy paths. Next, let's use our new knowledge... to start hacking into the system.

Chapter 20: The Magic `_controller` Attribute

Now that we've been stuffed full of knowledge about the request-response process, let's see what kind of trouble we can get into. Uh, I mean, let's do some cool and productive things with all this new information!

Overriding the Controller from a Listener?

Close all the files except for `index.php` and `HttpKernel`. Here's our first challenge: could we - from some event listener - *change* the controller for the page?

Hint: in Symfony, the answer to "can I do X" is *always* yes. In this case, it's not only possible, there are *multiple* ways to do it.

For example, when `HttpKernel` dispatches the `kernel.controller` event, it passes each listener a `ControllerEvent` object. And that class has a `setController()` method. Easy peasy! We can override the controller by adding a listener to that event. Heck, you can do the same thing down here with the `kernel.controller_arguments` event.

Overriding the Controller on kernel.request?

So... that was too easy. I'll close up my directory tree... and then open our `UserAgentSubscriber`. Here's my harder challenge: how can we override the controller from *here*: from a listener to the `kernel.request` event. In this case, there is no `setController()` method.

Callback Controller with `_controller`

The trick is to remember how the controller resolver works: it starts by fetching the `_controller` value from the `$request->attributes`. So if, for *some* reason, we wanted to completely replace the controller, we can do it right here: `$request->attributes->set('_controller', ...`. For fun, let's set this to an anonymous function... cause yea! That's allowed! Inside, return a `new Response()` with:

> I just took over the controller

38 lines | src/EventListener/UserAgentSubscriber.php

```php
... lines 1 - 9
class UserAgentSubscriber implements EventSubscriberInterface
{
... lines 12 - 18
    public function onKernelRequest(RequestEvent $event)
    {
... lines 21 - 22
        $request->attributes->set('_controller', function() {
            return new Response('I just took over the controller!');
        });
... lines 26 - 28
    }
... lines 30 - 36
}
```

Will it work? Refresh *any* page. Yep! We see our message here, on the homepage and *everywhere*. And our normal controller tricks work just fine: add a `$slug` argument... but give it a default value and then `dd($slug)`. On the article show page... this works thanks to the `{slug}` wildcard. On the homepage, it's `null` because that wildcard doesn't exist.

41 lines | src/EventListener/UserAgentSubscriber.php

```
// ... lines 1 - 23
        $request->attributes->set('_controller', function($slug = null) {
            dd($slug);
// ... lines 26 - 27
        });
// ... lines 29 - 41
```

RouterListener Skips when _controller is Set

Open up RouterListener.php one more time and find its onKernelRequest() method. *This* method is *normally* responsible for executing the router and setting the _controller key onto the request attributes. But back down at getSubscribedEvents(), ah! The kernel.request listener has a priority of 32. We didn't give our subscriber a priority, so it has a priority of 0. This means that RouterListener is called *before* our subscriber.

Ok, so then here is what's happening: RouterListener is called first and it *is* executing the router and setting an _controller key on the request attributes. *Then* our listener is called and we *override* that value.

So... if we *reversed* the order - if we made *our* listener be called first - our little _controller hack would *not* work, right? Because RouterListener would override *our* value.

Actually... no! At the top of onKernelRequest(), one of the *first* things it does is check to see if something has *already* set the _controller attribute. If it has, it does nothing: someone *else* has decided to be responsible for figuring out which controller to call. In reality, no matter *how* early the _controller attribute is set, it will *always* win over RouterListener.

Peaking at our First Sub-Request

Why is that important? Because *this* explains how ErrorListener was able to execute ErrorController. Open up ErrorListener.php. Remember: to execute ErrorController this *duplicated* the request. But it didn't create an exact copy: it overrode the attributes in order to set _controller to error_controller. Then it sent that new Request back through the *entire* $kernel->handle() process! This means that before *any* listeners were executed during that *second* trip through HttpKernel::handle(), the _controller attribute was already set.

So in reality, on an error page, RouterListener is called *two* times: once for the main request... when it does its job normally... and *again* for the "sub request". That second time, because the _controller attribute is already set, RouterListener does nothing.

In fact, let's see this. Before the if, dump($request->attributes->has('_controller')). Then, in your browser, go back to a 404 and try it.

177 lines | vendor/symfony/http-kernel/EventListener/RouterListener.php

```
// ... lines 1 - 42
class RouterListener implements EventSubscriberInterface
{
// ... lines 45 - 96
    public function onKernelRequest(RequestEvent $event)
    {
// ... lines 99 - 102
        dump($request->attributes->has('_controller'));
// ... lines 104 - 142
    }
// ... lines 144 - 175
}
```

Ah, boo! This hit the die statement in our fake controller. I didn't mean to do that. In UserAgentSubscriber, comment-out our controller hack so we can see the whole process.

43 lines | src/EventListener/UserAgentSubscriber.php

```
... lines 1 - 10
class UserAgentSubscriber implements EventSubscriberInterface
{
... lines 13 - 19
    public function onKernelRequest(RequestEvent $event)
    {
... lines 22 - 23
        /*
        $request->attributes->set('_controller', function($slug = null) {
... lines 26 - 28
        });
        */
... lines 31 - 33
    }
... lines 35 - 41
}
```

Ok, try it again. Hello 404 page! Hover over the target icon on the web debug toolbar. Yes! 2 dumps from `RouterListener`: `false` the first time it's called and, the second time it's called - which is due to the code in `ErrorListener` - it dumps `true` because that `Request` *does* already have the `_controller` attribute.

