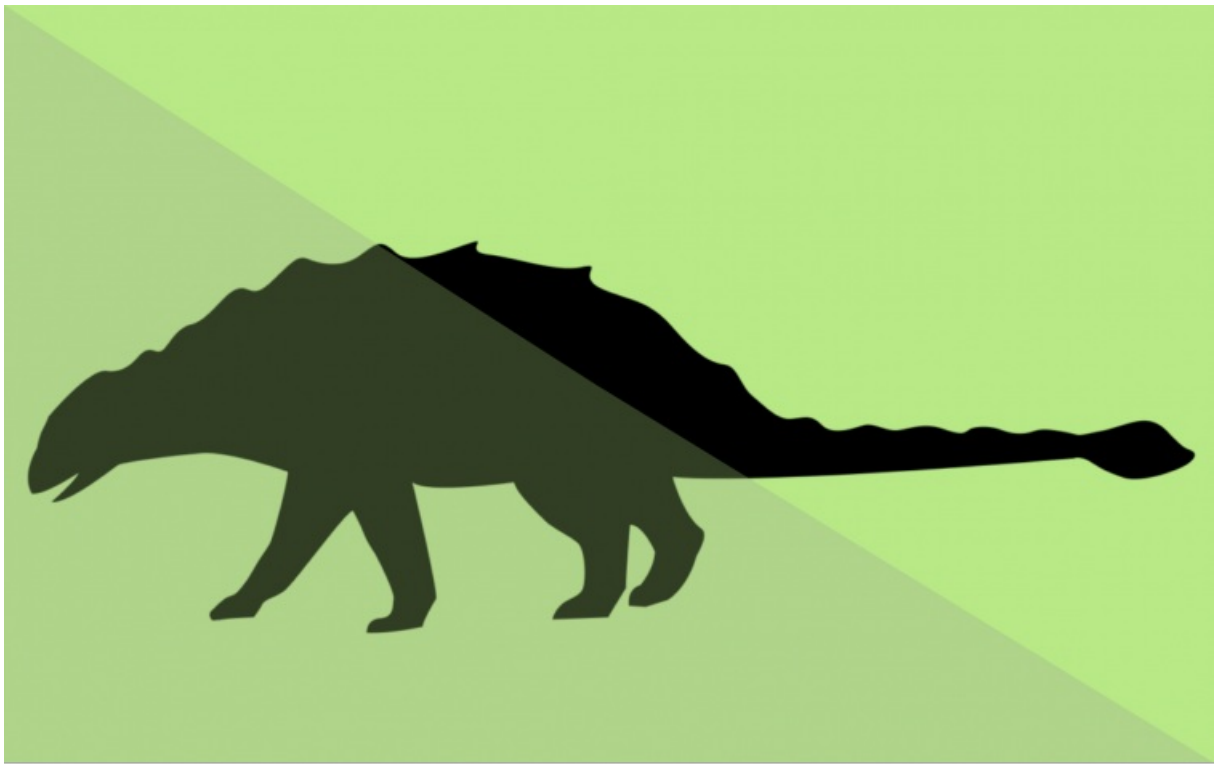
Journey to the Center of Symfony: HttpKernel Request-Response



With <3 from SymfonyCasts

Chapter 1: Interrupt Symfony with an Event Subscriber

Hey guys! Welcome to a series that we're calling: Journey to the Center of Symfony! In this first part, we'll be talking about the deep, dark core piece called the HttpKernel, a wondrous component that not only sits at the heart of Symfony, but also at the heart of Silex, Drupal 8, PhpBB and a lot of other stuff. How is that possible? We'll find out! And this stuff is really nerdy, so we're going to have some fun.

Getting the Project Running

I already have the starting point of our app ready on my computer. You can download this right on the screencast page. I've already run composer install, I've already created my database, I've already created my schema: I won't show those things here because you guys are a bit more of experts. We do have fixtures, so let's load those.

Now let's use the built-in PHP web server to get our site running.

Perfect!

So in true Journey to the Center of the "Symfony" theme, we're going to talk about dinosaurs. I've already created an app, which has 2 pages. We can list dinosaurs - these are coming out of the database - and if we click on one of them, we go to the show page for that dinosaur.

Big Picture: Request-Route-Controller-Response

No matter what technology or framework we're using, our goal is always to start with a request and use that to create a response. Everything in between those 2 steps will be different based on your tech or framework. In our app, and in almost every framework, two things that are going to be between the request and response are the route and controller. In this case, you can see our homepage has a route, our function is a controller, and our controller returns a Response object::

42 lines | src/AppBundle/Controller/DinosaurController.php

```
... lines 1 - 10
    /**
     * @Route("/", name="dinosaur_list")
     */
    public function indexAction()
    {
        $dinos = $this->getDoctrine()
            ->getRepository('AppBundle:Dinosaur')
            ->findAll();

        return $this->render('dinosaurs/index.html.twig', [
            'dinos' => $dinos,
        ]);
    }
... lines 24 - 42
```

And we have the same thing down here with the other page: it has a route, a controller, and that returns a response::

42 lines | src/AppBundle/Controller/DinosaurController.php

```
... lines 1 - 24
    /**
     * @Route("/dinosaurs/{id}", name="dinosaur_show")
     */
    public function showAction($id)
    {
        $dino = $this->getDoctrine()
            ->getRepository('AppBundle:Dinosaur')
            ->find($id);

        if (!$dino) {
            throw $this->createNotFoundException('That dino is extinct!');
        }

        return $this->render('dinosaurs/show.html.twig', [
            'dino' => $dino,
        ]);
    }
```

So what we're going to look at is *how* that all works. Who actually runs the router? Who calls the controller? How do events work in between the request-response flow?

But before we dive into that, what we're going to do first is create an event listener and hook into that process. Then we'll be able to play with that event listener as we dive into the core of things.

The Best Parts of the Web Profiler

I'm going to open up the profiler and go to the timeline. This is going to be our guide to this whole process. This shows everything that happens between the request and the response. Even if you don't understand what's happening yet, after we go through everything, this is going to be a lot more interesting. You can already see where our controller is called, and under the controller you can see the Twig template and even some Doctrine calls being made.

Before and after that, there are a lot of event listeners - you notice a lot of things that end in the word Listener. That's because most of the things that happen between the request and the response in Symfony are events: you have the chance to hook into them with event listeners.

In fact, one other tab I really like on here is the Events tab. You can see there's some event called kernel.request. Maybe you already understand what that means, maybe you don't, but you will soon. There's another event called kernel.controller with listeners and several other events. We're going to see where these events are dispatched and why you would add a hook to one versus another.

Creating an Event Subscriber/Listener

Let's create a listener on that kernel.request event! In my AppBundle, I'll create a new directory called EventListener and a new class. Inside this event listener, we're going to read the User-Agent header off the request and do some things with that. So I'll call this UserAgentSubscriber::

8 lines | src/AppBundle/EventListener/UserAgentSubscriber.php

```
<?php

namespace AppBundle\EventListener;

class UserAgentSubscriber
{
}
```

If you want to hook into Symfony, there are 2 ways to do it: with a listener or a subscriber. They're actually exactly the same, the only difference is where you configure *which* events you want to listen to.

I'm going to create a subscriber here because it's a little more flexible. So UserAgentSubscriber needs to implement EventSubscriberInterface::

21 lines | src/AppBundle/EventListener/UserAgentSubscriber.php

```
... lines 1 - 2
namespace AppBundle\EventListener;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class UserAgentSubscriber implements EventSubscriberInterface
{
... lines 9 - 19
}
```

Notice that it added the use statement up there. And we're going to need to implement 1 method which is getSubscribedEvents. What this is going to return is a simple array that says: Hey, apparently there's some event whose name is kernel.request - we don't necessary know why it's called or what it does yet - but when that event happens, I want Symfony to call this onKernelRequest function, which we're going to put inside of this class. For now, let's just put a die('it works');::

21 lines | src/AppBundle/EventListener/UserAgentSubscriber.php

```
... lines 1 - 6
class UserAgentSubscriber implements EventSubscriberInterface
{
    public function onKernelRequest()
    {
        die('it works');
    }

    public static function getSubscribedEvents()
    {
        return array(
            'kernel.request' => 'onKernelRequest'
        );
    }
}
```

Cool! The event subscriber is ready to go. No, Symfony doesn't automatically know this class is here or automatically scan the codebase. So to get Symfony to know that there's a new UserEventSubscriber that wants to listen on the kernel.request event, we're going to need to register this as a service.

Registering the Subscriber/Listener

So I'm going to go into app/config/services.yml and clear the comments out. And we'll give it a short, but descriptive name - user_agent_subscriber, the name of the service doesn't really matter in this case. There are no arguments yet, so I'll just put an empty array. Now in order for Symfony to know this is an event subscriber, we'll use something called a tag, and set its name to kernel.event_subscriber:

11 lines | app/config/services.yml

```
... lines 1 - 5
services:
    user_agent_subscriber:
        class: AppBundle\EventListener\UserAgentSubscriber
        tags:
            - { name: kernel.event_subscriber }
```

Now, that tag is called a dependency injection tag, which is really awesome, really advanced and really fun to work with inside of Symfony. And we're going to talk about it in a different part of this series. With just this configuration, Symfony will

boot, it'll know about our subscriber, and when that kernel.request event happens, it *should* call our function.

Sweet!

