Drupal 8: Under the Hood

With <3 from SymfonyCasts

Chapter 1: Installation, Composer and Git!

Hello Drupal people! I'm Ryan, and I come from the magical world of Symfony, full of gumdrops, rainbows, interfaces, services, dependency injection and lollipops. Along with a few other oompa loompas, I lead the Symfony documentation team, so I may not seem like the most obvious person to be teaching you about Drupal 8. But Drupal 8 has taken a *huge* leap forward by using common coding patterns and libraries. This makes Drupal a lot easier and more accessible to a lot of people.

This series is meant for developers who have used Drupal before. Instead of learning how to use it, we're going to rip apart the layers and see how this machine runs. That's going to make you more dangerous and uncover possibilities you wouldn't otherwise know about.

Download D8

Start by downloading Drupal 8, which at this moment isn't quite released, but it will *very* soon! This unzips the file to my Downloads directory. I'll move it to a drupal8 directory. We can see all of our shiny new files here in my shiny PhpStorm editor.

The Built-in PHP Web Server

Move into the drupal8 directory. We need a webserver! But I'm not going waste time setting up Apache or Nginx locally. Instead, I'll use the built-in PHP web server. Start it by running php -S localhost:8000:

```
$ php -S localhost:8000
```

This serves files from *this* directory and will hang there until you stop it. I highly recommend using this to develop.

In the browser, navigate to http://localhost:8000. Hello Drupal 8 install screen! Pick the standard installation to get a few more features.

Fixing php.ini problems

On the next step, I have a problem: the xdebug.max_nesting_level setting in php.ini is set too low. Wah wah.

Bah, it's easy to fix. Go back to the terminal and open a new tab. Run php --ini:

```
$ php --ini
```

This will tell you *where* the php.ini file lives. Open it with your favorite editor. I like vim, because it gives me street cred.

Tip

In some setups (I'm looking at you OSX), there will be *no* value for "Loaded Configuration File". Usually, there *is* a file in the "Configuration File (php.ini) Path" directory, but it's named something like php.ini.development. Rename this file to php.ini and run php --ini again.

Search for the setting! It already exists in my file, so I'll set it to 256. If it doesn't exist in your file, just add at the bottom:

```
xdebug.max_nesting_level = 256
```

For this change to take effect, restart your web server. For us, hit control+c to kill the PHP web server and then start it again:

```
$ php -S localhost:8000
```

That fixes it! Type in your database details: I'll call my database d8_under_hood and pass root with no password for my super secure local computer.

Now go grab some coffee or a sandwich while Drupal does it's installation thing.

Ding! Give your site a clever name and an email address. Um, but enter *your* email, not mine. The super-secret and secure password I'm using is admin. Select your country and hit save.

Phew! I mean congrats! You now have a working Drupal 8 site!

Storing in Git and talking Composer

You know what I love most about a new project? Creating a new git repo. Seriously, how often do you get to type git init?

```
$ git init
```

In PhpStorm, you can see an example.gitignore file. Refactor-Rename that to .gitignore. Open it and uncomment out the vendor line to ignore that directory:

37 lines | .gitignore

```
... lines 1 - 9
# Ignore core and vendor when managing dependencies with Composer.
# core
/vendor/
... lines 13 - 37
```

The project also has composer.json and composer.lock files:

45 lines | composer.json

```
{
  "name": "drupal/drupal",
... lines 3 - 5
  "require": {
    "composer/installers": "^1.0.21",
... line 8
  },
... lines 10 - 32
  "autoload": {
    "psr-4": {
      "Drupal\\Core\\Composer\\": "core/lib/Drupal/Core/Composer"
    }
  },
... lines 38 - 43
}
```

Composer is PHP's package manager, and it has changed *everything* in our world. If you aren't familiar with it, go watch our Composer tutorial! Seriously, you can use it in Drupal 7... we do in that tutorial...

Because of the composer.json file, you should *not* need to commit the vendor/ directory. You should also not need to commit the core/ directory where all of Drupal lives, due to some special Composer setup in Drupal. Another developer should be able to clone the project, run composer install and both vendor/ and core/ will be downloaded for them.

When I tried to do that, I had a little trouble with the core/ directory due to an autoloading quirk. Hey, it's not released yet, so there could be a bug. It's cool.

In another screencast, I'll show you the proper way to use Composer with Drupal. But for now it's safe to *not* commit the vendor/ directory at least. If you run composer install, it'll populate that directory correctly.

Zip back over to the terminal and run git add . and then git status:

```
$ git add .
$ git status
```

There are a lot of files in core/, so it *will* be nice to not have to commit those someday. But other than these core/ files, we're not committing much. A new Drupal "project" doesn't contain many files.

Finish this by typing git commit and typing in a clever commit message for your fellow contributors to enjoy. Done!

Please, Please use a Decent Editor

I have a secret to tell you that will make your Drupal 8 experience many times better: use a decent editor, the best is PhpStorm. Atom and Sublime are also pretty good. But if you use Notepad++ or open some directory explorer to dig for files manually, there will be *no* rainbows, Pixy Sticks or Gumball drops in your Drupal 8 experience. Your editor must be able to auto-complete, have a directory tree and have a keyboard shortcut to open files by filename. Ok, I've warned you!

PhpStorm Symfony Plugin = Joy

If you *do* use PhpStorm... which would make you my best friend... it has a Symfony plugin that plays nicely with Drupal too. Score! In Preferences, under plugins, click browse repositories and search for "Symfony". You'll find this awesome Symfony plugin that has over 1.3 million downloads! If you don't have this installed yet, do it. I already have it. After installing, it'll ask you to restart PhpStorm. Once it's open again, head back to Preferences, search for Symfony, and you'll find a new Symfony plugin menu. Make sure you check the Enable Plugin for this project box. Remember to check this for each new project.

This plugin will give you some pretty sweet autocompletion that's specific to Drupal and Symfony.

Sweet! We're up and running! Let's get into the code!

Chapter 2: Modules, Routes and Controllers

Creating a Module

Let's do something fun, like create a custom page. Like always, any custom code will live in a module. And modules live in the modules/ directory. Create a new one called dino_roar. To make Drupal fall in love with your module, create the info file: dino_roar.info.yml. If you loved the old .info files, then you'll feel all warm and fuzzy with these: it's the same thing, but now in the YAML format.

Inside give it a name: Dino ROAR, a type: module, description: Roar at you, package: Sample and core: 8.x:

6 lines | modules/dino_roar/dino_roar.info.yml

```
1  name: Dino ROAR
2  type: module
3  description: "ROOOOOAR at you"
4  package: Sample
5  core: 8.x
```

If YAML is new to you, cool! It's pretty underwhelming: just a colon separated key-value pair. But make sure you have at least one space after the colon. Yaml also supports hierarchies of data via indentation - but there's none of that in this file.

Module ready! Head back to the browser and go into the "Extend" section. With any luck we'll see the module here. There it is under "Sample": "Dino ROAR". It sounds terrifying. Check the box and press the install button anyways. What's the worst that could happen?

Nothing! But now we can build that page I keep talking about.

Add a Route

In any modern framework - and I *am* including Drupal in this category, yay! - creating a page is two steps. First, define the URL for the page via a *route*. That's your first buzzword in case you're writing things down.

Second, create a *controller* for that page. This is a function that you'll write that actually builds the page. It's also another buzzword: controller.

If these are new buzzwords for you, that's ok - they're just a new spin on some old ideas.

For step 1, create a new file in the module: dino_roar.routing.yml. Create a new route called dino_says: this is the internal name of the route and it isn't important yet:

7 lines | modules/dino_roar/dino_roar.routing.yml

```
1  dino_says:
   ... lines 2 - 7
```

Go in 4 spaces - or 2 spaces, it doesn't matter, just be consistent - and add a new property to this route called path. Set it to /the/dino/says: the URL to the new page:

7 lines | modules/dino_roar/dino_roar.routing.yml

```
1  dino_says:
2      path: /the/dino/says
   ... lines 3 - 7
```

Below path, a few more route properties are needed. The first, is defaults, with a _controller key beneath it:

7 lines | modules/dino_roar/dino_roar.routing.yml

```
1  dino_says:
2      path: /the/dino/says
3      defaults:
4          _controller: Drupal\dino_roar\Controller\RoarController::roar
   ... lines 5 - 7
```

The _controller key tells Drupal which *function* should be called when someone goes to the URL for this exciting page. Set this to Drupal\dino_roar\Controller\RoarController::roar. This is a namespaced class followed by :: and then a method name. We'll create this function in a second.