This second request is called a sub-request... but more on that topic later. Remove the `dump()` call.

Let's see what other ways we can hack into Symfony, like by adding *new* things that can be used as controller arguments. That's next.

Chapter 21: Custom Global Controller Arguments

Now that we understand a lot more about its flow, we're on a mission to find weird, crazy things that we can do in Symfony. For the next one, pretend that, for *some* reason, we need to know whether or not a visitor is using a Mac or not. In fact, we need this info *so* often, that we want the ability to add an `$isMac` argument to any controller, like this.

Let's `dump($isMac)` ... and then try it. No surprise, it explodes!

77 lines | src/Controller/ArticleController.php

```php
... lines 1 - 13
class ArticleController extends AbstractController
{
... lines 16 - 45
    public function show($slug, SlackClient $slack, ArticleRepository $articleRepository, $isMac)
    {
        dump($isMac);
... lines 49 - 61
    }
... lines 63 - 75
}
```

> Controller `show()` requires that you provide a value for the `$isMac` argument.

I'll go back to a real article page, though that won't make any difference.

Custom Arguments Via Request Attributes

So: how can we make this work? There are actually *two* answers, and we're going to try both. The first is a, kind of, lower-level way of doing it. We know that if we have a `{slug}` route wildcard, we are allowed to have a `$slug` argument. So, in theory, if we had an `{isMac}` wildcard, we could have an `$isMac` argument, though that's not what we want.

But it's not *really* that we're allowed to have a `$slug` argument because there's a `{slug}` *wildcard*. Nope, we're allowed to have a `$slug` argument because there is a `slug` key in the `$request->attributes`. The router *puts* `slug` into attributes *because* of the wildcard, but when it comes to figuring out what arguments to pass to a controller, it's all about the `$request->attributes`.

Inside of our listener, let's say `$isMac = stripos($userAgent, 'Mac') !== false`. Now, to make `isMac` available as an argument to any controller, add `$request->attributes->set('isMac', $isMac)`.

45 lines | src/EventListener/UserAgentSubscriber.php

```php
... lines 1 - 10
class UserAgentSubscriber implements EventSubscriberInterface
{
... lines 13 - 19
    public function onKernelRequest(RequestEvent $event)
    {
... lines 22 - 33
        $isMac = stripos($userAgent, 'Mac') !== false;
        $request->attributes->set('isMac', $isMac);
    }
... lines 37 - 43
}
```

And... that's it! Try the page now. It works! And for me, it's set to true.

Custom ArgumentValueResolver

The *second* way to add a custom controller argument is a bit more direct: create a custom `ArgumentValueResolver`. When we were deep-diving into how Symfony determines what arguments to pass to a controller, we found out that there are various classes that determine this called "argument value resolvers". And we can create our *own*.

Inside of the `src/` directory - it doesn't matter, let's put it in `Service/` - create a new class called: `IsMacArgumentValueResolver`. The only rule is that this class must implement `ArgumentValueResolveInterface`. I'll go to the Code -> Generate menu - or Command + N on a Mac - and select "Implement Methods" to generate the two methods that we need.

21 lines | src/Service/IsMacArgumentValueResolver.php

```
... lines 1 - 4
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\Controller\ArgumentValueResolverInterface;
use Symfony\Component\HttpKernel\ControllerMetadata\ArgumentMetadata;

class IsMacArgumentValueResolver implements ArgumentValueResolverInterface
{
    public function supports(Request $request, ArgumentMetadata $argument)
    {
        // TODO: Implement supports() method.
    }

    public function resolve(Request $request, ArgumentMetadata $argument)
    {
        // TODO: Implement resolve() method.
    }
}
```

Without doing anything else, this class is *already* being used by Symfony as an argument value resolver. When we talked about that system, I hinted that the way you get an argument value resolver into the system is by creating a service and *tagging* it with `controller.argument_value_resolver`. Find your terminal and, once again, run:

```
$ php bin/console debug:container --tag=controller.argument_value_resolver
```

And now... if you look at the service ids, one of them is for our `App\Service\IsMacArgumentValueResolver`. It's wrapped in *another* class because Symfony is decorating the services with `TraceableValueResolver`, but this *is* our service being used. Our new service *already* has the tag thanks to Symfony's auto-configuration feature.

Filling in the ArgumentValueResolver Logic

Let's go fill in the logic. Here's the plan: very simply, if the argument's name *exactly* matches `$isMac`, we'll fill in our value. So for `supports()`, return `$argument->getName() === 'isMac'`.

23 lines | src/Service/IsMacArgumentValueResolver.php

```
... lines 1 - 10
    public function supports(Request $request, ArgumentMetadata $argument)
    {
        return $argument->getName() === 'isMac';
    }
... lines 15 - 23
```

For `resolve()`, go grab the `$userAgent` code from the subscriber, paste it, and then also copy the `stripos()` logic. Delete the last two lines from the subscriber so that it stops setting this global argument.