Logging Something in the Subscriber

Now inside of onKernelRequest, let's do some real work. For now, I want to log a message. I'm going to need the logger so I'll add a constructor and even type hint the argument with the PSR LoggerInterface. And I'll use a little PHPStorm shortcut to create and set that property for me::

29 lines | src/AppBundle/EventListener/UserAgentSubscriber.php

```
... lines 1 - 4
use Psr\Log\LoggerInterface;
... lines 6 - 7
class UserAgentSubscriber implements EventSubscriberInterface
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }
... lines 16 - 27
}
```

Now in our function, we'll log a very important message::

29 lines | src/AppBundle/EventListener/UserAgentSubscriber.php

```
... lines 1 - 16
    public function onKernelRequest()
    {
        $this->logger->info('Yea, it totally works!');
    }
... lines 21 - 29
```

And of course this isn't going to work unless we go back to services.yml and tell Symfony: Hey, we need the @logger service:

12 lines | app/config/services.yml

```
... lines 1 - 5
services:
    user_agent_subscriber:
        class: AppBundle\EventListener\UserAgentSubscriber
        arguments: ["@logger"]
        tags:
            - { name: kernel.event_subscriber }
```

Cool!

Let's refresh! It works, and if we click into the profiler, one of the tabs is called "Logs", and under "info" we can see the message. So this is already working, and if we go back to the Timeline and look closely, we should see our UserAgentSubscriber. And it's right there. Also, if we go back to the events tab, we see the kernel.request with all of its listeners. And if you look at the bottom, you see our UserAgentSubscriber on that list too.

So we're hooking into that process already, even if we don't understand what's going on with it.

Every Listener Gets an Event Object

Whenever you listen to *any* event - whether it's one of Symfony's core events or it's an event from a third-party bundle you

installed, your function is passed an $event argument. So, we'll add $event. The only trick is that you don't automatically know what type of object that is, because every event you listen to is going to pass you a different type of event object.

But no worries! I'm going to use the new dump() function from Symfony 2.6::

30 lines | src/AppBundle/EventListener/UserAgentSubscriber.php

```
... lines 1 - 16
17      public function onKernelRequest($event)
18      {
19          dump($event);
20          $this->logger->info('Yea, it totally works!');
21      }
... lines 22 - 30
```

Let's go back a few pages, refresh, and the dump function prints that out right in the web debug toolbar. And we can see it's dumping a GetResponseEvent object. So that's awesome - now we know what type of object is being passed to us. And that's important because every event object will have different methods and different information on it.

Let's type-hint the argument. Notice I'm using PHPStorm, so that added a nice use statement to the top - don't forget that::

33 lines | src/AppBundle/EventListener/UserAgentSubscriber.php

```
... lines 1 - 6
7   use Symfony\Component\HttpKernel\Event\GetResponseEvent;
... lines 8 - 17
18      public function onKernelRequest(GetResponseEvent $event)
19      {
... lines 20 - 23
24      }
... lines 25 - 33
```

What I want to do is get the User-Agent header and print that out in a log message. Fortunately, this getResponseEvent object gives us access to the request object. And again, every event you listen to will give you a different event object, and every event object will have different methods and information on it. It just *happens* to be that this one has a getRequest method, which is really handy for what we want to do. Now I'll just read the User-Agent off of the headers, and log a message::

33 lines | src/AppBundle/EventListener/UserAgentSubscriber.php

```
... lines 1 - 17
18      public function onKernelRequest(GetResponseEvent $event)
19      {
20          $request = $event->getRequest();
21          $userAgent = $request->headers->get('User-Agent');
22
23          $this->logger->info('Hello there browser: '.$userAgent);
24      }
... lines 25 - 33
```

Let's try it! I'll get back into the profiler, then to the Logs... and it's working perfectly.

Even if we don't understand everything that's happening between the request and response, we already know that there are these listeners that happen. But next, we're going to walk through the code that handles *all* of this.

Chapter 2: HttpKernel::handle() The Heart of Everything

HTTPKERNEL::HANDLE() THE HEART OF EVERYTHING¶

We know we start with the request, we have a routing layer, eventually something calls the controller, and the controller returns a Response. Let's trace through the process in Symfony to see how all of this works. It's going to be awesome, I promise.

Front Controller: app.php/app_dev.php¶

Since we're using the built-in web server, this is actually executing the `app_dev.php` file in the `web/` directory. So let's go there to start. And really, `app.php` is basically identical, so it doesn't matter which one we start going through. Ignore all the stuff at the top of `app_dev.php`, I want to get to the good stuff:

```
// web/app_dev.php
use Symfony\Component\HttpFoundation\Request;
// ...

$kernel = new AppKernel('dev', true);
$kernel->loadClassCache();
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

The first important thing is that we instantiate the `AppKernel`, which is the same `AppKernel` you have in your `app/` directory where you register your bundles. We'll talk more about this class in another part of this series. We'll also talk then about this `loadClassCache`, but for now, comment it out - it can get in the way if you're debugging core classes:

```
// web/app_dev.php
use Symfony\Component\HttpFoundation\Request;
// ...

$kernel = new AppKernel('dev', true);
// comment this out for now
// $kernel->loadClassCache();

$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

For now, just know that `AppKernel` is the heart of your app. The first *real* important thing I want to look at is the `Request::createFromGlobals`. We all know that we start with the request, and hey, here's where that Request is born! The `Request` is just a simple object that holds data, and this creates a new one that's populated from the superglobal variables - those things like `$_GET`, `$_POST`, `$_SERVER`. In fact, if we look inside, it's creating an instance of itself populated from those variables:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpFoundation/Request.php
// ...

public static function createFromGlobals()
{
    $request = self::createRequestFromFactory($_GET, $_POST, array(), $_COOKIE, $_FILES, $_SERVER);
    // ...

    return $request;
}
```

The Most Important Function in Symfony¶

Now that we're starting with the request object, the next line is the most important line inside of Symfony: the `$kernel->handle()` method:

```
// web/app_dev.php
// ...

$response = $kernel->handle($request);
// ...
```

What's really cool about this is that you see that the `handle()` method has input request, output response. And that is actually *amazing*. It means that everything that our application is - the routing, the controller, the services - all of that stuff happens inside of this `handle()` function. It also means that our entire application - all of those layers - have been boiled down to a single function. And how pretty is it: input request, output response. Because after all, that's our job: read the incoming request, create a response.

Why Yes, you can Handle Many Requests at Once¶

What I want to do with you guys is look inside that `handle()` function and see what's going on. But first, I kind of want to prove a point about how our application has just been reduced down to a single function. One of the things about a well-written function is that you can call it over and over again. So in theory, if we wanted to here, we could actually create a second request by hand. I'll use a `create()` function:

```
// web/app_dev.php
// ...

$request2 = Request::create('/dinosaurs/22');
```

What I want to do is process *two* requests at once. As you'll see later, internally in Symfony, there actually is a real use-case for this. But for now, I'm just playing around, so we'll have this request look like it wants to go to `/dinosaurs/22`, which is.... one of the pages we actually have right now.

Then we'll say `$response2 = $kernel->handle($request2);`. We're processing two separate requests and processing two separate responses. I'm going to comment out the `send()` and `terminate()` calls temporarily, and to prove that things are working, I'm going to echo the content from the original request, then echo the content from the second response. So hey, let's see if we can print out two pages at once:

```
// web/app_dev.php
// ...

$request = Request::createFromGlobals();
$response = $kernel->handle($request);

$request2 = Request::create('/dinosaurs/22');
$response2 = $kernel->handle($request2);

echo $response;
echo $response2;
```

When we go back and refresh, this is really cool! On top, we see page that we're actually going to, and below, we see the whole other page that was processed beneath that. Our application is just a function: input request, output response. And that's a really powerful thing to realize.

Let me undo all of this, and get back to where we started.

Introducing HttpKernel::handle()¶

Let's look inside of that `$kernel->handle()` method. Again, the `$kernel` class is our `AppKernel`. If I hold cmd (or ctrl for other OS's) and click into the `handle()` function, it's going to take us not into `AppKernel`, but its parent class `Kernel`. This class is something we're going to talk about in a different part of this series. Ignore it for now, because it offloads the work to something called `HttpKernel`.

I'll use a shortcut my editor to open `HttpKernel`. In PhpStorm, you can go to Navigate, then Class or File. So I'll use the Cmd+O shortcut to open up the `HttpKernel` class. We're looking for that `handle()` function. Because effectively, when we call `handle()` in `app_dev.php`, it's being passed to this `handle()` function - you can see the `$request` argument:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

public function handle(Request $request, $type = HttpKernelInterface::MASTER_REQUEST, $catch = true)
{
    try {
        return $this->handleRaw($request, $type);
    } catch (\Exception $e) {
        if (false === $catch) {
            $this->finishRequest($request, $type);

            throw $e;
        }

        return $this->handleException($e, $request, $type);
    }
}
```

Awesome! Now, what is handle actually doing? So far, not much. The important thing to take-away here is that there is a try-catch block. This means that if you throw an exception from anywhere inside your application - like a controller or a service - it's going to get caught by this block. And when that happens, you'll get passed to that `handleException()` function, which is what tries to figure out what response to send back to the user when there's an error. That's something we'll talk about later.

The real guts of this are in a function called `handleRaw()`. And this lives just a little bit further down inside this same class:

```
private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // about 45 lines of awesome that we'll walk-through
}
```

We're going to walk through every line in this function. You can see that it's not that long, and what's really amazing is that `handleRaw` *is* the Symfony Framework. This is the dark core guts of it. But it's also the dark, core guts of Drupal 8, and the

dark, core guts of Silex, of PhpBB. All these very different pieces of software all use this same function. How is that possible? How could this one function be responsible for a Symfony application and a Drupal 8 application and a PhpBB application? We'll find out.