Also add a requirements key with a _permission key set to access content:

7 lines | modules/dino_roar/dino_roar.routing.yml

```
1  dino_says:
   ... lines 2 - 4
5      requirements:
6          _permission: 'access content'
```

We won't talk about permissions now, but this is what will allow us to view the page.

In YAML, you usually don't *need* quotes, except in some edge cases with special characters. But it's always safe to surround values with quotes. So if you're in doubt, use quotes! I don't need them around access content... but it makes me fee good.

Add a Controller Function

Step 1 complete: we have a route. For Step 2, we need to create the controller: the function that will actually build the page. Inside of the dino_roar module create an src directory and then a Controller directory inside of that. Finally, add a new PHP class called RoarController:

14 lines | modules/dino_roar/src/Controller/RoarController.php

```
1  <?php
   ... lines 2 - 14
```

Ok, stop! Fun fact: *every* class you create will have a namespace at the top. If you're not comfortable with namespaces, they're really easy. So easy that we teach them to you in 120 seconds in our namespaces tutorial. So pause this video, check that out and then everything we're about to do will seem much more awesome.

But you can't just set the namespace to *any* old thing: there are rules. It must start with Drupal\, then the name of the module - dino_roar\, then whatever directory or directories this file lives in after src/. This class lives in Controller. Your class name also has to match the filename, + .php:

14 lines | modules/dino_roar/src/Controller/RoarController.php

```
1  <?php
2
3  namespace Drupal\dino_roar\Controller;
   ... lines 4 - 6
7  class RoarController
8  {
   ... lines 9 - 12
13 }
```

If you mess any of this up, Drupal isn't going to be able to find your class.

The full class name is now Drupal\dino_roar\Controller\RoarController. Hey, this conveniently matches the _controller of our route!

In RoarController, add the new public function roar():

14 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 6
class RoarController
{
    public function roar()
    {
... line 11
    }
}
```

A Controller Returns a Response

Now, you might be asking yourself what a controller function like this should return. And to that I say - excellent question! Brilliant! A controller should *always* return a Symfony Response object. Ok, that's not 100% true - but let me lie for just a *little* bit longer.

Tip

The code-styling (4 spaces indentation, etc) I'm using is called PSR-2. It's a great PHP standard, but is (I admit) different than the recommended Drupal standard.

To return a response, say return new Response(). I'll let it autocomplete the Response class from Symfony's HttpFoundation namespace. When I hit tab to select this, PhpStorm adds the use statement to the top of the class automatically:

14 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 4
use Symfony\Component\HttpFoundation\Response;

class RoarController
{
    public function roar()
    {
        return new Response('ROOOOOAR!');
    }
}
```

That's important: whenever you reference a class, you *must* add a use statement for it. If you forget, you'll get the famous "Class Not Found" error.

For the page content, we will of course ROOOAR!.

That's it! That's everything. Go to your browser and head to /the/dino/says:

> http://localhost:8000/the/dino/says

Hmm page not found. As a seasoned Drupal developer, you may be wondering, "uhh do I need to clear some cache?" My gosh, you're right!

Chapter 3: The Drupal Console & Route Cache

The Drupal Console!

Google for a new utility called "Drupal Console". This is a *fantastic* console script that helps you debug, clear cache and generate code. If you love it, you should say thank you to Jesus Olivas and others for their work! It's a bit like Drush, but different, but kind of the same... I don't know. They seem to be co-existing and don't do the exact same things.

To download it, copy the curl statement and run that in your terminal. Next, move it into a global bin directory so that you can simply type drupal from anywhere in the terminal:

```
$ curl https://drupalconsole.com/installer -L -o drupal.phar
$ mv drupal.phar /usr/local/bin/drupal
$ chmod +x /usr/local/bin/drupal
$ drupal
```

Tip

If you're on windows, use the readfile line and don't worry about moving the file into a bin/ directory. Instead, you can type php drupal.phar to run the drupal.phar file that is downloaded.

Hello Drupal Console! Now try drupal list:

```
$ drupal list
```

This shows a *huge* list of all of the commands you can run. There is a lot of really good stuff in here - we'll cover some of these in this tutorial.

Clearing the Routing Cache

One of those commands is router:rebuild. Run that to clear the routing cache:

```
$ drupal router:rebuild
```

Ok, go back, refresh and congratulations!!! Seriously: you've just created your first custom page in Drupal 8. By the way, creating a page in the Symfony framework looks almost exactly the same. You're mastering two tools at once! You deserve a vacation.

Notice this page is *literally* only the text "ROOOOAR". It doesn't have any theming or templates applied to it. We *will* tap into the theme system later, but this is really interesting: if you want to return a Response all by yourself, you can do that and Drupal won't mess with it. Hey, this could even be a JSON response for an API. Drupal is a CMS, but it's also a modern, custom-development framework.

Chapter 4: Routing Wildcards

Routing is packed with cool little features, but the most common thing you'll see is the addition of wildcards. Add a /{count} to the end of the route's path:

7 lines | modules/dino_roar/dino_roar.routing.yml

```
1  dino_says:
2      path: /the/dino/says/{count}
   ... lines 3 - 7
```

Because this is surrounded with curly-braces, the route will now match /the/dino/says/*anything*,

As soon as you have a routing wildcard called count, you can suddenly have a $count argument in your controller function:

15 lines | modules/dino_roar/src/Controller/RoarController.php

```
   ... lines 1 - 6
7  class RoarController
8  {
9      public function roar($count)
10     {
   ... lines 11 - 12
13     }
14 }
```

The value for for the {count} part in the URL is passed to the $count argument. Both the wildcard and the argument must have the same name.

Use $count to change our scary greeting: $roar = 'R'.str_repeat('0', $count).'AR!'. Pass this to the Response:

15 lines | modules/dino_roar/src/Controller/RoarController.php

```
   ... lines 1 - 8
9      public function roar($count)
10     {
11         $roar = 'R'.str_repeat('O', $count).'AR!';
12         return new Response($roar);
13     }
   ... lines 14 - 15
```

We just changed the route configuration, so we need to rebuild the routing cache:

```
$ drupal router:rebuild
```

Once I do, the /the/dino/says page returns a 404! Ah! As soon as you add /{count} to the route path, the route only matches when *something* is passed there. We need /the/dino/says/*something*, anything but blank. There *are* ways to make the wildcard optional - check out the Symfony routing docs.

Add 10 to the end of the URL:

> http://localhost:8000/the/dino/says/10

Roooooooooooar! Make it 50 and the roooooooooooo....ooooooooar gets longer.

Woah! You now know half of the new stuff in Drupal 8. It's this routing/controller layer. Make a route, then a controller and make that controller return a Response. Seriously, can we go on vacation now?

Chapter 5: Debugging!

Debugging in Drupal 8 is pretty sweet... if you know the tools to use.

Turning on Debugging

First, let's activate a debug mode so we can see errors... *just* in case some *other* developer makes a mistake and we have to debug it.

In sites/ there's an example.settings.local.php file that can be used to activate development settings locally. But first, we need to play with permissions: Drupal makes some files in this directory readonly for security. Start by making sites/default writable by us:

```
$ chmod 755 sites/default
```

Now, copy sites/example.settings.local.php to sites/default/settings.local.php:

```
$ cp sites/example.settings.local.php sites/default/settings.local.php
```

Finally, make sure we can write to settings.php:

```
$ chmod 644 sites/default/settings.php
```

Tip

On a production server, it's best to make sure these files are *not* writeable by whatever user runs your web server.