23 lines | src/Service/IsMacArgumentValueResolver.php

```
// ... lines 1 - 15
    public function resolve(Request $request, ArgumentMetadata $argument)
    {
        $userAgent = $request->headers->get('User-Agent');
// ... lines 19 - 20
    }
// ... lines 22 - 23
```

Finish up the resolver by saying `return stripos($userAgent, 'Mac') !== false`.

23 lines | src/Service/IsMacArgumentValueResolver.php

```
// ... lines 1 - 15
    public function resolve(Request $request, ArgumentMetadata $argument)
    {
// ... lines 18 - 19
        yield stripos($userAgent, 'Mac') !== false;
    }
// ... lines 22 - 23
```

Let's try it! Find your browser, refresh and.. boo!

> Can use "yield from" only with arrays and Traversables

That's a funny way of saying that I forgot to `yield` instead of `return` from this method: `resolve()` returns a *traversable*. Try it now and... it works! We still see `true` for the dump.

Next, let's uncover one *last* mystery about controller arguments. Back in `ArticleController::show()`, we *originally* had an `$article` argument that was type-hinted with an `Article` entity class. How did that work? Who was making that automatic query for us?

Chapter 22: How Entity Controller Arguments Work

In `ArticleController::show()` we're using the `$slug` argument to manually query for the `Article` object and triggering a 404 page if needed. But *originally* this code looked different: instead of a `$slug` argument, we had an `$article` argument type-hinted with the `Article` entity class. We didn't need to make the query because something else was doing it for us. The same is true for the 404 logic.

I love this feature! But the question for *us* is: how does this work? What is *allowing* us to have this `$article` argument. *Something* is *noticing* that we're type-hinting an argument with an entity class and is automatically querying for it based on the `{slug}` wildcard. But where is that code?

At first, you might think this is another argument value resolver. But, there's nothing in that list that mentions "doctrine" or "entity". In reality, this is working via a *different* system.

If we refresh now, the page still works. But if you change the URL to a slug that *won't* be found, the error gives us a hint:

> `App\Entity\Article` object not found by the `@ParamConverter` annotation

ParamConverterListener

This error is coming from a class called `DoctrineParamConverter`, which if you look further down the stack trace, is called by some `ParamConverterListener`.

Let's start by checking out that class. I'll hit Shift + Shift to open `ParamConverterListener.php`. Ah - this class comes from `SensioFrameworkExtraBundle`. The first thing I want you to notice is that this implements `EventSubscriberInterface`. Yep! This is an *event* listener and it listens to the `kernel.controller` event: the event that's dispatched *after* the controller is determined, but *before* the controller is *called*. And also before the arguments are determined.

That makes sense! If this class is going to do some magic on the arguments to our controller, it's going to need to know which controller is about to be called so it can *look* at its arguments.

We're not going to study the details of this class *too* closely, but we can the basic idea pretty easily. This function loops over all of the parameters - all of the arguments of the controller. That `$param` is a `ReflectionParameter` which holds info about that argument. Most importantly, it knows what *class* the argument is type-hinted with.

DoctrineParamConverter

Anyways, that method collects info about each argument and then, eventually - if you look at the stacktrace - it executes something called `DoctrineParamConverter`. That's where the magic happens. I'll hit Shift + Shift to open that file: `DoctrineParamConverter.php`.

Hmm. This starts by getting the `$class`: that's the type-hint on the argument. Then, it tries different ways of querying for that. For example, `$this->find()` tries to use the `id`: it tries to see if the primary key is a wildcard in the URL, gets the entity manager for that class and ultimately tries to call a method to make that query.

This feature has a lot of options and can query for your entity in a lot of different ways. But we're not going to get into the weeds about all of that now.

The *really* cool part is that - no matter *how* it finds your entity - if we follow the logic to the bottom of `apply()`, *eventually* it takes that `Article` object and sets it onto the `$request->attributes`! The `$name` variable is the *name* of the argument - `article` for us - and `$object` will be the full entity that was fetched from the database.

SensioFrameworkExtraBundle is *full* of magic like this and *all* that magic works via *listeners*. If you want to know more about how one of its features works, find the listener that it's using. Oh, and if you're wondering why this `DoctrineParamConverter` *doesn't* just use the "argument value resolver" system, the answer is that it *pre-dates* it. It may, some day, be converted to use it.

Next, let's start talking about a *fascinating* topic that we've already seen a few times. I want to talk about sub requests.

Chapter 23: Sub Requests

Before we finish our adventure, I want to talk about a *fascinating* feature of the request-response process. It's something that we've already seen... but not explored. I want to talk about sub-requests.

Rendering a Controller from a Template

To do that, we need to add a feature! On the homepage, see these trending quotes on the right? I'm going to close a few files... and open this template: `templates/article/homepage.html.twig` . The trending quotes are hardcoded right here. Let's make this a bit more realistic: let's pretend that these quotes are coming from the database.

That would be simple enough: we could open the homepage controller, query for the quotes and pass them into the template. Except... I'm going to complicate things. Pretend that we want to be able to easily reuse this "trending quotes" sidebar on a *bunch* of different pages. To do that nicely, we need to somehow encapsulate the markup *and* the query logic.

There are at least 2 different ways to do this. The first option would be to move the markup to another template and, inside that template, call a custom Twig function that fetches the trending quotes from the database.