Chapter 3: kernel.request and the RouterListener

KERNEL.REQUEST AND THE ROUTERLISTENER¶

So let's start working through this. The first thing we have is this `$this->requestStack->push()` :

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    $this->requestStack->push($request);
    // ...
}
```

This is going to make more sense a few lessons from now. But as we saw before, Symfony has the ability to process many requests at a time via something called *What about Sub Requests?*. The `$requestStack` is just an object that keeps track of which request is being worked on right now. We'll talk more about that idea later - for now, just ignore it.

THE KERNEL.REQUEST EVENT¶

For us, the first interesting thing that happens inside of this function is that it creates a `GetResponseEvent` object:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    // request
    $event = new GetResponseEvent($this, $request, $type);

    // ...
}
```

Despite its name, that doesn't do anything, that just creates a new object with some data in it. Some of you may already be saying: "Hold on, that class sounds familiar." If we look inside our `UserAgentSubscriber` , you'll remember that it was passed a `GetResponseEvent` object:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...
}
```

And this is where that object was created. It then goes to a dispatcher and dispatches our first event:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    // request
    $event = new GetResponseEvent($this, $request, $type);
    $this->dispatcher->dispatch(KernelEvents::REQUEST, $event);

    // ...
}
```

That `KernelEvents::REQUEST` is just a constant that means `kernel.request`. So ultimately, this line causes the dispatcher to loop over and call *all* of the functions that are "listening" to the `kernel.request` event.

And because of this nice constant, we can actually use it inside our subscriber. So I'll change this to `KernelEvents::REQUEST` and add a comment so I don't forget what this really means:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...

use Symfony\Component\HttpKernel\KernelEvents;

class UserAgentSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            // constant that means kernel.request
            KernelEvents::REQUEST => 'onKernelRequest'
        );
    }
}
```

So when every listener like to this is called, it's passed that `GetResponseEvent` that's created in `HttpKernel`.

KING OF ROUTING: ROUTERLISTENER¶

There are a lot of listener to `kernel.request`, but there's only one that's mission critical to how the Symfony Framework works. Let's go back to the browser, refresh the page, and click into the profiler. Back on the events tab, we can see all of the functions that are called when this event is dispatched. The one that's important to understand how the framework works is that `RouterListener` one that we can see in the middle. So let's go back and open up `RouterListener` in PhpStorm. If you ever see a class in the `app/cache` directory like this, ignore that, you want the one that's really inside of the `vendor/` directory.

If you look at the bottom, this is an event subscriber, just like our event subscriber:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/EventListener/RouterListener.php
// ...

class RouterListener implements EventSubscriberInterface
{
    // ...

    public static function getSubscribedEvents()
    {
        return array(
            KernelEvents::REQUEST => array(array('onKernelRequest', 32)),
            KernelEvents::FINISH_REQUEST => array(array('onKernelFinishRequest', 0)),
        );
    }
}
```

And you can see it listens on `kernel.request` and its `onKernelRequest` method is called. As the name of this class sounds, this is the class that's responsible for actually executing your routing. And remember, we can write routing in a lot of different ways: in YAML files, in annotations like in our app or other formats. But ultimately, if you run `router:debug`, you'll get a list of all of the routes:

```
php app/console router:debug
```

Most of these are internal debugging routes, but our 2 are at the bottom:

```
Name                Path
_wdt                /_wdt/{token}
...

dinosaur_list       /
dinosaur_show       /dinosaurs/{id}
```

What does Routing Do?¶

My question is, what's the end result of router? For example, if we're on the homepage, does it return the string `dinosaur_list`? Something else? If we trace down in the function, between lines 124 and 128, that's where the router is actually being run:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/EventListener/RouterListener.php
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    if ($this->matcher instanceof RequestMatcherInterface) {
        $parameters = $this->matcher->matchRequest($request);
    } else {
        $parameters = $this->matcher->match($request->getPathInfo());
    }

    // ...
}
```

You can see there's an if statement, but that's not important - both sides do the same thing: they figure out which route matched.

Routing Parameters¶

And you can see that it returns some `$parameters` variable. What's in that? Is that the name of the route or something else?

Let's use the `dump()` function to find out:

```
if ($this->matcher instanceof RequestMatcherInterface) {
    $parameters = $this->matcher->matchRequest($request);
} else {
    $parameters = $this->matcher->match($request->getPathInfo());
}

dump($parameters);die;
```

This runs on every request, so let's go back, refresh, and there it is. When the router matches a route, what it returns is an array of information about that route:

```
array(
    '_controller' => 'AppBundle\Controller\DinosaurController::indexAction',
    '_route' => 'dinosaur_list',
)
```

And the most important piece of information is what controller to execute, which it puts on an `_controller` key. Because remember, our flow is always request, routing, routing points to a controller, and the controller is where we return the `Response`.

The format of the controller is the class name, `::` then the method name. If you open up the `DinosaurController`, this all comes from here:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

/**
 * @Route("/", name="dinosaur_list")
 */
public function indexAction()
{
    // ...
}
```

When you use annotation routing, it's smart enough to create a route where the `_controller` is set to whatever method is below it.

The _controller key and Yaml Routes¶

If you've used YAML routes before, it's even more interesting. I'll open up my `app/config/routing.yml` file. At the bottom in comments, I prepared a route that's identical to the one we're building in annotations:

```
# app/config/routing.yml
# ...

#dinosaur_list:
#    path: /
#    defaults:
#        _controller: AppBundle:Dinosaur:index
```

When you use YAML routing, to point to the controller you have a `defaults` key, and below that you have an `_controller` that uses a three-part syntax to point to the `DinosaurController` and `indexAction`.

So just to prove this is the same. I'm going to load *only* that route:

```
# app/config/routing.yml
# delete the top annotations import temporarily

dinosaur_list:
    path: /
    defaults:
        _controller: AppBundle:Dinosaur:index
```

And when we refresh, the routing parameters are exactly the same. Let's comment that back out:

```
# app/config/routing.yml
_app_bundle_annotations:
    resource: @AppBundle/Controller
    type:     annotation

#dinosaur_list:
#    path: /
#    defaults:
#        _controller: AppBundle:Dinosaur:index
```

Routing Wildcards are Parameters Too!¶

Now let's go to a different page - `/dinosaurs/22`. And what we see *now* is that in addition to `_controller` and `_route`, now we have the `id` value from the route:

```
array(
    '_controller' => 'AppBundle\Controller\DinosaurController::showAction',
    'id' => '22',
    '_route' => 'dinosaur_show',
)
```

This shows that the end result of the routing layer is an array that has the `_controller` key plus any wildcards that are in your URL. It also gives you the `_route` in case you need it, but that's not important.

THE DUMPED ROUTER (IT'S COOL!)¶

One more cool thing. If you look in the cache directory, you'll see a file called `appDevUrlMatcher.php`:

```
// app/cache/dev/appDevUrlMatcher.php
// ...

class appDevUrlMatcher extends Symfony\Bundle\FrameworkBundle\Routing\RedirectableUrlMatcher
{
    // ...
    public function match($pathinfo)
    {
        // ...

        // dinosaur_list
        if (rtrim($pathinfo, '/') === '') {
            if (substr($pathinfo, -1) !== '/') {
                return $this->redirect($pathinfo.'/', 'dinosaur_list');
            }

            return array (  '_controller' => 'AppBundle\\Controller\\DinosaurController::indexAction',  '_route' => 'dinosaur_list',);
        }

        // dinosaur_show
        if (0 === strpos($pathinfo, '/dinosaurs') && preg_match('#^/dinosaurs/(?P<id>[^/]++)$#s', $pathinfo, $matches)) {
            return $this->mergeDefaults(array_replace($matches, array('_route' => 'dinosaur_show')), array (  '_controller' => 'A
        }

        throw 0 < count($allow) ? new MethodNotAllowedException(array_unique($allow)) : new ResourceNotFoundExceptio
    }
}
```

This is the end result of parsing through all of your routes, whether they're written in annotations, YAML or some other format. So when we see `$this->matcher->match()` in `RouterListener`, that's actually calling the `match()` function you see inside of that cached class. Symfony is smart enough to parse all of our routes, then generate this big crazy, regex, if-statement matching algorithm. If we scroll to the bottom, you'll see our dinosaur pages. Ok, this isn't important to understand, I just think it's cool.

INTRODUCING: THE REQUEST ATTRIBUTES¶

So let's get rid of the `dump()` in `RouterListener` and trace through what happens next. If you look below, you'll see that it takes that `$parameters` array, which has `_controller` and `id`, and it puts onto a `$request->attributes` property:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/EventListener/RouterListener.php
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    if ($this->matcher instanceof RequestMatcherInterface) {
        $parameters = $this->matcher->matchRequest($request);
    } else {
        $parameters = $this->matcher->match($request->getPathInfo());
    }

    // ...