The settings.local.php file activates several things that are good for debugging, like a verbose error level. It also loads a development.services.yml file that we're going to talk about soon:

92 lines | sites/default/settings.local.php

```
... lines 1 - 31
/**
 * Enable local development services.
 */
$settings['container_yamls'][] = DRUPAL_ROOT . '/sites/development.services.yml';

/**
 * Show all error messages, with backtrace information.
 *
 * In case the error level could not be fetched from the database, as for
 * example the database connection failed, we rely only on this value.
 */
$config['system.logging']['error_level'] = 'verbose';
... lines 44 - 92
```

But just having settings.local.php isn't enough! Open settings.php. At the bottom, uncomment out the lines so that this file is loaded:

727 lines | sites/default/settings.php

```
... lines 1 - 701
/**
 * Load local development override configuration, if available.
 *
 * Use settings.local.php to override variables on secondary (staging,
 * development, etc) installations of this site. Typically used to disable
 * caching, JavaScript/CSS compression, re-routing of outgoing emails, and
 * other things that should not happen on development and testing sites.
 *
 * Keep this code block at the end of this file to take full effect.
 */
if (file_exists(__DIR__ . '/settings.local.php')) {
  include __DIR__ . '/settings.local.php';
}
... lines 715 - 727
```

Of course, we need to rebuild our cache and the Drupal Console in all its wisdom has a command for that:

```
$ drupal cache:rebuild
```

From this command you can clear all kinds of different caches, like the menu cache, render cache or leave it blank to do everything.

Side note: the Drupal Console app is built on top of the Symfony Console library, and you can take advantage of it to create really cool command line scripts like this if you want to. It's one of the *best* pieces of Symfony.

Back in the browser, if you refresh now, there's no difference: everything is roaring along. But, try deleting the roar() function and refresh. Now we get a nice error that says:

> The controller for URI /the/dino/says/50 is not callable.

That's a way of saying "Yo! This route is pointing to a function that doesn't exist!" And when we put the function back and refresh, things are back to a roaring good time.

List *all* the Routes

Every page of the site - including the homepage and admin areas - has a route. And that leads me to the next natural question: what time is dinner? I mean, can I get a list of every single route in the system? Well, of course! And dinner is at 6.

Once again, go back to the trusty Drupal Console app. To list *every* route, run router:debug:

```
$ drupal router:debug
```

Wow! This prints *every* single route in the system, which is *wonderful* when you're trying to figure out what's going on in a project. This includes the admin route and our custom route.

To get more information about a route, copy it's internal name - that's the part on the left - and pass it as an argument to router:debug. This route has several curly brace routing wildcards that are passed to the getForm method of this controller class. This is pretty sweet, but we can go further!

Chapter 6: The webprofiler

Google for "Drupal Devel": it will lead you to a project on Drupal.org that has some nice tools in it. What *we're* after is the webprofiler module. Copy the installation link address so we can install it! In the future, you should be able to install modules via Composer. More on that later.

For now, click install new module, paste the URL, press the install button and wait with great anticipation! When it finishes, follow the enable newly added modules link. I'm only interested in the web profiler. Check its box and press the install button. It'll also ask me to install the devel, module which is fine. Hit continue... and watch closely.

The Web Debug Toolbar

On the very next request a really cool web debug toolbar pops up at the bottom. This is the oracle of information about the request that was just executed - in this case - a request to this admin page. It shows us tons of stuff like database queries, who is authenticated, stats about the configuration, the css and js that's being loaded, the route and controller that's being used for this page and - somewhere in there - I'm pretty sure it knows where my car keys are.

Before you go crazy with this, go back down on this page to the webprofiler. Expand it and click "Configure". Here, you can see that there's even *more* information that you can display if you want to. Check the boxes for Events, Routing and Services and then press the "Save configuration" button.

The Profiler

Ooo look: a few new things on the toolbar. If you click any icon, you'll go to a new page called the profiler. It turns out that the web debug toolbar was just a short summary of the information about the last request. The profiler has *tons* more.

If you want to see all of the routes, click the Routing tab. This is the same list that the Drupal console showed us.

There are lots of other treats in here: it's like a Drupal Halloween! For example, Performance Timing checks how fast the frontend of your site is rendering. The timeline is probably my *favorite*... and for some reason it's broken in this version. Wah wah. It normally shows you this great graph of how long it takes each part of Drupal to execute. It's great for profiling, but also great to see what all the magic layers of Drupal are.

If you follow the documentation for the webprofiler, you also need to install a couple of JavaScript libraries to help the profiler do its job. But it seems to work pretty well without them, so I skipped that part to save us time.

Reverse Engineering an Admin Page

Now that we have this, click on the admin "Structure" page. Obviously, this page comes from Drupal. But how does it work? Go down to the toolbar and hover over the 200 status code. Ha! This tells us exactly what controller renders this page:

> Drupal\System\Controller\SystemController::systemAdminMenuBlockPage()

If you see the D\s\C stuff, that stands for Drupal\System\Controller. The web profiler tries to shorten things: just Hover over this syntax to see the full class name.

If you wanted to reverse engineer this page, you could! I'll use the keyboard shortcut shift+shift to search the project for SystemController. Here's the class! Now, look around for the method systemAdminMenuBlockPage(). And *this* is the actual function that renders the admin "Structure" page:

345 lines | core/modules/system/src/Controller/SystemController.php

```
... lines 1 - 24
class SystemController extends ControllerBase {
... lines 26 - 184
  /**
   * Provides a single block from the administration menu as a page.
   */
  public function systemAdminMenuBlockPage() {
    return $this->systemManager->getBlockContents();
  }
... lines 191 - 343
}
```

In fact, if you add return new Response('HI!') and refresh, it'll completely replace the page! Try this and see if your co-workers can figure out what's going on!

We don't know *yet* what this systemManager thing is or how to debug it, but we're going there next.

I just think it's really cool that we can see exactly what's going on in the page, dive into the core code and find out how things work.

Chapter 7: What is the Service Container?

Ok, the first half of the Symfony stuff was the route, controller, Response flow. Check that off your list.

The second half is all about these magical, wonderful things called unicorns, ahem services! Write it down on your buzzword notepad - you will hear this word a lot, normally by people who are trying to scare you.

Wtf is a Service???

Here's the truth: a service is.... a useful object. Wooooh, ooo, it's not so scary, don't believe the hype! A service is a class that you or someone else creates that does some work for you.

For example, suppose you have a class that logs messages. Hey, that's a service, because you can call a method on it and that will save a log somewhere. Or say you have another class - a mailer that sends emails. OMG, it's a service!!!!

Ok, so what's an example of a class that does *not* help you? Multivariable Calculus -- jk I studied math in college, and Multivariable Calculus helps me every day to calculate the tip at restaurants. Totally worth it.

But really, a class that is *not* a service might be the Node class or some Product class you create that stores data. These classes don't do a lot of work, they mostly exist to store data.

Enough with your theory! If this doesn't make sense, it's cool guys. The takeaway is that when I say "Service", you should yell back "An object that does work for me". Try it: Service! "An object that does work for me". That was actually me.

Wtf is a Service Container?

The next buzz word is "service container". We also call it a "dependency injection container": that's our favorite term to use when we *really* want to terrify someone.

Here's the deal: in Drupal - and Symfony - there is a single object called the container. It's basically an associative array of services. In fact, it holds *every* useful object in the system and each has a nickname. When you want to get access to a service so you can use it to do work for you, you just say

> Yo, Mr Container! Can I get the service who's nickname is logger.factory, please?

Saying please boosts performance by 20%, so mind your manners.

Here's the coolest part: *everything* that Drupal does is actually done by one of the services in the container. The execution of the routing, the handling of the cache, the reading of configuration: these are all done by different services. And guess what? You have access to use these at any time. You can even override and replace *any* core service you want. I'm blowing my own mind.