The *second* option - and a *particularly* interesting one if you want to use HTTP caching - is to use a sub-request. You may have done this before without realizing that you were *actually* doing something *super* cool.

Remove this entire section and replace it with `{{ render(controller()) }}` . Together, these two functions allow you to *literally* render a controller from inside Twig. The content of that Response will be printed right here.

Let's execute a new controller: `App\\Controller` - you need 2 slashes because we're inside a string - `\\PartialController` . For the method, how about, `::trendingQuotes` .

58 lines | templates/article/homepage.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
... lines 6 - 45
            <div class="col-sm-12 col-md-4 text-center">
... lines 47 - 52
                {{ render(controller('App\\Controller\\PartialController::trendingQuotes')) }}
            </div>
        </div>
    </div>
{% endblock %}
```

Creating the Sub-Request Controller

Cool! Let's go make that! Click on `Controller/` and create a new PHP class: `PartialController` . Make it extend the usual `AbstractController` and create the `public function trendingQuotes()` .

39 lines | src/Controller/PartialController.php

```
... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class PartialController extends AbstractController
{
    public function trendingQuotes()
    {
... lines 11 - 15
    }
... lines 17 - 37
}
```

But instead of making a real database query, let's fake it. I'll paste in a new private function called `getTrendingQuotes()`: it returns an array with the data for the 3 quotes.

39 lines | src/Controller/PartialController.php

```
... lines 1 - 17
    private function getTrendingQuotes()
    {
        return [
            [
                'author' => 'Wernher von Braun, Rocket Engineer',
                'link' => 'https://en.wikipedia.org/wiki/Wernher_von_Braun',
                'quote' => 'Our two greatest problems are gravity and paperwork. We can lick gravity, but sometimes the paper
            ],
            [
                'author' => 'Aaron Cohen, NASA Administrator',
                'link' => 'https://en.wikipedia.org/wiki/Aaron_Cohen_(Deputy_NASA_administrator)',
                'quote' => 'Let\'s face it, space is a risky business. I always considered every launch a barely controlled explosio
            ],
            [
                'author' => 'Christa McAuliffe, Challenger Astronaut',
                'link' => 'https://en.wikipedia.org/wiki/Christa_McAuliffe',
                'quote' => 'If offered a seat on a rocket ship, don\'t ask what seat. Just get on.',
            ],
        ];
    }
... lines 38 - 39
```

Above, call this: `$quotes = $this->getTrendingQuotes()`... and then render the template: `return $this->render()`, `partial/trendingQuotes.html.twig` passing in the `$quotes` variable.

39 lines | src/Controller/PartialController.php

```
... lines 1 - 8
    public function trendingQuotes()
    {
        $quotes = $this->getTrendingQuotes();

        return $this->render('partial/trendingQuotes.html.twig', [
            'quotes' => $quotes
        ]);
    }
... lines 17 - 39
```

Finally add the template: create the new `partial/` directory first... then the new `trendingQuotes.html.twig` inside. Perfect! I'll paste some code here that loops over the quotes and prints them. Remember that you can get *any* of the code I'm pasting from the code blocks on this page.

11 lines | templates/partial/trendingQuotes.html.twig

```
<div class="quote-space pb-2 pt-2">
  <h3 class="text-center pb-3">Trending Quotes</h3>
  <div class="px-5">
    {% for quote in quotes %}
      <p{{ loop.first ? '' : 'class="pt-4"' }}>
        <i class="fa fa-comment"></i> "{{ quote.quote }}" <br>— <a href="{{ quote.link }}">{{ quote.author }}</
      </p>
    {% endfor %}
  </div>
</div>
```

Ok! Let's see if it works! Move over and refresh. Woo! That was amazing! We just made a sub-request!

Seeing the Sub Request in the Profiler

Oh... you're not as excited as I am? Ok fine. Click any of the icons down on the web debug toolbar to open the profiler and then go to the Performance section. Look closely: it has all the normal stuff right? I see `RouterListener` and our controller. But, there's a funny shaded background coming from inside the Twig template.

This is indicating that there was a sub-request during this time. And if you scroll down, you can see it! Sub-requests 1 for `trendingQuotes()`.

This will make more sense if you scroll up and set the Threshold input box back down to 0 to show everything.

Look again at the shaded area. This is when the sub-request is being handled, which *literally* means that another `Request` object was created and sent into `HttpKernel::handle()`! Scroll down... and behold!

Symfony didn't just "call" the controller: it went through the *entire* `HttpKernel::handle()` process again! It dispatched another `kernel.request` event, executed all the listeners - including our `UserAgentSubscriber` - called the controller and dispatched `kernel.response`. It also dispatched the other normal events too - they're just hard to see.

So... yea! `{{ render(controller()) }}` sends a second `Request` object through the `HttpKernel` process. It's bonkers.

In fact, that second request even gets its own entire profiler! Yep, click the controller link to go to the *profiler* for *that* sub-request! Check out the URL: this is a kind of, internal URL that identifies this sub-request. Set the threshold to 0 here to get a *big* view of that sub-request.

Sub-Requests & _controller

So... how did this work? How does Symfony go through the entire HttpKernel process and render this controller... if there is no *route* to the controller? How does the routing work for a sub-request?

The truth is: the routing *doesn't* execute. Click into the "Request / Response" section and scroll down to the request attributes. Check it out: the request attributes have an `_controller` key set to `App\Controller\PartialController::trendingQuotes`.

This works a lot like what we saw in `ErrorListener`, when it rendered `ErrorController`. Symfony created a `Request` object and set the `_controller` on its attributes. Then, when `RouterListener` was called for this sub-request - because it *was* called - it saw that the request already had an `_controller` attribute and returned immediately. The router is never called and the `ControllerResolver` reads the `_controller` string that was originally set.

Sub-Requests are Expensive

So *this* is a sub request. We're going to explore it further and talk about some special properties of it. But before we do, I want to mention one thing. Sub-Requests are *awesome* if you want to leverage HTTP caching: when you *cache* the `Response` of a sub-request. For example, you could cache the `trendingQuotes()` `Response` for 1 hour, and then *not* cache the rest of the page at all. Or you could do the opposite! It's a *blazingly* fast way to cache.

But if you're *not* using HTTP caching, be careful not to over-use sub-requests. Remember: they execute an *entire* `HttpKernel::handle()` flow. So if you have a lot of them, it will slow down performance.

Next: let's make our sub-request a little bit more interesting. It will uncover something mysterious.

Chapter 24: Sub Requests & Request Data

Remember that cool trick that we did a few minutes ago with the argument value resolver that allowed us to have an `$isMac` argument to any controller in our system? Does that *also* work in a controller that's called by sub request? Of course! Because there's nothing special about this controller: it was called thanks to a complete cycle through `HttpKernel::handle()`. All the same listeners and all the same argument value resolvers are called.

So... cool! Let's use that! Add an `$isMac` argument... then pass it into the template.

40 lines | src/Controller/PartialController.php