    $request->attributes->add($parameters);

    // ...
}
```

Symfony's request object has a bunch of these public properties. I'll open up the Components documentation for the HttpFoundation component, because it talks about this.

Every public property except for one, has a real-world equivalent. For example, if you want to get the query parameters, you say `$request->query->get()`. If you want to get the cookies, it's `$request->cookies->get()`. For the headers, it's

`$request->headers->get`. All of these are ways to get information that comes from the original HTTP request message.

The one weird guy is `$request-attributes`. It has no real-world equivalent. It's just a place for you to store application-specific information about the request. And route info is exactly that.

Putting information onto the `$request->attributes` property doesn't actually do anything. It's just a place to store data - it's not triggering any other systems. We're just modifying the request object, and that's it for the `RouterListener`.

Let's close this up and go back to `HttpKernel`. At this point, the *only* thing we've done is dispatch the `kernel.request` event and the only listener that's really important is the `RouterListener`. And all it did was modify the `$request->attributes`:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    // request
    $event = new GetResponseEvent($this, $request, $type);
    $this->dispatcher->dispatch(KernelEvents::REQUEST, $event);

    // ...
}
```

So not a lot has happened yet.

CREATING THE RESPONSE IMMEDIATELY IN A LISTENER¶

If we follow this down, there's a really interesting `if` statement:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    $this->dispatcher->dispatch(KernelEvents::REQUEST, $event);

    if ($event->hasResponse()) {
        return $this->filterResponse($event->getResponse(), $request, $type);
    }

    // ...
}
```

If the event has a response - and I'll show you what that means - it exits immediately before calling the controller or doing anything else. We'll look at that `filterResponse` method later, but it doesn't do anything mission critical.

This means that any listener to `kernel.request` can just create a `Response` object and say "I'm done". For example, if you had a maintenance mode, you could create a listener, check some flag to see if you're in maintenance mode, and return a response immediately that says: "We're fixing some things."

Let's try this. In `UserAgentSubscriber`, I'll create a new `Response` object. Make sure you use the one from the `HttpFoundation` component. PHPStorm did just add a `use` statement for me up on line 7. And we'll say "Come back later":

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...

use Symfony\Component\HttpFoundation\Response;
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    $response = new Response('Come back later');
}
```

And the `GetResponseEvent` object we're passed has an `$event->setResponse()` method on it. Remember, every event passes a different object, and this one happens to have this `setResponse` method on it. We'll make things more interesting and put this in a block so it only randomly sets the response:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...

use Symfony\Component\HttpFoundation\Response;
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    if (rand(0, 100) > 50) {
        $response = new Response('Come back later');
        $event->setResponse($response);
    }
}
```

If we go and refresh now, the page works fine, refresh again... mine is being stubborn. There we go. You can see that as soon the response is set, it just stops entirely.

I'll leave this code in there, but let's comment it out so it doesn't ruin our project:

```
if (rand(0, 100) > 50) {
    $response = new Response('Come back later');
    // $event->setResponse($response);
}
```

Perfect!

In reality, there's no listener that's setting the response for us. So our next job will be to figure out which controller function to call to create the Response.

Chapter 4: Finding and Instantiating the Controller

FINDING AND INSTANTIATING THE CONTROLLER¶

Ok guys, we made it through the routing layer! Looking back, all it really did was add an array with a few items to the `$request->attributes`. The *really* important thing it added is `_controller`, and that's a string that points to the class and function of our controller. But it *also* added the value of each wildcard that's in our route:

```
// effectively, RouterListener does this
$request->attributes->add(array(
    '_controller' => 'AppBundle\Controller\DinosaurController::showAction',
    'id'          => 22,
    '_route'      => 'dinosaur_show',
));
```

Now, Symfony needs to figure out which controller to execute for this request. Wait, isn't that what the `_controller` value is? Yes! Wait, no, not exactly. That's a string, and while it *looks* like a function, it's not technically a "callable" thing yet. You'll see what I mean.

The Controller Resolver¶

Symfony figures out which controller function to call for the request by using something called the `ControllerResolver` and calling `getController`. Ready to go one step deeper into the core? Great - let's open this class up. Oh, and there are *two* classes called `ControllerResolver`: one inside the HttpKernel component and the other is inside FrameworkBundle. Open both of them. The one in FrameworkBundle extends the other. See `BaseControllerResolver` is the one that lives in the component:

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.php
namespace Symfony\Bundle\FrameworkBundle\Controller;

// ...
use Symfony\Component\HttpKernel\Controller\ControllerResolver as BaseControllerResolver;

class ControllerResolver extends BaseControllerResolver
{
    // ...
}
```

Most functions we'll look at are in the parent class, but one is overridden in the child class.

The Controller comes from _controller¶

The `getController()` function lives in the HttpKernel `ControllerResolver`, so let's find it! A controller is *any* callable function and Symfony's HttpKernel doesn't care if that's an anonymous function or whether a method inside of an object. A lot of the code you'll see here is to support this very important fact.

Ok, this cool! What's the first thing this function does? It goes out to the `request->attributes` and looks for that `_controller` key:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

public function getController(Request $request)
{
    if (!$controller = $request->attributes->get('_controller')) {
        // log a message ...

        return false;
    }

    // ...
}
```

Ah, hah! So why do we use `_controller` in our Yaml routes? And why do annotation routes create a route with `_controller` behind the scenes? Simply because, *this* line looks for that key. And if it doesn't find one, this function doesn't do anything: it panics and exits immediately.

Is the _controller Already a Callable?¶

Most of the rest of the code is basically trying to figure out: "Hey, is the `_controller` key in the request attributes maybe *already* a callable function?" In our case it's not, and let's dump() it to get a reminder of what it looks like at this point:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

public function getController(Request $request)
{
    if (!$controller = $request->attributes->get('_controller')) {
        // log a message ...

        return false;
    }

    dump($controller);die;
    // ...
}
```

Go back to the homepage, refresh!

```
(string) 'AppBundle\Controller\DinosaurController::indexAction'
```

Our controller is this string, not technically a *callable* function yet, even though it looks like a function name. A variable is callable if you can invoke it via the `call_user_func` function.

But in other circumstances, the controller might already be callable. Silex is a *perfect* example! There, your controllers are anonymous functions, and behind the scenes, the function is set on the `_controller` key of the route:

```
// example silex app
$app = new Silex\Application();
$app->get('/dinosaurs/{id}', function($id) {
    return 'ROAR Dinosaur #'.$id;
});

// this is what happens in RouterListener
$request->attributes->add(array(
    '_controller' => function($id) {
        return 'ROAR Dinosaur #'.$id;
    },
    'id' => 22,
));
```

So for Silex, `_controller` *is* callable, and so it would exit earlier in this process.

Transforming _controller into a Callable¶

But in the Symfony Framework, we don't exit early. Instead, we fall down into the `createController` function:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

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

    $callable = $this->createController($controller);
    // ..

    return $callable;
}
```

This is overridden in the child `ControllerResolver`, so switch to the one that's in the `FrameworkBundle`. And there's our function!

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.php
// ...

protected function createController($controller)
{
    if (false === strpos($controller, '::')) {
        // ...
    }

    // ...
}
```

Ah, this is awesome again! Look at the first part: it looks to see if your controller already has a `::` syntax in the middle of it. And if you *don't* you fall into this first block.

Transforming AppBundle:Default:index¶

Remember that `AppBundle:Default:index` syntax you use in Yaml routing? That's handled here. The `$this->parser` you see on line 60 is responsible for transforming that 3-part syntax and into the longer class name `::` method name that *our's* already has:

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.php
// ...

protected function createController($controller)
{
    if (false === strpos($controller, '::')) {
        // ...

        // controller in the a:b:c notation then
        $controller = $this->parser->parse($controller);

        // ...
    }
    // ...
}
```

Tip

In reality, there is a similar layer that runs during the process of compiling routes that converts the `AppBundle:Default:index` syntax to `AppBundle\Controller\DefaultController::indexAction`. But the idea is still the same - this just makes runtime performance a bit better.

The Controller as a Service Syntax¶

The block below this - around line 63 - handles the service syntax (e.g. `my_dinosaur_controller:indexAction`):

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.php
// ...

protected function createController($controller)
{
    if (false === strpos($controller, '::')) {
        // ...

        // controller in the service:method notation
        list($service, $method) = explode(':', $controller, 2);

        return array($this->container->get($service), $method);

        // ...
    }
    // ...
}
```

So if you decide to register your controller as a service, you have a different syntax, and this is the logic that handles that.

Ultimately, we fall down to the bottom, and one way or another, we end up with a string, which is the class name, `::`, and then the method name.

Next, this splits those on the `::` and the strange `list()` function sets the first part to a variable called `$class` and everything after the `::` to a variable called `$method`. I'll dump those variables to be totally clear:

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.php
// ...

protected function createController($controller)
{
    // ...

    list($class, $method) = explode('::', $controller, 2);
    dump($class, $method);die;
    // ...
}
```

The `list()` function has confused me in the past, and the *real* key is what these two new variables are set to.

At this point, it's time to see if we've messed up! Maybe this class doesn't exist - maybe there's a typo somewhere. It gives us a nice error message in that case.

Instantiating the Controller¶

Ready? On line 79, you can see the line that *actually* instantiates your controller object:

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.php
// ...

protected function createController($controller)
{
    // ...

    $controller = new $class();
    // ...
}
```

We knew that it had to happen somewhere, because our methods are non-static, and there it is. And because Symfony doesn't know anything about your `Controller` class, one of the rules - unless you register your controller as a service - is that your controller class can't have any constructor arguments. Because you can see - it just says `new $class()`.

Injecting the Container (ContainerAwareInterface)¶

The next line is really really important to just about everything you do every day in Symfony. It says: if your controller object implements the `ContainerAwareInterface`, then call `$controller->setContainer($container)`:

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.php
// ...

protected function createController($controller)
{
    // ...