Chapter 8: Create a Service

In RoarController, we use the $count variable to construct either a little roar or a big roooooar, depending on what's in the URL:

15 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 6
class RoarController
{
    public function roar($count)
    {
        $roar = 'R'.str_repeat('O', $count).'AR!';
        return new Response($roar);
    }
}
```

Let's use our imagination. Pretend that this line is actually quite complicated. Maybe it takes 50 lines of code to figure this out! We could keep all 50 lines in this controller, but that sucks: it makes this function hard to read. Oh, and any code that lives in a controller can't be reused somewhere else. And don't even get me started on unit testing...

It's time to grow up and move out of our parent's basement. I mean, it's time to organize this code and put it somewhere else: in a new class that's independent of Drupal. In other words, in a *service*.

Creating the Service Class

In src/, create a new directory called Jurassic - this directory could be called anything - it's up to you to organize your services. Inside, create a new PHP class called RoarGenerator:

This class, well *any* class, needs to have a namespace that follows the standard of Drupal\ module name \ whatever directory or directories that you're in. This class is in one subdirectory called Jurassic. And make sure the class name matches the filename RoarGenerator:

12 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 2
namespace Drupal\dino_roar\Jurassic;

class RoarGenerator
{
... lines 7 - 10
}
```

This class has *nothing* to do with Drupal, it's completely our's. So you don't need to make it extend any weird Drupal core class. *And* we can make it do whatever we want. Create a public function getRoar() with a $length argument. Head over to RoarController and copy the code that creates that. Replace $count with $length and return that value:

12 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 4
class RoarGenerator
{
    public function getRoar($length)
    {
        return 'R'.str_repeat('O', $length).'AR!';
    }
}
```

Using a Service (the long way)

Great! Now our imaginary, complex code lives somewhere else. How do we use it? It's simple!

Create a $roarGenerator variable and set it new RoarGenerator();:

18 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 4
use Drupal\dino_roar\Jurassic\RoarGenerator;
... lines 6 - 7
class RoarController
{
    public function roar($count)
    {
        $roarGenerator = new RoarGenerator();
... lines 13 - 15
    }
}
```

When I hit tab while typing the class, it auto-completes the line *and* adds the use statement above the class. Do *not* forget your use statements people! You'll get a "Class not found" error, and I'm pretty sure Dries gets a text message whenever it happens. So you know, try not to make him angry: he's really tall.

After instantiating the RoarGenerator object, update the last line to $roar = $roarGenerator->getRoar(); and pass it $count:

18 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 9
    public function roar($count)
    {
        $roarGenerator = new RoarGenerator();
        $roar = $roarGenerator->getRoar($count);

        return new Response($roar);
    }
... lines 17 - 18
```

There is *nothing* special going on here: we moved the logic to an outside class, instantiated that object and then called a method on it. This has nothing to do with Drupal or Symfony or Dries getting text messages: it's just good object oriented code.

Let's give it a shot!

Go back to the browser and navigate to http://localhost:8000/the/dino/says/50. There's the scary roar meaning everything is working.

If you're thinking, "This seems like *not* a big deal - we're just moving code around". You're *so* right. Now keep watching!

Chapter 9: Configuring a New Service

The service container is the magician's hat of Drupal: it's full of useful objects, I mean "services" - we're trying to sound sophisticated. By default, the container is *filled* will cool stuff, like a logger factory, a translator, a white rabbit and a database connection, just to name a few.

List All Services

Head over to the terminal and run a new Drupal Console command:

```
$ drupal container:debug
```

This prints *every single service in the container*: over 400 tools that we have access to out-of-the-box. The nickname of each service is on the left: you'll use that to get that service. On the right, it tells you what type of object this will be.

Most of the services here you won't need to use directly: some like cron, database, file_system and a few other ones might be *really* useful to you.

The module_name.services.yml File

Before we get into how to use these services, I want to add our RoarGenerator to the service container. This means that instead of instantiating it directly, we'll teach Drupal's container how to instantiate the RoarGenerator for us. Then we'll ask the container for the "roar generator" and it will create it for us.

Why would you do this? Hold that thought: you'll see the benefits soon.

To register a service, go to the root of your module and create a dino_roar.services.yml file. Inside, start with a services key:

4 lines | modules/dino_roar/dino_roar.services.yml

```
1  services:
   ... lines 2 - 4
```

In order for the container to create the RoarGenerator for us, it needs to know two things: what's your spirit animal and your favorite color.

Scratch that: the first thing it needs to know is what "nickname" to use for the service. This can be anything, how about dino_roar.roar_generator:

4 lines | modules/dino_roar/dino_roar.services.yml

```
1  services:
2      dino_roar.roar_generator:
   ... lines 3 - 4
```

The only rule is that this needs to be unique in your project and use lowercase and alphanumeric characters with exceptions for _ and ..

The second thing the container needs to know is class for the service. To tell it, add a class key below the nickname and type RoarGenerator:

4 lines | modules/dino_roar/dino_roar.services.yml

```
1  services:
2      dino_roar.roar_generator:
3          class: Drupal\dino_roar\Jurassic\RoarGenerator
```

I'll hit tab because I'm *super* lazy and PhpStorm isn't. It gives me the fully qualified namespace. Thanks Storm!

Go to the terminal and rebuild the cache:

```
$ drupal cache:rebuild
```

Now run container:debug and pipe it into grep for "dino":

```
$ drupal container:debug | grep dino
```

There it is! It's nickname is dino_roar.roar_generator and it'll give us a RoarGenerator object. Yay! Um, but what now? How can we get that service from the container? Actually, I have no idea. I'm kidding - next chapter!

Chapter 10: How to Get a Service in the Controller

There's a dino_roar.roar_generator service in the container and gosh darnit, I want to use this in my controller!

The ControllerBase Class

First, notice that RoarController is *not* extending anything. That's cool: your controller does *not* need to extend anything: Drupal doesn't care. That being said, most of the time a controller will extend a class called ControllerBase. Add it and hit tab so the use statement is added above the class:

36 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 4
use Drupal\Core\Controller\ControllerBase;
... lines 6 - 9
class RoarController extends ControllerBase
{
... lines 12 - 34
}
```

This has a lot of cool shortcut methods - we'll look at some soon. But more importantly, it gives us a new super-power: the ability to get services out of the container.

Override create()

Tip

An alternative to the following method is to register your controller as a service and refer to it in your routing with a your_service_name:methodName syntax (e.g. dino.roar_controller:roar). This allows you to pass other services into your controller without needing to add the create function. For more info, see Structure of Routes.

I'll use the shortcut command+n, select "Override" from the menu and override the create function that lives in the base class:

36 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 6
use Symfony\Component\DependencyInjection\ContainerInterface;
... lines 8 - 9
class RoarController extends ControllerBase
{
... lines 12 - 21
    public static function create(ContainerInterface $container)
    {
... lines 24 - 26
    }
... lines 28 - 34
}
```

You don't need to use PhpStorm to override this, it's just fast and fancy. It also added the use statement for the ContainerInterface. When your controller needs to access services from the container, this is step 1.

Before we had this, Drupal instantiated our controller automatically. But now, it will call *this* function and expect *us* to create a new RoarController and return it. And hey, it passes us the $container!!! There it is, finally! The container is the *most important* object in Drupal... and guess what? It has only *one* important method on it: get(). I bet you can guess what it does.

Create a $roarGenerator variable, set it to $container->get(''); and pass it the name of the service: dino_roar.roar_generator:

36 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 21
22      public static function create(ContainerInterface $container)
23      {
24          $roarGenerator = $container->get('dino_roar.roar_generator');
... lines 25 - 26
27      }
... lines 28 - 36
```

Behind the scenes, Drupal will instantiate that object and give it to us. To create the RoarController, return new static(); and pass it $roarGenerator:

36 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 21
22      public static function create(ContainerInterface $container)
23      {
24          $roarGenerator = $container->get('dino_roar.roar_generator');
25
26          return new static($roarGenerator);
27      }
... lines 28 - 36
```

This may look weird, but stay with me. The new static part says:

> Create a new instance of RoarController and return it, please".

Again, manners are good for performance in D8.

Controller __construct() Method

Next, create a constructor: public function __construct().When we instantiate the controller, we're choosing to pass it $roarGenerator. So add that as an argument:

36 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 9
10  class RoarController extends ControllerBase
11  {
... lines 12 - 16
17      public function __construct(RoarGenerator $roarGenerator)
18      {
... line 19
20      }
... lines 21 - 34
35  }
```

I'll even type-hint it with RoarGenerator to be super cool. Type-hinting is optional, but it makes us best friends.