```
... lines 1 - 6
class PartialController extends AbstractController
{
    public function trendingQuotes($isMac)
    {
... lines 11 - 12
        return $this->render('partial/trendingQuotes.html.twig', [
... line 14
            'isMac' => $isMac
        ]);
    }
... lines 18 - 38
}
```

Inside `trendingQuotes.html.twig`, near the bottom, add `{% if isMac %}{% endif %}` and inside, put an `<hr>`, a `<small>` tag, and then say:

16 lines | templates/partial/trendingQuotes.html.twig

```
<div class="quote-space pb-2 pt-2">
... lines 2 - 10
    {% if isMac %}
        <hr>
        <small>BTW, you're using a Mac!</small>
    {% endif %}
</div>
```

> BTW, you're using a Mac!

Easy enough! Find your browser... navigate back to the homepage... and refresh. On the right, there it is! We're using a Mac.

Just for the heck of it, let's add that same logic to the sidebar above this. This lives in the homepage template, so find `ArticleController::homepage`... add an `$isMac` argument and pass *this* into the template.

73 lines | src/Controller/ArticleController.php

```
... lines 1 - 13
class ArticleController extends AbstractController
{
... lines 16 - 32
    public function homepage(ArticleRepository $repository, LoggerInterface $logger, $isMac)
    {
... lines 35 - 37
        return $this->render('article/homepage.html.twig', [
... line 39
            'isMac' => $isMac,
        ]);
    }
... lines 43 - 71
}
```

Steal the `isMac` logic from the trending quotes template, open `homepage.html.twig` and... right below the "Buy Now!" button, paste.

63 lines | templates/article/homepage.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
... lines 6 - 45
            <div class="col-sm-12 col-md-4 text-center">
                <div class="ad-space mx-auto mt-1 pb-2 pt-2">
... lines 48 - 51
                    {% if isMac %}
                        <hr>
                        <small>BTW, you're using a Mac!</small>
                    {% endif %}
                </div>
... lines 57 - 58
            </div>
        </div>
    </div>
{% endblock %}
```

When we try the page now, no surprise: *both* places show the message.

Adding the ?mac Override

Since I *am* using a Mac, it's kind of hard to test whether or not this feature correctly hides for people who are *not* on a Mac. To make testing easier, let's add a way to override the real logic. I want to be able to add a `?mac=true` or `?mac=false` to the URL to have full control.

The code for setting the argument is in `IsMacArgumentValueResolver`. So, if we want to "short-circuit" the real logic, it's no problem. Before we read the `User-Agent`, add if `$request->query->has('mac')`, then `yield $request->query->getBoolean('mac')`. `getBoolean()` is a cool function that grabs the `mac` query parameter but runs it through PHP's `filter_var()` function with the `FILTER_VALIDATE_BOOLEAN` flag. That means a value like a `false` string will turn into a `false` boolean. Kinda fun. Anyways, after this, return so the function doesn't continue.

```
29 lines | src/Service/IsMacArgumentValueResolver.php
... lines 1 - 8
class IsMacArgumentValueResolver implements ArgumentValueResolverInterface
{
... lines 11 - 15
    public function resolve(Request $request, ArgumentMetadata $argument)
    {
        if ($request->query->has('mac')) {
            yield $request->query->getBoolean('mac');

            return;
        }
... lines 23 - 26
    }
}
```

Ok: if I refresh without changing the URL, it still reads my `User-Agent` and everything looks right. Now add `?mac=false`. And... it works! The message is gone. Oh wait! The *first* message is gone, but the one coming from the sub-request controller is still there! What the heck?