    $controller = new $class();
    if ($controller instanceof ContainerAwareInterface) {
        $controller->setContainer($this->container);
    }

    return array($controller, $method);
}
```

So if I open up `DinosaurController` and click to open Symfony's base `Controller`, you'll see that it extends a `ContainerAware` class:

```
namespace Symfony\Bundle\FrameworkBundle\Controller;

use Symfony\Component\DependencyInjection\ContainerAware;
// ...

class Controller extends ContainerAware
{
    // ...
}
```

Let's click to open that. And we see that *it* implements the `ContainerAwareInterface`:

```php
// vendor/symfony/symfony/src/Symfony/Component/DependencyInjection/ContainerAware.php
namespace Symfony\Component\DependencyInjection;

abstract class ContainerAware implements ContainerAwareInterface
{
    protected $container;

    public function setContainer(ContainerInterface $container = null)
    {
        $this->container = $container;
    }
}
```

So if our controller extends Symfony's base `Controller`, we automatically implement that interface. Because of that, the `ControllerResolver` *does* call `setContainer` on our controller class, which is this function here. And what does it do? It sets that on a protected `$container` property. And *this* is the reason why in any controller function, we can say `$this->container->get()` and then get out whatever service we want:

```php
public function indexAction()
{
    $this->container->get('logger')->alert('DINOS, RUN!');
}
```

If, for some reasons, you didn't want to extend Symfony's base `Controller`, but still wanted access to the container, that would be fine: you'd just need to implement that `ContainerAwareInterface` and then have, maybe, a similar `setContainer` method that sets it on a `$container` property.

Back in `ControllerResolver`, we now have a `$controller` object, we have the `$method` that's going to be called on it, and it returns an array with those two things:

```php
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerResolver.php
// ...

protected function createController($controller)
{
    // ...

    $controller = new $class();
    if ($controller instanceof ContainerAwareInterface) {
        $controller->setContainer($this->container);
    }

    return array($controller, $method);
}
```

This is a "callable" format syntax in PHP. This ultimately goes back to the other `ControllerResolver` and is returned all the way back to `HttpKernel::handleRaw()`.

Close the FrameworkBundle `ControllerResolver` because we're done with it, but leave the other one open. Now, `$controller` is *some* callable function:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    // load controller
    if (false === $controller = $this->resolver->getController($request)) {
        throw new NotFoundHttpException(sprintf('Unable to find ...'));
    }

    // the next steps...
}
```

Inside of Symfony, it's going to be an object with a method name, but in Silex it will be an anonymous function, and it really could be anything callable.

Chapter 5: kernel.controller Event & Controller Arguments

KERNEL.CONTROLLER EVENT & CONTROLLER ARGUMENTS¶

Ok guys, what next? Hey, let's dispatch another event:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    $event = new FilterControllerEvent($this, $controller, $request, $type);
    $this->dispatcher->dispatch(KernelEvents::CONTROLLER, $event);
    $controller = $event->getController();

    // ...
}
```

This one is called `kernel.controller`. And like I keep promising, this one is passed a totally different event object called `FilterControllerEvent`. Why would you listen to this event? Well, `FilterControllerEvent` has a `getController()` method on it. So if you needed to do something based on what the controller is, or even *change* the controller in a listener, maybe to mess with your co-workers, a listener to this can do that.

But for understand the Symfony Framework, this is just a hook point. There are no mission-critical listeners to this event. So let's keep moving!

We have `$controller`, we can call it, right! Wooooh - slow down. We don't know what *arguments* to pass to the controller callable yet. To figure that out, we'll go back to `ControllerResolver` into a function called `getArguments`:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    // controller arguments
    $arguments = $this->resolver->getArguments($request, $controller);

    // ...
}
```

Open HttpKernel's `ControllerResolver` back up and scroll down to find this.

Arguments Metadata¶

This entire `if` statement is just a way to get some information about the arguments to the controller function by using PHP Reflection:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

public function getArguments(Request $request, $controller)
{
    if (is_array($controller)) {
        $r = new \ReflectionMethod($controller[0], $controller[1]);
    } elseif (is_object($controller) && !$controller instanceof \Closure) {
        $r = new \ReflectionObject($controller);
        $r = $r->getMethod('__invoke');
    } else {
        $r = new \ReflectionFunction($controller);
    }

    return $this->doGetArguments($request, $controller, $r->getParameters());
}
```

It needs the `if` statements because getting that info is different if your controller is an anonymous function, a method on an object or something different.

Ultimately, that info about the arguments is what's passed as the `$parameters` argument to `doGetArguments()`. Let me show you what I mean. Let's dump `$parameters` and put `die`:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

protected function doGetArguments(Request $request, $controller, array $parameters)
{
    dump($parameters);die;

    // ...
}
```

If we look at `DinosaurController::showAction`, we have one argument: `$id`. The route has `{id}`, so this is `$id`: we all know how those match by name:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

/**
 * @Route("/dinosaurs/{id}", name="dinosaur_show")
 */
public function showAction($id)
{
    // ...
}
```

In fact, we're about to see how that works!

So if I go to `/dinosaurs/22`, you can see that the dumped `$parameters` has just one item in it, and in that `ReflectionParameter` object is a name with `id`, since this is info about the `$id` argument:

```
array(
    0 => ReflectionObject(
        'name' => 'id',
        // ...
    )
)
```

Add a few more arguments - like `$foo` and `$bar` - and refresh:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

/**
 * @Route("/dinosaurs/{id}", name="dinosaur_show")
 */
public function showAction($id, $foo, $bar)
{
    // ...
}
```

Now we have *three* items in `$parameters`:

```
array(
    0 => ReflectionObject(
        'name' => 'id',
        // ...
    ),
    1 => ReflectionObject(
        'name' => 'foo',
        // ...
    ),
    2 => ReflectionObject(
        'name' => 'bar',
        // ...
    )
)
```

This one has `foo` and this one has `bar`. So it's just metadata about what arguments are on that controller function.

Finding Values for Each Argument¶

In `doGetArguments()`, the first thing it does is *so* important and *so* geeky cool. It goes *back* to `$request->attributes` - that same thing that was populated by the routing. Let's dump this out quickly to remember what's in there:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

protected function doGetArguments(Request $request, $controller, array $parameters)
{
    $attributes = $request->attributes->all();
    dump($attributes);die;

    // ...
}
```

When we refresh, it has `_controller` and `id` from the routing wildcard:

```
array(
    '_controller' => 'AppBundle\Controller\DinosaurController::showAction',
    'id'          => '22',
    '_route'      => 'dinosaur_show',
    '_route_params' => array(...),
)
```

It also has a couple of other things that honestly aren't very important.

Keep that array in mind. The `doGetArguments()` function iterates over `$parameters`: the array of info about the arguments to our controller. And the first thing it does is check to see if the *name* of the parameter - like `id` - exists in the `$attributes` array. And if it does, it uses that value for the argument:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

protected function doGetArguments(Request $request, $controller, array $parameters)
{
    $attributes = $request->attributes->all();
    $arguments = array();

    foreach ($parameters as $param) {
        if (array_key_exists($param->name, $attributes)) {
            $arguments[] = $attributes[$param->name];
        } elseif {
            // ...
        }
    }

    return $arguments;
}
```

This is exactly why if we have a `{id}` inside our route, its value is passed to a `$id` argument in the controller. This is *why* and how that mapping is done by *name*, because `ControllerResolver` say: “Hey, I have an `$id` argument, is there an `id` inside the `$request->attributes`, which is populated by the routing?”

The Request Argument¶

And *if* there is nothing in the attributes that matches up, it goes to the `elseif`: because there’s *one* other case. And you’ve probably seen it while doing form processing. This is when you have an argument type-hinted with the `Request` class:

```
// src/AppBundle/Controller/DinosaurController.php
// ...
use Symfony\Component\HttpFoundation\Request;

/**
 * @Route("/dinosaurs/{id}", name="dinosaur_show")
 */
public function showAction($id, Request $request)
{
    // ...
}
```

In ControllerResolver, it says: `if ($param->getClass() && $param->getClass()->isInstance($request))`:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

protected function doGetArguments(Request $request, $controller, array $parameters)
{
    $attributes = $request->attributes->all();
    $arguments = array();

    foreach ($parameters as $param) {
        if (array_key_exists($param->name, $attributes)) {
            $arguments[] = $attributes[$param->name];
        } elseif ($param->getClass() && $param->getClass()->isInstance($request)) {
            $arguments[] = $request;
        }
        // ...
    }

    return $arguments;
}
```

The `$param->getClass()` asks if the argument has a type-hint. The second part checks to see if the type-hint is for Symfony's `Request` class. If it is, the `$request` object is passed to this argument. This is *completely* special to the Request object: it doesn't work with *anything* else.

But guys, it's really pretty if you think about it. Our whole job is to read the request and create the response. So, It makes a lot of sense to be able to have the `Request` as an argument to a function that will return the `Response`. Input `Request`, output `Response`.

Errors and Seeing the Arguments in Action¶

And those are really the only two cases that work. The other `elseif` is there just to see if maybe you have an optional argument:

```
// src/AppBundle/Controller/DinosaurController.php
// ...
use Symfony\Component\HttpFoundation\Request;

/**
 * @Route("/dinosaurs/{id}", name="dinosaur_show")
 */
public function showAction($id, Request $request, $foo = 'defaultValue')
{
    // ...
}
```

And if you do, it uses the default value instead of blowing up:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Controller/ControllerResolver.php
// ...

protected function doGetArguments(Request $request, $controller, array $parameters)
{
    $attributes = $request->attributes->all();
    $arguments = array();

    foreach ($parameters as $param) {
        if (array_key_exists($param->name, $attributes)) {
            $arguments[] = $attributes[$param->name];
        } elseif ($param->getClass() && $param->getClass()->isInstance($request)) {
            $arguments[] = $request;
        } elseif ($param->isDefaultValueAvailable()) {
            $arguments[] = $param->getDefaultValue();
        } else {
            // it throws an exception
        }
    }

    return $arguments;
}
```

If all else fails, it tries to get a nice exception message to say: "Hey dude, you have an argument and I don't know what to pass to it." For example, if I add a `$bar` argument that's not optional, there's no `{bar}` in the route, so we should see this "Controller requires that you provide a value" error:

```php
// src/AppBundle/Controller/DinosaurController.php
// ...
use Symfony\Component\HttpFoundation\Request;

/**
 * @Route("/dinosaurs/{id}", name="dinosaur_show")
 */
public function showAction($id, Request $request, $foo = 'defaultValue', $bar)
{
    // ...
}
```

And we do! That's actually where that comes from.