Finally, create a private $roarGenerator property and set it with $this->roarGenerator = $roarGenerator;:

36 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 9
class RoarController extends ControllerBase
{
... lines 12 - 14
    private $roarGenerator;

    public function __construct(RoarGenerator $roarGenerator)
    {
        $this->roarGenerator = $roarGenerator;
    }
... lines 21 - 34
}
```

Ok, this was a *big* step. As soon as we added the create() function, it was now *our* job to create a new RoarController. And of course, we can pass it whatever it needs to its constructor - like objects or configuration. That's really handy since we have access to the $container and can fetch out any service and pass it to the new controller object.

In the __construct function, we don't use the RoarGenerator yet: we just set it on a property. That saves it for use later. Then, 5, 10, 20 or 100 miliseconds later when Drupal finally calls the roar() function, we know that the roarGenerator property holds a RoarGenerator object.

Delete the new RoarGenerator line and instead use $this->roarGenerator directly:

36 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 28
    public function roar($count)
    {
        $roar = $this->roarGenerator->getRoar($count);

        return new Response($roar);
    }
... lines 35 - 36
```

Woh. Moment of truth: go back to the browser, change the URL and hit enter to reload the page. OMG! It still works!

It's Dependency Injection!

It is *ok* if this was confusing for you. This - by the way - is called dependency injection. Buzzword! Actually, it's kind of a hard application of dependency injection. I'll show you a simpler and more common example in a second. But once you wrap your head around this pattern, you will be unstoppable.

Why did I put my Service in the Container?

Why did we go to all this trouble? After all, this only saved us *one* line in the controller: the new RoarGenerator() line.

Two reasons, *big* reasons. First, I keep telling you the container is like an array of all the useful objects in the system. Ok, that's *kind* of a lie. It's more like an array of *potential* objects. The container doesn't instantiate a service until *and unless* someone asks for it. So, until we actually hit the line that asks for the dino_roar.roar_generator service, your app doesn't use the memory or CPUs needed to create that.

For something big like Drupal, it means you can have a *ton* of services without slowing down your app. If you don't use a service, it's not created.

And if *ten* places in your code ask for the dino_roar.roar_generator service, it gives each of them the same *one* object. That's awesome: you might need a RoarGenerator in 50 places but you don't want to waste memory creating 50 objects. The container takes care of that for us: it only creates *one* object.

The second big benefit of registering a service in the container isn't obvious yet, but I'll show that next. It deals with constructor arguments.

Fetch another Service: a Logger

Now that we have this pattern with the create function and __construct, we're dangerous! We can grab *any* service from the container!

Go to the terminal and run container:debug and grep for log:

```
$ drupal container:debug | grep log
```

Interesting: there's a service called logger.factory that can be used to, um ya know, log stuff. Let's see if we can log the ROOOOOAR message from the controller.

In RoarController add $loggerFactory = $container->get('logger.factory'); and pass that as the second constructor argument when creating RoarController:

45 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 27
28      public static function create(ContainerInterface $container)
29      {
30          $roarGenerator = $container->get('dino_roar.roar_generator');
31          $loggerFactory = $container->get('logger.factory');
32
33          return new static($roarGenerator, $loggerFactory);
34      }
... lines 35 - 45
```

Type-Hinting Core Services

The container:debug command tells us that this is an instance of LoggerChannelFactory. Use that as the type-hint. In the autocomplete, it suggests LoggerChannelFactory and LoggerChannelFactoryInterface. That's pretty common. Often, a class will implement an interface with a similar name. Interfaces are a bit more hipster, and in practice, you can type-hint the original class name or the interface if you want to look super cool in front of co-workers.

Call the argument $loggerFactory. I'll use a PhpStorm shortcut called initialize fields to add that property and set it:

45 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 5
6   use Drupal\Core\Logger\LoggerChannelFactoryInterface;
... lines 7 - 10
11  class RoarController extends ControllerBase
12  {
... lines 13 - 19
20      private $loggerFactory;
21
22      public function __construct(RoarGenerator $roarGenerator, LoggerChannelFactoryInterface $loggerFactory)
23      {
24          $this->roarGenerator = $roarGenerator;
25          $this->loggerFactory = $loggerFactory;
26      }
... lines 27 - 43
44  }
```

If you want to dive into PhpStorm shortcuts, you should: we have a full tutorial on it.

Now in the roar() function, use that property! Add $this->loggerFactory->get(''): this returns one specific *channel* - there's one called default. Finish with ->debug() and pass it $roar:

45 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 35
    public function roar($count)
    {
        $roar = $this->roarGenerator->getRoar($count);
        $this->loggerFactory->get('default')
            ->debug($roar);
... lines 41 - 42
    }
... lines 44 - 45
```

Congrats: we're now using our *first* service from the container.

Refresh to try it! To check the logs, head to a page that has the main menu, click "Reports", then go into "Recent Log Messages." There it is!

Not only did we add a service to the container, but we also used an existing one in the controller. Considering how many services exist, that makes you very dangerous.

Oh, and if this seemed like a lot of work to you, you're in luck! The Drupal Console has *many* code generation commands to help you build routes, controllers, services and more.

Chapter 11: The Magic Behind Shortcuts Methods is: Services

When we extended ControllerBase, we got access to the create() function. And *that* gave us the ability to pass ourselves services. Color us dangerous!

But wait, there's more! ControllerBase gives us a *bunch* of helper functions.

For example, Drupal gives you access to a key value store that can be backed with a database or something like Redis. As soon as you extend ControllerBase you can get a key-value store by typing $this->keyValue() and passing it some collection string:

48 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 35
36      public function roar($count)
37      {
38          $keyValueStore = $this->keyValue('dino');
... lines 39 - 45
46      }
... lines 47 - 48
```

Hey, let's take it for a test drive: $keyValueStore->set('roar_string', $roar);:

48 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 35
36      public function roar($count)
37      {
38          $keyValueStore = $this->keyValue('dino');
39
40          $roar = $this->roarGenerator->getRoar($count);
... lines 41 - 42
43          $keyValueStore->set('roar_string', $roar);
... lines 44 - 45
46      }
... lines 47 - 48
```

Ok cool - let's store something: go to the url, with 50 as the value. Ding! Ok, nothing visually happened, but the roar_string *should* be stored.

Let's see for sure: comment out the $roar equals and key-value store lines. Instead, say $roar = $keyValueStore->get() and pass it roar_string:

49 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 35
36      public function roar($count)
37      {
38          $keyValueStore = $this->keyValue('dino');
... lines 39 - 41
42          $roar = $keyValueStore->get('roar_string');
... lines 43 - 45
46          return new Response($roar);
47      }
... lines 48 - 49
```

Refresh! The key-value store WORKS!

And if I change the URL to 500, the length doesn't change: it's *still* pulling from the store. This has nothing to do with understanding how Drupal works, but isn't that cool?

But, question: what does this keyValue() function *really* do?

In PHPStorm, hold command - or control in Windows - and click this method! Bam! It opens up ControllerBase - *deep* in the heart of Drupal - and shows us the real keyValue() method:

277 lines | core/lib/Drupal/Core/Controller/ControllerBase.php

```
... lines 1 - 37
38  abstract class ControllerBase implements ContainerInjectionInterface {
... lines 39 - 185
186   protected function keyValue($collection) {
187     if (!$this->keyValue) {
188       $this->keyValue = $this->container()->get('keyvalue')->get($collection);
189     }
190     return $this->keyValue;
191   }
... lines 192 - 271
272   private function container() {
273     return \Drupal::getContainer();
274   }
275
276 }
```

And hey, look at this: there is a function in ControllerBase called container() - it's a shortcut to get the service container, the same container that is passed to us in the create() function. It uses it to fetch out a service called keyvalue.

Here's the really important thing: the "key value store" functionality isn't some weird, core part of Drupal: it's just a service called keyvalue like any other service. This means that if you need to use the key value store somewhere outside of the controller, you just need to get access to the keyvalue service. And that is *exactly* what I want to do inside of RoarGenerator.