If you're thinking that somehow the argument value resolver isn't called on a sub request, that's not it. A sub request is handled *exactly* like the main request. This function is being called *twice* on this page: once for the main request and again for the sub request. So why do those two calls produce a different result?

The Request in the Sub Request is not the Same

Click into the profiler and go to the Performance section. The `Request` object that's being processed on top is *not* the same as the `Request` object that's being processed down here for the sub-request. Symfony creates two, *distinct* `Request` objects. The first `Request` object represents the data for the *real* HTTP request that's coming into our app. And so, it contains the query parameter info. But that *second* `Request` is kind of a "fake" request. It mainly exists so that the `_controller` attribute can be set on it. It's not *really* a representation of the "real" request. And so, it may not have all the same data. It doesn't have the query parameters, for example.

Let's see this: `dump($request)` inside of the `resolve()` method... then refresh.

```
30 lines | src/Service/IsMacArgumentValueResolver.php
... lines 1 - 15
    public function resolve(Request $request, ArgumentMetadata $argument)
    {
        dump($request);
... lines 19 - 27
    }
... lines 29 - 30
```

Hover over the target icon on the web debug toolbar. Yep, *two* dumps. If we look at the query parameters for the first `Request`... it's got it! `mac=false`. But down on the second request, it has some `_path` query parameter, but *no* `mac`.

The point is: there are two different requests. And the fact that they don't all contain the same data is on *purpose*. Because of this, whenever you're handling a *sub-request*, it's not a good idea to read information from the request... because you're not *really* reading data from the correct request!

So how can we correctly read the `mac` query parameter from a sub-request? To learn how, let's get crazy and make our *own* sub-request directly in PHP.

Chapter 25: Manually Making a Sub Request

To understand more about sub requests, let's create one by hand! Because, it's not super obvious *what* these two Twig functions are *really* doing behind-the-scenes.

Insides our homepage controller, let's execute a sub request right here. How? It's simpler than you might think. Step 1: create a new request object: `$request = new Request()`. This is a totally empty `Request` object: it basically has nothing in it.

```
86 lines | src/Controller/ArticleController.php
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 34
    public function homepage(ArticleRepository $repository, LoggerInterface $logger, $isMac, HttpKernelInterface $httpKern
    {
... lines 37 - 39
        // manual sub-request example
        $request = new Request();
... lines 42 - 54
    }
... lines 56 - 84
}
```

It's not like the `Request::createFromGlobals()` method that we saw earlier. *That* method pre-populates the object with all the current request information. This does *not* do that. To render the partial controller, set the request attribute: `$request->attributes->set('_controller')` and set that to the same string we have inside our Twig template. I'll copy that... and paste it here: `'App\\Controller\\PartialController::trendingQuotes'`.

```
86 lines | src/Controller/ArticleController.php
... lines 1 - 34
    public function homepage(ArticleRepository $repository, LoggerInterface $logger, $isMac, HttpKernelInterface $httpKern
    {
... lines 37 - 41
        $request->attributes->set('_controller', 'App\\Controller\\PartialController::trendingQuotes');
... lines 43 - 54
    }
... lines 56 - 86
```

We now have a `Request` object with nothing in it except for an `_controller` attribute. And... that's all we need! Well, to work around some internal validation that checks for a valid IP address, we *also* need to say `$request->server->set('REMOTE_ADDR', '127.0.0.1')`.

```
86 lines | src/Controller/ArticleController.php
... lines 1 - 34
    public function homepage(ArticleRepository $repository, LoggerInterface $logger, $isMac, HttpKernelInterface $httpKern
    {
... lines 37 - 42
        $request->server->set('REMOTE_ADDR', '127.0.0.1');
... lines 44 - 54
    }
... lines 56 - 86
```

To send this into `HttpKernel`, we can fetch that *service*. Yes, even the mighty `HttpKernel` is a service in the container. Add another argument: `HttpKernelInterface $httpKernel`. Then, down here, we can say

`$response = $httpKernel->handle()`. We're going to pass this *two* arguments. We already know from `index.php` that the first argument is the `Request`. So, pass `$request`. But there is also an optional *second* argument: the request "type". This allows you to pass a flag that indicates if this is a "master" request - that's the default - or if this is a sub-request - some request that is happening *inside* the main one. That's our situation, so pass: `HttpKernelInterface::SUB_REQUEST`.

86 lines | src/Controller/ArticleController.php

```
... lines 1 - 10
use Symfony\Component\HttpKernel\HttpKernelInterface;
... lines 12 - 34
    public function homepage(ArticleRepository $repository, LoggerInterface $logger, $isMac, HttpKernelInterface $httpKern
    {
... lines 37 - 44
        $response = $httpKernel->handle(
            $request,
            HttpKernelInterface::SUB_REQUEST
        );
... lines 49 - 54
    }
... lines 56 - 86
```

What difference will that make? Not much. But listeners to almost *every* event that we've seen are *passed* this flag on the event object and can behave *differently* based on whether or not a master or sub request is being handled. We'll see that in a few minutes.

To check if this works, `dump($response)`.

86 lines | src/Controller/ArticleController.php

```
... lines 1 - 34
    public function homepage(ArticleRepository $repository, LoggerInterface $logger, $isMac, HttpKernelInterface $httpKern
    {
... lines 37 - 48
        dump($response);
... lines 50 - 54
    }
... lines 56 - 86
```

Um... ok! Let's try this! We added this to the homepage... so refresh. Everything looks normal on this *main* request. Now hover over the target icon on the web debug toolbar. There it is! A dumped `Response` with the trending quotes content inside.