If we get rid of that, it should pass the request to one argument and it should be ok with `$foo` being optional. When we refresh, it's happy!

Now that we're done, the array of argument values is passed all the way back to `HttpKernel`. And now we're dangerous: we have the controller callable *and* the arguments to pass to it.

Executing the Controller¶

Any ideas on what should happen next? Yep, we *finally* call the controller:

```php
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    // call controller
    $response = call_user_func_array($controller, $arguments);

    // ...
}
```

So on line 145, that's literally where your controller is executed, and it passes it all of the arguments.

And what do controllers in Symfony *always* return? They always return a `Response` object, and we also see that on line 145.

Unless... they don't return a `Response`. And that's what we're going to talk about next.

Chapter 6: The kernel.view Event

THE KERNEL.VIEW EVENT¶

We've journeyed far, brave traveler. And now that Symfony has called our controller, let's start our adventure home.

Our controller is returning a Response object. We're calling `render()` and that's a shortcut that renders the template and put it inside of a Response object:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

public function showAction($id)
{
    // ...

    return $this->render('dinosaurs/show.html.twig', [
        'dino' => $dino,
    ]);
}
```

And even though I love to say that the one job of your controller is to create and return a `Response`, it's a lie! You can return whatever you want from a controller.

And if you don't return a response, what does Symfony do? It dispatches another event:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ..

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    // call controller
    $response = call_user_func_array($controller, $arguments);

    if (!$response instanceof Response) {
        $event = new GetResponseForControllerResultEvent($this, $request, $type, $response);
        $this->dispatcher->dispatch(KernelEvents::VIEW, $event);

        if ($event->hasResponse()) {
            $response = $event->getResponse();
        }

        // ...
    }

    // ...
}
```

This time it's called `kernel.view`. The purpose of a listener to this event is to save Symfony. It really *needs* a Response object, and if the controller didn't give it to us, it calls each listener to this event and says: "Hey, here's what the controller *did* return, can you somehow transform this into a real response and save the day?"

Ok, but what's the use-case for this? In a true MVC framework, the controller is supposed to just return some data, not a response. For example, imagine if our `DinosaurController::showAction` just returned a Dinosaur object instead of the HTML response. And *then*, imagine we added a listener on this event that saw that `Dinosaur` object, rendered the template and created the Response for us.

The result would be the same, but we'd be splitting things into two pieces. The fetching and preparation of the data would happen in the controller. The creation of a representation of that data would happen in the listener.

To some of you, this might sound ridiculous. I mean, why do all this extra work? One *real* use-case is for APIs. Imagine the endpoints of your API need to be able to return both HTML or JSON depending on the `Accept` header sent by the client. In your controller, instead of having a big `if` statement for the two different formats, it just return the data:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

public function showAction($id)
{
    $dino = $this->getDoctrine()
        ->getRepository('AppBundle:Dinosaur')
        ->find($id);

    // ...

    return $dino;
}
```

Then you'd have a listener on `kernel.view` that first checks to see if the user wants HTML or JSON. If the user wants HTML, it would render the template. If it wants JSON, it would take that Dinosaur object and turn it into JSON:

```
// some imaginary listener class

public function onKernelView(GetResponseForControllerResultEvent $event)
{
    $format = $this->checkAcceptHeader($event->getRequest());
    $dino = $event->getControllerResult();
    if ($format == 'html') {
        $html = $this->templating->render('dinosaurs/show.html.twig', [
            'dino' => $dino,
        ]);

        return new Response($html);
    } elseif ($format == 'json') {
        $json = $this->serializeToJson($dino);

        return new Response($json, 200, [
            'Content-Type' => 'application/json'
        ]);
    }
}
```

The FOSRestBundle has something that does exactly this: you can return data from your controller, and it has a listener on the `kernel.view` event that transforms that data into whatever response is asked for.

Tip

If you register controllers as services, there's an additional use-case for this. See Lightweight Symfony2 Controllers.

In normal Symfony, there's nothing important that listens to this event. But when it's dispatched, it's hoping that one of those listeners will be able to set the response on the `$event` object. If we *still* don't have a `Response`, *that's* when you get the error that the controller must return a `Response`. Oh, and this is one of my favorite parts of the code: if the `$response` is actually null, and it says "Hey, *maybe* you forgot to add a return statement somewhere in your controller?":

```php
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ..

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    if (!$response instanceof Response) {
        // dispatch the event
        // ...

        if ($event->hasResponse()) {
            $response = $event->getResponse();
        }

        if (!$response instanceof Response) {
            $msg = sprintf('The controller must return a response (%s given).', $this->varToString($response));

            // the user may have forgotten to return something
            if (null === $response) {
                $msg .= ' Did you forget to add a return statement somewhere in your controller?';
            }
            throw new \LogicException($msg);
        }
    }

    // ...
}
```

Yep, I've seen that error a few times in my days.

Since *our* controller *is* returning a Response, if we go back and look at the events tab in the profiler, we'll see that there's no `kernel.view` in the list at the top. But below there's a "Not Called Listeners" section, and there *is* one listener to `kernel.view`, which comes from the SensioFrameworkExtraBundle, that's not being executed.

Chapter 7: Finishing with kernel.response and kernel.exception

FINISHING WITH KERNEL.RESPONSE AND KERNEL.EXCEPTION¶

Our controller returns a Response, so we skip the entire `kernel.view` block and go straight down to the `filterResponse()` call:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // ...

    return $this->filterResponse($response, $request, $type);
}
```

This function shows up in one other place further up. If one of our `kernel.request` listeners sets the response, the response *also* goes through this function. So no matter *how* we create the response, `filterResponse()` is called.

The kernel.response Event¶

What does it do? Come on, you should be able to guess by now. It dispatches yet another event, and this time, it's called `kernel.response`:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function filterResponse(Response $response, Request $request, $type)
{
    $event = new FilterResponseEvent($this, $request, $type, $response);

    $this->dispatcher->dispatch(KernelEvents::RESPONSE, $event);

    $this->finishRequest($request, $type);

    return $event->getResponse();
}
```

What's awesome about `kernel.response`? Listeners to it have access to the Response object by calling `getResponse()` on the `FilterResponseEvent` object. So if you want to modify the response, like adding a header, this is your event.

How the Web Debug Toolbar Works¶

In fact, the web debug toolbar works via a listener to this event. When we load up a page, the web debug toolbar shows up at the bottom. This works because at the bottom of the HTML source, there's a bunch of JavaScript that makes an AJAX request that loads it. But how does this JavaScript get there? The answer: with a listener on the `kernel.response` event. That listener injects the extra JavaScript into the HTML code of our page when we're in development mode.

So all `filterResponse()` does is give us another hook point. It also calls `finishRequest()`, which is honestly less important:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function finishRequest(Request $request, $type)
{
    $this->dispatcher->dispatch(KernelEvents::FINISH_REQUEST, new FinishRequestEvent($this, $request, $type));
    $this->requestStack->pop();
}
```

It dispatches another event and removes our request from the request stack. The request stack is an object that keeps track of all the requests we're processing. We'll talk about subrequests in a second, and then it'll make sense how Symfony can be handling multiple requests at once.

After all this, the Response object is returned *all* the way back to `app_dev.php`. We made it through Symfony's core and came back out alive with the response in hand:

```
// web/app_dev.php
// ...

$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

What do we do with it? We call `send()`. This sends all the headers with the `header()` function and echo's the content.

The kernel.terminate Event¶

The absolute last thing that happens is `$kernel->terminate()`, which is back inside HttpKernel:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

public function terminate(Request $request, Response $response)
{
    $event = new PostResponseEvent($this, $request, $response);
    $this->dispatcher->dispatch(KernelEvents::TERMINATE, $event);
}
```

Surprise! It dispatches *another* event called `kernel.terminate`. Listeners to this event are able to do work *after* the response has already been sent to the user. In other words, you can do work *after* your user is already happily seeing the page. Crazy, right?

So if you have something heavy, like sending an email, you could queue the email to be sent inside of your controller, but offload the sending to a listener on this event. The user would see the page first, and *then* the email would be sent. You have to have your web server setup correctly, but we have that all documented.