Chapter 12: Service Arguments

Pretend like the ROAR string calculation takes a really long time - like 2 seconds:

33 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 6
class RoarGenerator
{
... lines 9 - 15
    public function getRoar($length)
    {
... lines 18 - 24
        sleep(2);
... lines 26 - 29
        return $string;
    }
}
```

If that were true, we wouldn't want to generate it more than once: we'd want to cache it in the key value store.

We already know how to access services from our controller: just use the create() function workflow to pass more arguments to __construct(). And hey, sometimes, it's even easier because there are shortcut methods that help do the common stuff.

But how can we get access to a service - like keyvalue from inside of another service, like RoarGenerator? If you're thinking that we could make RoarGenerator extend ControllerBase.... well, you're clever. But nope, sorry! That only works for your controller.

Accessing a Service inside a Service

Instead: here's the rule: as soon as you need access to a service from within a service, we need to create a __construct() method and pass it as an argument. Run container:debug and grep it for keyvalue:

```
$ drupal container:debug | grep keyvalue
```

This tells me that the keyvalue service is an instance of KeyValueFactory. Create the public function __construct() and type-hint its first argument with this class. Woh, but wait! Like before, there is a concrete class *and* a KeyValueFactoryInterface that it implements. You can use either: interfaces are technically more correct and much more hipster, but really, it doesn't matter. Name the argument $keyValueFactory and open the method. I'll use another shortcut, alt enter on a mac, to initialize the field:

33 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 4
use Drupal\Core\KeyValueStore\KeyValueFactoryInterface;
... line 6
class RoarGenerator
{
    private $keyValueFactory;

    public function __construct(KeyValueFactoryInterface $keyValueFactory)
    {
        $this->keyValueFactory = $keyValueFactory;
    }
... lines 15 - 31
}
```

That doesn't do anything special: it just creates this private property and sets it.

Ok, step back for a second. This is *really* similar to what we did in our controller when we needed the dino_roar.roar_generator service. We're saying that *whoever* creates the RoarGenerator will be *forced* to pass in an object that implements KeyValueFactoryInterface. *Who* does that or *how* they do that, well, that's not our problem. But once they do, we store it on a property so we can use it.

And use it we shall! First, create a cache $key called roar_ and then the $length:

33 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 6
7   class RoarGenerator
8   {
... lines 9 - 15
16      public function getRoar($length)
17      {
... line 18
19          $key = 'roar_'.$length;
... lines 20 - 30
31      }
32  }
```

That'll give us a different cache key for each.

Next, grab the key-value store itself with $store = $this->keyValueFactory->get() and then the name of our store: dino. If the store *has* the key, return $store->get($key) and save us from the long, slow 2 second sleep:

33 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 15
16      public function getRoar($length)
17      {
18          $store = $this->keyValueFactory->get('dino');
19          $key = 'roar_'.$length;
20
21          if ($store->has($key)) {
22              return $store->get($key);
23          }
... lines 24 - 29
30          return $string;
31      }
... lines 32 - 33
```

At the bottom, set the string to a variable and then store it with $store->set($key, $string). And don't forget to return $string:

33 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 15
16      public function getRoar($length)
17      {
18          $store = $this->keyValueFactory->get('dino');
19          $key = 'roar_'.$length;
... lines 20 - 24
25          sleep(2);
26
27          $string = 'R'.str_repeat('O', $length).'AR!';
28          $store->set($key, $string);
29
30          return $string;
31      }
... lines 32 - 33
```

That's a perfect cache setup.

Let's give this *cacheable* RoarGenerator a try. Back in the controller, undo everything so we're using that service again:

45 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 10
11  class RoarController extends ControllerBase
12  {
... lines 13 - 35
36      public function roar($count)
37      {
38          $roar = $this->roarGenerator->getRoar($count);
... lines 39 - 41
42          return new Response($roar);
43      }
44  }
```

Ok, refresh!

Configure Service Arguments

Ah, error!

> Call to a member function get*() on null, RoarGenerator line 22

Go check that out. Huh. Somehow, the $keyValueFactory is *not* set: it wasn't passed into the __construct() method.

But wait. Who is instantiating the RoarGenerator anyways? The container is! We registered it as a service, and Drupal says new RoarGenerator() as soon as we ask for it. But it doesn't pass it *any* constructor arguments.

Somehow, we need to *teach* Drupal's container that "Hey, when you instantiate RoarGenerator, it has a constructor argument. I need you to pass in the keyvalue service.". To do that, add an arguments key:

6 lines | modules/dino_roar/dino_roar.services.yml

```
1   services:
2       dino_roar.roar_generator:
3           class: Drupal\dino_roar\Jurassic\RoarGenerator
4           arguments:
... lines 5 - 6
```

This is an array, so I can hit enter and indent four spaces, or two spaces. Two spaces is the Drupal standard. If I put the string keyvalue, it will *literally* pass the string keyvalue as the first argument. That's not what we want! We want the container to pass in the *service* called keyvalue.

The secret way to do that is with the @ symbol:

6 lines | modules/dino_roar/dino_roar.services.yml

```
services:
    dino_roar.roar_generator:
        class: Drupal\dino_roar\Jurassic\RoarGenerator
        arguments:
            - '@keyvalue'
```

Ok, we *just* made a configuration change, so rebuild the Drupal cache:

```
$ drupal cache:rebuild
```

Refresh! Ok, super slow - it's sleeeeping. Shhh... let it sleep. There it is! But next time, it's super quick! Try 50. Slow..... then fast the second time!

Maybe you didn't realize it, but we just had another big Eureka, buzzword-esque moment. Yes! And that is: when you are inside a service - like RoarGenerator - and you need access to another service or configuration value, you need to add a __construct() argument for it and update your service's arguments to pass that in.

So if tomorrow, we need to *log* something from inside RoarGenerator, what are we going to do? PANIC... is *not* the correct answer. No, we're going to calmly add a *second* argument to the __construct() method then update dino_roar.services.yml to configure this new argument.

A cool side-effect of this stuff is that even though we had to change how RoarGenerator is created, we *didn't* need to change any of our code that uses it. In the controller, we just ask for dyno_roar.roar_generator:

45 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 35
    public function roar($count)
    {
        $roar = $this->roarGenerator->getRoar($count);
... lines 39 - 42
    }
... lines 44 - 45
```

The container looks to see if the keyvalue service is already created. If it *isn't*, it creates it first and then passes it to the RoarGenerator. No matter how complex creating RoarGenerator might become, all we need to do is ask the container for it. All the ugly complications are hidden.

Chapter 13: Configuration Parameters

New challenge: what if we need to turn off the key value store stuff while we're developing, but keep it for production? Maybe you're thinking "just comment it out temporarily!". That might work for you, but eventually, I'm going to forget to uncomment it and deploy this to production with caching off. Then, traffic will run-over the site and nobody will get any roars. I think we can do better.

Injecting Configuration

Solution? Make RoarGenerator configurable. That means, add a new argument to __construct: $useCache. Hit option+enter to create the property and set it:

37 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 6
class RoarGenerator
{
... line 9
    private $useCache;

    public function __construct(KeyValueFactoryInterface $keyValueFactory, $useCache)
    {
... line 14
        $this->useCache = $useCache;
    }
... lines 17 - 35
}
```

This will be a boolean that controls whether or not we should cache.

Down below, update the if statement: if $this->useCache *and* the store has it, then return it. Below, add another if statement that says we should *only* store this in the cache if $this->useCache is true:

37 lines | modules/dino_roar/src/Jurassic/RoarGenerator.php

```
... lines 1 - 17
    public function getRoar($length)
    {
... lines 20 - 22
        if ($this->useCache && $store->has($key)) {
            return $store->get($key);
        }
... lines 26 - 28
        $string = 'R'.str_repeat('O', $length).'AR!';
        if ($this->useCache) {
            $store->set($key, $string);
        }
... lines 33 - 34
    }
... lines 36 - 37
```

The RoarGenerator is now perfect: whoever creates it can control this behavior. Because we added a second constructor argument, we need to update the service configuration. Add another line with a - and set the second argument to true... for now:

7 lines | modules/dino_roar/dino_roar.services.yml

```
services:
  dino_roar.roar_generator:
    class: Drupal\dino_roar\Jurassic\RoarGenerator
    arguments:
      - '@keyvalue'
      - true
```

Time to test: rebuild the cache:

```
$ drupal cache:rebuild
```

Refresh! The cache is activated... and everything is still really, really fast. If you try 51, that's not cached yet, but it's fast on the second load.