And, yes, if we click the time icon on the web debug toolbar to get to the Performance section of the profiler, we can see our sub request! Heck, now we have *two* sub requests: our "manual" sub-request and then the one from the template.

Set the threshold back down to 0 milliseconds. *Way* down on the main profiler, the sub-request shows up as this strange `__section__.child` thing.

Go back to the homepage controller and comment out the sub request logic.

```
88 lines | src/Controller/ArticleController.php
... lines 1 - 34
    public function homepage(ArticleRepository $repository, LoggerInterface $logger, $isMac, HttpKernelInterface $httpKern
    {
... lines 37 - 39
        /*
        // manual sub-request example
        $request = new Request();
        $request->attributes->set('_controller', 'App\\Controller\\PartialController::trendingQuotes');
        $request->server->set('REMOTE_ADDR', '127.0.0.1');

        $response = $httpKernel->handle(
            $request,
            HttpKernelInterface::SUB_REQUEST
        );
        dump($response);
        */
... lines 52 - 56
    }
... lines 58 - 88
```

I wanted you to see that this is *all* that *really* happens to trigger a sub request.

Listeners and the isMasterRequest() Flag

As we talked about, many listeners will use this `SUB_REQUEST` flag to *change* their behavior. Because sometimes, it only makes sense for a listener to do its work on the main, *master* request. For example - if you wrote a custom listener that checked the URL and denied access based on some custom logic, that listener only needs to *do* that check on the *main* request. It either denies access or allows access initially, and then the rest of the page should render normally.

Our `UserAgentSubscriber` is a perfect example of this. It makes no sense to read the `User-Agent` off of a sub request. It might work - because, in reality, sub-requests copy *some* of the data from the main request, but trying to read real information off of the request in a sub-request is asking for trouble. I *really* want you to think of the master and sub requests as *totally* independent objects.

So, what can we do? At the very top of our listener, if *not* `$event->isMasterRequest()`, simply return.

```
47 lines | src/EventListener/UserAgentSubscriber.php
... lines 1 - 10
class UserAgentSubscriber implements EventSubscriberInterface
{
... lines 13 - 19
    public function onKernelRequest(RequestEvent $event)
    {
        if (!$event->isMasterRequest()) {
            return;
        }
... lines 25 - 37
    }
... lines 39 - 45
}
```

The `isMasterRequest()` method is a shortcut to check the flag that was originally passed to `HttpKernel::handle()`. Our listener *will* still be called on a sub-request, but now it will do *nothing*. And that makes sense: this class is doing *nothing* more than logging the `User-Agent`. We didn't realize it before, but thanks to our sub-request, each page refresh was logging the `User-Agent` *twice*: one for the main request and once for the sub-request.

Ok, but! We still haven't fixed our original problem: when we add `?mac=false` to the URL, this is *correctly* read on the master request but *incorrectly* on the sub request. That's because we're trying to read that query parameter from *inside* the sub request... which doesn't work.

How can we fix that? The answer leverages an *old* friend of ours and will also touch on the proper way to pass data from the main request to the sub-request if you want to use HTTP caching with edge side includes. That's next.

Chapter 26: Sub Request Attributes

We have this problem: we know the *cause*, but not the *solution*. Basically, we're reading this ?isMac=false from both our master request - where it's being read correctly - *and* our sub-request - where it is *not*. That's because those are two separate request objects. And, in general, we should avoid reading request info from a sub-request... because we're not working with the *real* Request object!

But accidentally trying to *use* data from the real request in a sub-request was *so* easy to do! Inside of PartialController , we simply added an $isMac argument like we do everywhere else. This argument *was* passed thanks to our custom IsMacArgumentValueResolver : this is executed on *every* request. But the *second* time it's called, the request is *not* the real request.

Here's the plan: I want to prevent any controllers that are being called on a sub-request from being able to have an $isMac argument. If we try, I want an exception to be thrown.

By the way, it *is* actually possible to use the RequestStack from *anywhere* to get the "master" request: it has a getMasterRequest() method. But! If you're using HTTP caching with edge side includes, this will *break*. The solution we're going to show is the proper one... and a lot more fun to implement for a deep dive.

Passing isMac through the Request Attributes

The *easiest* way to do this would be to go to our argument value resolver, find out if this is a sub-request and return false from supports() in that case. But... there's not a *great* way to figure out if this is the master or sub-request from here. That's mostly a superpower of event listeners.

So, let's use a trick: let's allow the *listener* to figure out whether or not we're using a Mac. And then, it can *pass* that information over to our argument value resolver.

Inside the subscriber, create a new private function called isMac() : it will take in the Request object from HttpFoundation and return a boolean. For the logic, copy everything from inside of resolve() , remove it and paste it here. Change both of the yield calls to return .