When Things go Wrong: kernel.exception¶

That's it guys - there's nothing more to see... unless something goes wrong. In the "Not Called Listeners" list, there's one more important event: `kernel.exception`. Look back at the original `handle()` function that had the try-catch block:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

public function handle(Request $request, $type = HttpKernelInterface::MASTER_REQUEST, $catch = true)
{
    try {
        return $this->handleRaw($request, $type);
    } catch (\Exception $e) {
        // ...

        return $this->handleException($e, $request, $type);
    }
}
```

You can probably guess what's about to happen. If there *was* an exception, this calls `handleException()`. This lives further below and - surprise! It dispatches an event called `kernel.exception`:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleException(\Exception $e, $request, $type)
{
    $event = new GetResponseForExceptionEvent($this, $request, $type, $e);
    $this->dispatcher->dispatch(KernelEvents::EXCEPTION, $event);

    // ...
}
```

The purpose of a listener to this event is to look at the exception object that was thrown, and somehow convert that to a `Response`. Because even if the servers are on fire, our user ultimately need a `Response`: they need to see an illustration showing that our servers are being eaten by gremlins. *Some* listener needs to create that final response for us.

And that's exactly what this code does. After dispatching the event, it checks to see if the event *has* a response and gives up if no listeners have helped out:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/HttpKernel.php
// ...

private function handleException(\Exception $e, $request, $type)
{
    // ...

    if (!$event->hasResponse()) {
        $this->finishRequest($request, $type);

        throw $e;
    }

    // ...
}
```

Core Exception Handling¶

In the real world, when an exception is thrown - like on a 404 page - we see this pretty exception page while we're developing. In the `prod` environment, we would see an error template. These responses are created by a listener to this event called `ExceptionListener`. I know, not the most creative name. Anyways, if you want to see how the exception handling works inside Symfony, you can open up this `ExceptionListener` and trace through some of its code. I won't talk about this right now, but it uses a sub-request! And that's the next topic.

To be *really* hip, you could register your *own* event listener and do whatever the heck you want with it, like showing an XKCD comic to random users on your 404 page. Ya know, get creative.

Simple: Event, Controller Event¶

Call me crazy, but when we zoom out, I think the request-response flow for Symfony is pretty darn simple. It's basically: event, controller, event.

Go back and look at the Timeline in the profiler, because now, it tells a beautiful story. At the top, we see that `kernel.request` happens first, and everything below its bar are listeners. Then, it figures out which controller we want - that's the `controller.get_callable` part. Cool. Next, it dispatches `kernel.controller`, and you can see *its* listeners. After that, the `controller` is called and the stuff under that is *our* work. We can see some Doctrine calls we're making and the time it takes to render the template. After the controller, the `kernel.response` event is dispatched, it has a few listeners, and there's `kernel.terminate`. Brilliant!

So guys, it's just events, call the controller, then more events. Yep, that's it. And now that we've journeyed to the core of Symfony's request and response flow, let's bend this to our will to do some crazy, custom things.

Chapter 8: Symfony Magic: Replace the _controller

SYMFONY MAGIC: REPLACE THE _CONTROLLER¶

If this were the Matrix, you'd be seeing 1's and 0's running across your code. Your Symfony world is now one without rules or controls, without borders or boundaries. Let's prove it.

We know that Symfony reads the `_controller` key in the request attributes and executes that as the controller. This is set by the `RouterListener` but it could be set anywhere.

Let's go adventuring!. `UserAgentSubscriber` listens to `kernel.request`. What if we just replaced the `_controller` key here with something else? Let's do that, and set it to an anonymous function:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...
use Symfony\Component\HttpFoundation\Response;

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    $request->attributes->set('_controller', function() {
        return new Response('Hello Dinosaur!');
    });
}
```

Symfony doesn't see any difference between this function and a traditional controller. To prove it, return a normal `Response`:

```
return new Response('Hello world!')
```

When we refresh, would you believe it? That actually works! Comment this line out.

Forcing RouterListener NOT to Route¶

Even within listeners to a single event, there is an order of things. In the events tab of the Profiler, you can see that our listener is the *last* one that's called by `kernel.request`. The `RouterListener` is executed before us. That means that the routing *is* being run and the `_controller` key *is* being set. Later, our listener replaces it.

But even if our listener were run before the router, our little hack would still work. That's because `RouterListener` has a special piece of code near the top that checks to see if the `_controller` key is already set. If it *is* set somehow, the routing never runs:

```
// vendor/symfony/symfony/src/Symfony/Component/HttpKernel/EventListener/RouterListener.php
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    if ($request->attributes->has('_controller')) {
        // routing is already done
        return;
    }

    // ...
}
```

That'll be important when we talk about sub-requests.

And since our anonymous function is a valid controller, we can still use arguments on it just like normal. So if I go to `/dinosaurs/22`, which has a `{id}` in the route, we're allowed to have an `$id` argument:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...
use Symfony\Component\HttpFoundation\Response;

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    $request->attributes->set('_controller', function($id) {
        return new Response('Hello '.$id);
    });
}
```

So this will work just fine. All controllers are created equal.

Now, let's comment that out temporarily, or more likely permanently:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...
use Symfony\Component\HttpFoundation\Response;

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    /*
    $request->attributes->set('_controller', function($id) {
        return new Response('Hello '.$id);
    });
    */
}
```

Chapter 9: Making an Argument Available to All Controllers

MAKING AN ARGUMENT AVAILABLE TO ALL CONTROLLERS¶

Here's our next challenge: pretend that it's really important to know if the user is on a Mac or not. We're really wanting to start measuring how hipster our user base is. In fact, it's *so* important, that we want to be able to have an `$isMac` argument in *any* controller function *anywhere* in the system. This *won't* come from a routing wildcard like normal - we'll figure this out by reading the `User-Agent`.

As an example, I'm going to put `$isMac` into `indexAction`:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

public function indexAction($isMac)
{
    // ...

    return $this->render('dinosaurs/index.html.twig', [
        'dinos' => $dinos,
        'isMac' => $isMac
    ]);
}
```

We'll pass that into our template, and inside there, use it to print out a threatening message if the user is on a Mac:

```
{# app/Resources/views/dinosaurs/index.html.twig #}
{# ... #}

{% if isMac %}
    <h3>We love eating Mac's...RAWR!</h3>
{% endif %}
```

Back to the homepage! If we try this now, we know what's going to happen. We get a huge error because there is no `{isMac}` in the route. Symfony has no idea what to pass to the argument. And up until now that's been the rule: whatever we have in our routing curly brace is available as an argument and there are no exceptions to that rule, except for the Request object.

Allowing Other Controller Arguments¶

But guys, that's not true! And we know it. We know that it's not *really* about the routing layer. The arguments to the controller come from the request attributes. And sure, the only thing that normally modifies those is the routing layer. But there's nothing stopping someone else from adding some extra stuff.

In our subscriber, let's first get an `$isMac` variable. We'll look for the `User-Agent` header and we'll look for the word `mac`. Check to see if `stripos` doesn't equal false:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

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

To make this available as an argument, all we need to do is put it in the request attributes:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    $isMac = stripos($userAgent, 'Mac') !== false;
    $request->attributes->set('isMac', $isMac);
}
```

Seriously, that's it. When we refresh, it works!

And since I never trust when things work on the first try, let's change our code to look for `mac2`. I'm on a mac, but the message hides since we changed that. Go ahead and change that back

The Person Behind the Curtain (Event Listeners)¶

So why is this important? Because understanding the core of Symfony is letting you do things that previously looked impossible. You're also going to be able to figure out how magic from outside libraries is working.

For example look at the SensioFrameworkExtraBundle. This is basically a bundle of shortcuts that work via magic. Now that you've journeyed to the center of Symfony and back, if you look at each shortcut, you should be able to explain the magic behind each of these.

The one I want to look at now is the ParamConverter:

```
/**
 * @Route("/blog/{id}")
 * @ParamConverter("post", class="SensioBlogBundle:Post")
 */
public function showAction(Post $post)
{
}
```

In the example, you can see that the controller has a `$post` argument, but also that there's no `{post}` in the routing. So this *should* throw an error. The `ParamConverter`, which works via a listener to `kernel.controller`, grabs the `id` off of the request attributes, queries for a `Post` object via Doctrine with that `id`, and then adds a new request attribute called `post` that's set to that object:

```
// Summarized version of ParamConverterListener
public function onKernelController(FilterControllerEvent $event)
{
    $request = $event->getRequest();
    $id = $request->attributes->get('id');

    $entity = $this->em->getRepository('SensioBlogBundle:Post')
        ->find($id);
    if (!$entity) {
        throw new NotFoundHttpException('No Post found for '.$id);
    }

    $request->attributes->set('post', $entity);
}
```

And just by doing that, the `showAction` can have that `$post` argument.

If that makes any sense at all, you're on the verge of *really* mastering a big part of Symfony.

Before we talk about sub requests, I want to point something out. If you're playing with things, inside the profiler, there is a

Request tab, which is interesting because it shows you the request `attributes`. You can see the `_controller`, the routes stuff and the `isMac` key. By the way, not that it's necessarily useful, but the fact that there is an `_route` key *does* mean that you can have a `$_route` argument to any controller.

Chapter 10: What about Sub Requests?

WHAT ABOUT SUB REQUESTS?¶

Have you ever been in a Twig template, and you suddenly need access to a variable you don't have? Open up `base.html.twig`, and let's pretend that we want to render the latest tweets from our Twitter account at the bottom.

Since there's no *one* controller that fuels this template, there's no way for us to pass the latest tweets to this template. When you're in this spot, there are 2 fixes. First, you could create a new Twig function in a Twig extension. That's usually the best approach. The second option is with the Twig `render()` function, which is our gateway to sub requests! This will also let you cache this chunk of HTML, but we'll talk about that another time.