Introducing: Parameters

But this didn't solve our problem, it just moved the code we need to change from RoarGenerator into the service file.

In addition to the services key - that can hold many services - these files are allowed to have one other root key: parameters. The parameter system is a key-value configuration system. This means... I lied to you! The service container isn't *just* an array-like object that holds services. It also holds a key-value configuration system called parameters.

Add a new parameter called dyno.roar.use_key_value_cache and set it to true:

10 lines | modules/dino_roar/dino_roar.services.yml

```
parameters:
  dino.roar.use_key_value_cache: true
... lines 3 - 10
```

To use this, I gotta tell you about the one *other* magic syntax in these files. That is, use % - the name of the parameter - then %:

10 lines | modules/dino_roar/dino_roar.services.yml

```
... lines 1 - 3
services:
  dino_roar.roar_generator:
... line 6
    arguments:
... line 8
      - '%dino.roar.use_key_value_cache%'
```

When you surround something with percent signs, the container finds a parameter by this name and passes that.

And there's a bonus: these parameters can be accessed in any of these service files. That means that if we define parameter A in one module, you can use it - or even change it - somewhere else. This ends up being *critical* to how you can control the *core* of Drupal. And yes, we'll talk about that soon!

But first, rebuild the cache real quick for a sanity check:

```
$ drupal cache:rebuild
```

Refresh! Everyone is still happy! We're awesome now with parameters... but we *still* haven't *quite* solved our problem.

Now that the value we want to change is stored as a parameter, we can override it *only* on the local machine. Let me show you how.

development.services.yml to Change Local Behavior

Earlier, we created a settings.local.php file, which is itself loaded from settings.php. This file loads a development.services.yml file:

12 lines | sites/development.services.yml

```
# Local development services.
#
# To activate this feature, follow the instructions at the top of the
# 'example.settings.local.php' file, which sits next to this file.
services:
  cache.backend.null:
    class: Drupal\Core\Cache\NullBackendFactory
... lines 8 - 12
```

And that's the key: this file gives us tremendous power: power to override services and parameters. These changes will *only* affect our local environment... well, really, *any* environment where you've chosen to create the settings.local.php file.

Hey, you know what we should do? Override the parameter! Copy the name and set it to false With any luck, this will turn off the cache:

12 lines | sites/development.services.yml

```
... lines 1 - 8
parameters:
  dino.roar.use_key_value_cache: false
... lines 11 - 12
```

Rebuild it:

```
$ drupal cache:rebuild
```

Refresh! OMG, it's *so* slow! Every time I refresh, it's slow. This is *awesome*. I mean, not the slow part - but the fact that we can tweak behavior locally.

Parameters are the *number one* way that you control the behavior of core and third party modules. In fact, check out the core directory - that's where Drupal lives! And hey, look at that core.services.yml file!

1576 lines | core/core.services.yml

```
... lines 1 - 35
36  services:
... lines 37 - 348
349   keyvalue:
350     class: Drupal\Core\KeyValueStore\KeyValueFactory
351     arguments: ['@service_container', '%factory.keyvalue%']
... lines 352 - 360
361   logger.factory:
362     class: Drupal\Core\Logger\LoggerChannelFactory
363     parent: container.trait
364     tags:
365       - { name: service_collector, tag: logger, call: addLogger }
... lines 366 - 1576
```

All the really important, base services are defined here: they're defined *just* like we define *our* services. That's really cool. Most of the services we see in container:debug are coming from here.

Site-Specific Services

At the top, it *also* has a parameters key with all kinds of parameters stored under it:

1576 lines | core/core.services.yml

```
1   parameters:
2     session.storage.options:
3       gc_probability: 1
4       gc_divisor: 100
5       gc_maxlifetime: 200000
6       cookie_lifetime: 2000000
7     twig.config:
8       debug: false
9       auto_reload: null
10      cache: true
... lines 11 - 16
17    factory.keyvalue:
18      default: keyvalue.database
... lines 19 - 1576
```

These are values that *you* can override to control core behavior for your app. How? In sites/default, you already have a default.services.yml. If you rename this to services.yml, Drupal will load it. That's thanks to a line in settings.php... that's hiding from me... there it is!

727 lines | sites/default/settings.php

```
... lines 1 - 650
651 /**
652  * Load services definition file.
653  */
654 $settings['container_yamls'][] = __DIR__ . '/services.yml';
... lines 655 - 727
```

The config container_yamls.

Overriding Core Parameters

Open up default.services.yml. Hey, parameters!

156 lines | sites/default/default.services.yml

```
parameters:
  session.storage.options:
... lines 3 - 9
    gc_probability: 1
... line 11
    gc_divisor: 100
... lines 13 - 38
  twig.config:
    # Twig debugging:
    #
    # When debugging is enabled:
    # - The markup of each Twig template is surrounded by HTML comments that
    #   contain theming information, such as template file name suggestions.
    # - Note that this debugging markup will cause automated tests that directly
    #   check rendered HTML to fail. When running automated tests, 'debug'
    #   should be set to FALSE.
    # - The dump() function can be used in Twig templates to output information
    #   about template variables.
    # - Twig templates are automatically recompiled whenever the source code
    #   changes (see auto_reload below).
    #
    # For more information about debugging Twig templates, see
    # https://www.drupal.org/node/1906392.
    #
    # Not recommended in production environments
    # @default false
    debug: false
... lines 59 - 156
```

In fact, these are the same parameters we saw in core.services.yml. So everything is setup to allow you to easily control many different parts of the system.

One of the settings under twig.config is called debug, and it's set to false. I want to set this to true, to make theming easier. I could rename this file to services.yml and change that value. But then when we deploy, debug would *still* be true. No, this is a change that I only want to make *locally*. That's the job of development.services.yml.

Add twig.config there with debug set to true:

14 lines | sites/development.services.yml

```
... lines 1 - 8
parameters:
... lines 10 - 11
  twig.config:
    debug: true
```

Cool, that should replace the original value from core.services.yml. Let's clear some cache:

```
$ drupal cache:rebuild
```

Change the URL to the home page where Drupal is using Twig. It looks the same... until you view the source. There it is: the source is now *filled* with HTML comments that tell you exactly which templates each bit of markup comes from and how you can override them. And with debug set to true, you won't have to rebuild your cache after every template change: Drupal will do that automatically.

This is great! By understanding a few concepts, we're overriding core features in Drupal and actually understanding how it works. That's awesome!

Chapter 15: Drupal Events versus Hooks

When I say Drupal, most people think: hooks. And then some people roll their eyes. Yep, the infamous, but, honestly, super-powerful hooks system still exists. But not everything uses hooks anymore: some use a brand new event system. With events, you create a function and then tell Drupal to call your function when something happens. Hmm.... that sounds a lot like hooks.

Yep! Events are the object-oriented version of hooks. And like hooks, if you can learn how to harness events, you're going to be very, very dangerous.

First, get into the profiler by clicking any link on the web debug toolbar. Well look at that: an Events tab. I think we should click it.

Events versus Hooks

This is awesome: it tells you all of the events and the listeners that have been called during this request. Wait, back up, new terminology. When something happens in Drupal's core that we might want to hook into, Drupal dispatches an event. What that really means is that Drupal calls all the functions that want to be executed when this event happens.

And this works *just* like hooks. For example, when Drupal 7 builds the menu, it knows you might want to hook into this process. So, it executes hook_menu(). What this *really* means is that it calls all the functions that implement this hook.

So you can think of hook_menu as an "event", and all the callbacks that implement it as the "listeners". In reality, the *only* difference between the hook system and the event system is *how* you tell Drupal that you have a listener. With hooks, create a function with *just* the right name - like dino_roar_menu() - and boom! Drupal automagically calls it. With events, create a function with *any* name and tell Drupal about it with configuration.

Each event has a name, and apparently there's one called kernel.request. This is the *first* event that happens when the request is being processed. If you need some code to run early on every page, this is your guy. On the right, there is a class called ProfilerListener with an onKernelRequest() function. That's the listener function that was called. In typical Drupal 8 fashion, it's not a flat-function anymore: it's a method inside an object.