58 lines | src/EventListener/UserAgentSubscriber.php

```
... lines 1 - 10
class UserAgentSubscriber implements EventSubscriberInterface
{
... lines 13 - 46
    private function isMac(Request $request): bool
    {
        if ($request->query->has('mac')) {
            return $request->query->getBoolean('mac');
        }

        $userAgent = $request->headers->get('User-Agent');

        return stripos($userAgent, 'Mac') !== false;
    }
}
```

Perfect! We now have a function to tell us if we're using a Mac. Now, up in onKernelRequest() , if this is a sub-request, the method returns immediately. But if it's a *master* request, let's work some magic. How can we pass the isMac value to the argument value resolver? We could create and call a setter on it. *Or*, we could put that into the request attributes! I mean, that *is* the proper place for information about the request that is specific to your app. This is a perfect example!

Do it with $request->attributes->set() and create a key called _isMac set to $this->isMac($request) .

```
60 lines | src/EventListener/UserAgentSubscriber.php
... lines 1 - 19
    public function onKernelRequest(RequestEvent $event)
    {
... lines 22 - 38
        $request->attributes->set('_isMac', $this->isMac($request));
    }
... lines 41 - 60
```

We're calling it `_isMac` because if we called it just `isMac`, we wouldn't even *need* the argument value resolver! It would immediately be possible to have an `$isMac` argument without an error. So... I want to try to do this, kind of, the hard way.

Move over to the `IsMacArgumentValueResolver`. Here, we're going to *read* that attribute, which will *only* exist on the *master* request. Inside of `supports()`, add `&& $request->attributes->has('_isMac')`. Supports will *now* return false for a sub request.

```
21 lines | src/Service/IsMacArgumentValueResolver.php
... lines 1 - 8
class IsMacArgumentValueResolver implements ArgumentValueResolverInterface
{
    public function supports(Request $request, ArgumentMetadata $argument)
    {
        return $argument->getName() === 'isMac' && $request->attributes->has('_isMac');
    }
... lines 15 - 19
}
```

In `resolve`, `yield $request->attributes->get('_isMac')`

```
21 lines | src/Service/IsMacArgumentValueResolver.php
... lines 1 - 15
    public function resolve(Request $request, ArgumentMetadata $argument)
    {
        yield $request->attributes->get('_isMac');
    }
... lines 20 - 21
```

That's it! Does it work? In your browser, *first* open an article show page in a new tab. Here, it works: you can see it on the web debug toolbar: we're dumping the `$isMac` argument in the controller. This *proves* that we haven't broken the master request.

But now, refresh the homepage. It explodes!

> An exception has been throwing during the rendering of a template: could not resolve `$isMac` argument.

And you can see that it hits this on our `{{ render(controller()) }}` line. This is *exactly* what I wanted: I can no longer accidentally use this argument: it *only* works on the *master* request.

Passing Attributes Directly to a Sub-Request

Of course... the question *now* is: how do we fix this? What if I really *do* need to know the `$isMac` value from inside a sub request? I mean, that's a valid thing to want to know!

The solution is to *pass* the `isMac` value from your *main* request *to* your sub request object. In a Twig template, you can do this by passing an optional second array argument. Pass `isMac` set to `isMac`.

65 lines | templates/article/homepage.html.twig

```
... lines 1 - 45
        <div class="col-sm-12 col-md-4 text-center">
... lines 47 - 57
            {{ render(controller('App\\Controller\\PartialController::trendingQuotes', {
                isMac: isMac
            })) }}
        </div>
... lines 62 - 65
```

Before we chat about this, let's see if it works. Refresh and... it *does*! Woh! Let... me explain. The second argument to `controller()` is an array of values that you want to pass to the *attributes* of the sub-request object. Why does that work? Well, for the sub-request, the `$isMac` controller argument is actually *not* being passed to us thanks to the `IsMacArgumentValueResolver`. Nope! It works simply because we put an `isMac` key into the request attributes... and anything inside request attributes are *allowed* as controller arguments.

It *feels* like the second argument to the Twig `controller()` function is just an "array of variables to pass to the controller". And while that's *essentially* true, it *really* works thanks to the request attributes.

Back on the homepage, because we still have the `?mac=false` query parameter, the message is *not* rendered in both places. If we remove that query parameter... yes! It's displayed consistently.

Attributes and Edge Side Includes

As an added benefit, this was of passing variables from the master to the sub-request works *perfectly* with edge side includes and HTTP caching. Open the profiler, go to the Performance tab, find the sub request and click to go into *its* profiler. See this internal URL up here? If we were *truly* using edge side includes, this URL *would* be a real URL used by your HTTP cache to fetch this fragment *and* the URL would have an extra `isMac=true` or `false` query parameter. For ESI, request attributes becomes *part* of the URL. And since the URL operates as the cache *key* for HTTP caching, this means that you would have a *separate* cache for the version of your trending quotes *with* the "isMac" message and *without*. That's perfect.

Close up that tab. Now that we know that this array become request attributes, *another* solution would be to pass `_isMac`. If we did that, it would be used by our argument value resolver. `UserAgentSubscriber` would still do nothing because it's a sub request... but the `_isMac` attribute *would* be there and the `IsMacArgumentValueResolver` would make the `isMac` argument available to `PartialController`. Two different ways to do the exact same thing... and both are *super* nerdy.

Ok friends! Wow! That was *fun*! I hope you enjoyed this deep dive into the heart of Symfony's HttpKernel as much as I did. If you have still have questions or want to deep-dive into a different part of Symfony, let us know in the comments.

Happy hacking! And we'll see ya next time.