When we use `render(controller('...'))`, this lets us execute a totally different controller. Let's render a new controller called `_latestTweetsAction` in our same controller class:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{{ render(controller('AppBundle:Dinosaur:_latestTweets')) }}
```

I put the underscore in front of the name just as a reminder to myself that the controller only returns a fragment of HTML, not a full page.

In the controller, create the `public function _latestTweetsAction`. I don't *really* want to bother with Twitter's API right this second, so I'll copy in some fake tweets instead. Now let's render a template and pass those in:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

public function _latestTweetsAction()
{
    $tweets = [
        'Dinosaurs can have existential crises too you know.',
        'Eating lollipops... ',
        'Rock climbing... '
    ];

    return $this->render('dinosaurs/_latestTweets.html.twig', [
        'tweets' => $tweets
    ]);
}
```

Yea, this is only going to return a page fragment, we still *always* return a Response object from a controller. And that's what we're doing here. Now, let's create this template, `_latestTweets.html.twig`. I'll just paste some code in that loops over these tweets and puts them in a list.

```
{# app/Resources/views/dinosaurs/_latestTweets.html.twig #}

<div class="navbar-left tweets">
    <p class="text-center">Tweets from T-Rex Problems</p>
    <ul>
        {% for tweet in tweets %}
            <li>{{ tweet }}</li>
        {% endfor %}
    </ul>
</div>
```

Nothing scary here!

The Twig `render()` function will call out to that function, get the Response, grab its content, then put it at the bottom of the page. Scroll down - there it is!

OMG: You just made a Sub-Request!¶

Why did I show this? Why is this important to understanding the core? Open up the profiler and go back to the Timeline. Hmm, it looks normal at first and we can see where our controller and template are run. But hey, what's that darker bar behind the template? Scroll way down to find a *second* request called a sub-request. OMG!

The sub-request is totally independent: it goes through the entire same process we just learned. It executes the `kernel.request` listeners, it dispatches the `kernel.controller` event, it calls the `_latestTweetsAction` controller and has `kernel.response` on the end. It really *is* like we are handling two totally separate request-response cycles.

Heck, you can even click to see the profiler for *just* the sub-request.

Request Attributes in Sub-Request Controllers¶

Remember that in `UserAgentSubscriber`, we're adding `isMac` to the request attributes and that means it's available as an argument to *any* controller. That's no different with our sub request controller, since this listener is called for that request too. To prove it, I'm going to add the `$isMac` argument. Let's pass this into our template:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

public function _latestTweetsAction($isMac)
{
    // ...

    return $this->render('dinosaurs/_latestTweets.html.twig', [
        'tweets' => $tweets,
        'isMac' => $isMac
    ]);
}
```

Let's print it out to make sure it's working:

```
{# app/Resources/views/dinosaurs/_latestTweets.html.twig #}

<p class="text-center">{{ isMac ? 'on a Mac' : 'Not on a Mac' }}</p>
{# ... #}
```

When we go back and refresh we see that on the top it shows that we're on a Mac, and on the bottom inside the Tweets area, we're on a Mac too! Yay, no surprises!

A Disturbance in the Request¶

Here is where things get crazy. Go back to `UserAgentSubscriber`. Let's add an override so it's easier for us to play with this "is Mac" stuff, since I'm pretty permanently using one.

If there's a query parameter, called `notMac`, that's set to some value like 1, then let's always set `$isMac` to false:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    // ...

    $isMac = stripos($userAgent, 'Mac') !== false;
    if ($request->query->get('notMac')) {
        $isMac = false;
    }
    $request->attributes->set('isMac', $isMac);
}
```

Back on the browser, when I refresh, I'm still on a Mac. But if I add a `?notMac=1` to the URL, it goes away. The override correctly makes it look like I'm not on a Mac.

Now scroll down. Woh! The sub request *still* thinks we're on a Mac. Something just short circuited in the system. But before we fix it, let's dive one level deeper and see how sub-requests really work.

Chapter 11: How Sub-Requests Work

HOW SUB-REQUESTS WORK¶

To learn how sub request *really* work, let's leave this behind for a second and go back to `DinosaurController::indexAction`. I'm going to create a sub request right here, by hand. To do that, just create a `Request` object. Next, set the `_controller` key on its attributes. Set it to `AppBundle:Dinosaur:latestTweets`:

```
// AppBundle/Controller/DinosaurController.php
// ...

public function indexAction($isMac)
{
    // ...

    $request = new Request();
    $request->attributes->set(
        '_controller',
        'AppBundle:Dinosaur:_latestTweets'
    );

    // ...
}
```

Then, I'm going to fetch the `http_kernel` service. Yep, that's the same HttpKernel we've been talking about, and it lives right in the container.

Now, let's call the familiar `handle` function on it: the exact same `handle` function we've been studying. Pass it the `$request` object and a `SUB_REQUEST` constant as a second argument. I'm going to talk about that constant in a second:

```
// AppBundle/Controller/DinosaurController.php
// ...

public function indexAction($isMac)
{
    // ...

    $request = new Request();
    $request->attributes->set(
        '_controller',
        'AppBundle:Dinosaur:_latestTweets'
    );
    $httpKernel = $this->container->get('http_kernel');
    $response = $httpKernel->handle(
        $request,
        HttpKernelInterface::SUB_REQUEST
    );

    // ...
}
```

Let's think about this. We're *already* right in the middle of an `HttpKernel::handle()` cycle for the main request. Now, we're starting another `HttpKernel::handle()` cycle. And because I'm setting the `_controller` attribute it knows which controller to execute. That sub request is going to go through that whole process and ultimately call `_latestTweetsAction`.

I'm not going to do anything with this `Response` object: I'm just trying to prove a point. If we refresh the browser and click into the Timeline, we now have *two* sub requests. One of them is the one I just created. If you scroll down you can see that. The other one is coming from the `render()` call inside the template.

Let's comment this silliness out. I wanted to show you this because that's *exactly* what happens inside `base.html.twig` when we use the `render()` function. It creates a brand new request object, sets the `_controller` key to whatever we have here, then passes it to `HttpKernel::handle()`.

Sub-Requests have a Different Request Object¶

Now we know why we're getting the weird `isMac` behavior in the sub request! The `UserAgentSubscriber` - in fact all listeners - are called on both requests. But the second time, the request object is **not** the *real* request. It's just some empty-ish Request object that has the `_controller` set on it. It doesn't, for example, have the same query parameters as the main request.

That's why the first time `UserAgentSubscriber` runs, it reads the `?notMac=1` correctly. But the second time, when this is run for the sub request, there are *no* query parameters and the override fails.

Properly Handling Sub-Request Data¶

Here's the point: when you have a sub request, you need to *not* rely on the information from the main request. That's because the request you're given is **not** the real request. Internally, Symfony duplicates the main request, so some information remains and some doesn't. That's why reading the `User-Agent` header worked in the sub request. But don't rely on this: think of the sub-request as a totally independent object.

So whenever you read something off of the request, you need to ask yourself... do you feel lucky?... I mean... you need to make sure you're working with what's called the "master" request.

This means that our `UserAgentSubscriber` can only do its job properly for the master request. On a sub-request, it shouldn't do anything. So let's add an `if` statement and use an `isMasterRequest()` method on the event object. If this is *not* the master request, let's do nothing:

```
// src/AppBundle/EventListener/UserAgentSubscriber.php
// ...

public function onKernelRequest(GetResponseEvent $event)
{
    if (!$event->isMasterRequest()) {
        return;
    }
    // ...

}
```

And how does Symfony know if it's handling a master or sub-requests? That's because the second argument here. So when you call `HttpKernel::handle()`, we pass in a constant that says "hey this is a sub request, this is not a master request". And then your listeners can behave differently if they need to.

When we refresh, we get a huge error! This makes sense. `UserAgentSubscriber` doesn't run on the sub-request, so `isMac` is missing from the request attributes. And because of that, we can no longer have an `$isMac` controller argument.

Passing Information to a Sub-Request Controller¶

But wait! I *do* want to know if the user is on a Mac from my sub request. What's the solution?

The answer is really simple: just pass it to the controller. The second argument to the `controller()` function is an array of items that you want to make available as arguments to the controller. Behind the scenes, these are put onto the attributes of the sub request. So we can add a `userOnMac` key and set its value to the *true* `isMac` attribute stored on the master request. So, `app.request.attributes.get('isMac')`:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{{ render(controller('AppBundle:Dinosaur:_latestTweets', {
    'userOnMac': app.request.attributes.get('isMac')
})) }}
```

Inside of the controller, add a `userOnMac` variable and pass it into the template:

```
// src/AppBundle/Controller/DinosaurController.php
// ...

public function _latestTweetsAction($userOnMac)
{
    // ...

    return $this->render('dinosaurs/_latestTweets.html.twig', [
        'tweets' => $tweets,
        'isMac' => $userOnMac
    ]);
}
```

Now when we refresh, we still have the `?notMac=1`, so the Mac message is missing from the master request part at the top. And if we scroll down, the sub request *also* knows we're not on a mac because we're passing that information through.

When we take off the query parameter, it looks like we're on a mac up top on the bottom. Brilliant!

The lesson is that you need to be careful *not* to read outside request information, like query parameters from the URL, from inside a sub-request. This also ties into Http caching and ESI which are topics we'll cover later. If we follow this rule and you *do* want to cache this latest tweets fragment, it's going to be super easy.

Seeya next time!