There are a bunch of listeners on kernel.request and several other events, like kernel.controller and render.page_display_variant.select. Don't worry about *why* you would want to listen to an event. Like with "hooks", you'll eventually Google "How do I do X", and the answer will be "add a listener to some event".

At the bottom, there's another section: Non called listeners, with *more* stuff you can hook into? Wait, why aren't they called? Like hooks, not all events happen on every request. Like this one - routing.route_finished: it's only called when the routing cache is built. And if you wanted to hook into the route-building process, you could attach a listener to this.

Chapter 16: Event Subscribers and Dependency Injection Tags

Here's the challenge: add a function that will be called on *every* request, right at the beginning, before any controller is called. This means we should hook into the kernel.request event.

Create an Event Subscriber

In the Jurassic directory - well, it doesn't matter where - add a new DinoListener class. Next, make this class implement EventSubscriberInterface:

22 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
<?php

namespace Drupal\dino_roar\Jurassic;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
... lines 6 - 7
class DinoListener implements EventSubscriberInterface
{
... lines 10 - 21
}
```

If you're not *super* cool with interfaces yet, hit pause and go practice with them in the Object Oriented Level 3 course.

This interface forces us to have 1 method: getSubcribedEvents(). In PhpStorm, hit command+n to bring up the Generate menu, select "Implement Methods", and highlight this. That's a nice shortcut to add the *one* method we need:

22 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 7
class DinoListener implements EventSubscriberInterface
{
... lines 10 - 14
    public static function getSubscribedEvents()
    {
... lines 17 - 19
    }

}
```

We need to tell Drupal *which* event we want to listen to and *what* method to call when that event happens. This method returns an array that says exactly that. Use KernelEvents::REQUEST as the key and hit tab to auto-complete and get the use statement. But hold on, that is *just* a constant that means kernel.request. You can totally use the string kernel.request if you want to. The value is the method to call. Use, onKernelRequest():

22 lines | modules/dino_roar/src/Jurassic/DinoListener.php

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

}
```

Up top, create that method: public function onKernelRequest(). Every event listener is passed exactly *one* argument: an event object:

22 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 7
class DinoListener implements EventSubscriberInterface
{
    public function onKernelRequest($event)
    {
... line 12
    }
... lines 14 - 21
}
```

The cool thing is that this object will hold any information your function will need. The tricky thing is that this is a different object depending on *which* event you're listening to.

No worries, let's just var_dump() it and see what it is!

22 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 9
    public function onKernelRequest($event)
    {
        var_dump($event);die;
    }
... lines 14 - 22
```

Configuring an Event Subscriber

Ok, this class is setup. The last step is to tell Drupal we have an event subscriber. How do you do that? Register this class as a service. Get used to that answer.

In dino_roar.services.yml, add a new service - the name doesn't matter. Set the class to the full namespace. In this case, there are *no* constructor arguments. Add an arguments key, but leave it blank with square brackets:

16 lines | modules/dino_roar/dino_roar.services.yml

```
... lines 1 - 3
services:
... lines 5 - 10
    dino_roar.dino_listener:
        class: Drupal\dino_roar\Jurassic\DinoListener
        arguments: []
... lines 14 - 16
```

Congratulations! You've created a normal, boring service... but Drupal still doesn't know it's an event subscriber. Somehow, we need to tell Drupal:

> Yo, this is *not* a normal service, this is an event subscriber! I want you to call the getSubscribedEvents() method so you know about my kernel.request listener!

Whenever you want to raise your hand and scream "Drupal, this service is special, use it for this core purpose", you're going to use a tag. The syntax for a tag is ugly, but here it goes. Add tags, add a new line, indent, add a dash, then a set of curly braces. Every tag has a name: this one is event_subscriber:

16 lines | modules/dino_roar/dino_roar.services.yml

```
... lines 1 - 3
services:
... lines 5 - 10
    dino_roar.dino_listener:
        class: Drupal\dino_roar\Jurassic\DinoListener
        arguments: []
        tags:
            - { name: event_subscriber }
```

By doing this, you've now told Drupal's core that our DynoListener service is an event subscriber. It knows to go in and find out what events we're subscribed to.

What are these Dependency Injection Tags?

***seealso Tags work via a system called "Compiler Passes". These are shared with Symfony, and we talk a lot more about them here: Compiler Pass and Tags.

There are other tags that for other things. For example, if you want to add some custom functions to Twig, you'll create a class that extends Twig_Extension, register it as a service, and tag it with twig.extension. This tells Twig, "Yo, I have a Twig Extension here - use it!".

If you're using tags, then you're probably doing something geeky-cool, hooking into some core part of the system. And you don't need to know about all the tags. You'll Google "how do I register an event subscriber" and see that it uses a tag called event_subscriber. You just need to understand how they work: that it's your way of making your service special.

Ok, back to the action. Rebuild the cache:

```
drupal cache:rebuild
```

Since the kernel.request event happens on every request, we should be able to refresh and see the var_dump(). Great Scott! There it is! Now, let's do something in this!

Chapter 17: Event Arguments and the Request

We *now* know that when you listen to *this* event, it passes you an event object called GetResponseEvent. Type-hint the argument to enjoy auto-completion:

28 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 5
use Symfony\Component\HttpKernel\Event\GetResponseEvent;
... lines 7 - 8
class DinoListener implements EventSubscriberInterface
{
    public function onKernelRequest(GetResponseEvent $event)
    {
... lines 13 - 18
    }
... lines 20 - 27
}
```

Using the Event Argument

This event object has a getRequest() method, use that to set a new $requestvariable:

28 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 10
    public function onKernelRequest(GetResponseEvent $event)
    {
        $request = $event->getRequest();
... lines 14 - 18
    }
... lines 20 - 28
```

This is Drupal's - and Symfony's - Request object. If you want to read some GET params, POST params, headers or session stuff, this is your friend. You can of course get the Request object inside a controller.

Here's the goal: if the URL has a ?roar=1 on it, then I want to log a message. If not, we'll do nothing. Make a new variable called $shouldRoar. To access the GET, or *query* parameters on the request, use $request->query->get('roar'):

28 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 10
    public function onKernelRequest(GetResponseEvent $event)
    {
        $request = $event->getRequest();
        $shouldRoar = $request->query->get('roar');
... lines 15 - 18
    }
... lines 20 - 28
```

If it's not there, this returns null:

Next, if ($shouldRoar), just var_dump('ROOOOAR') and die to test things:

28 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 10
    public function onKernelRequest(GetResponseEvent $event)
    {
... lines 13 - 15
        if ($shouldRoar) {
            var_dump('ROOOOOOOOAR');die;
        }
    }
... lines 20 - 28
```

Since we didn't touch any configuration, we can refresh without clearing anything.

Page not found! We're on the profiler page for a past request, and this information is stored in the cache... which we just cleared. So go find a real page. Ok, it works perfectly. Now add ?roar=1. It hits! And this will work on *any* page.

Dependency Inject All the Things

How can we log something? We faced this problem earlier when we wanted to use the keyvalue store inside RoarGenerator. We solved it with dependency injection: create a __construct() method, pass in what you need, and set it on a property. This is no different.

Add public function __construct(). Look for the logger in container:debug:

```
$ drupal container:debug | grep log
```

The logger.factory is an instance of LoggerChannelFactory. Type-hint using that. And like with other stuff, this has an interface, which is a trendier option. I'll hit option+enter to add the property and set it:

37 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 4
use Drupal\Core\Logger\LoggerChannelFactoryInterface;
... lines 6 - 9
class DinoListener implements EventSubscriberInterface
{
    private $loggerChannel;

    public function __construct(LoggerChannelFactoryInterface $loggerChannel)
    {
        $this->loggerChannel = $loggerChannel;
    }
... lines 18 - 36
}
```

Below, if you wanna roar, do it! $this->loggerFactory->get() - and use the default channel. Then add a debug message: 'Roar requested: ROOOOOAR':

37 lines | modules/dino_roar/src/Jurassic/DinoListener.php

```
... lines 1 - 9
class DinoListener implements EventSubscriberInterface
{
... lines 12 - 18
    public function onKernelRequest(GetResponseEvent $event)
    {
... lines 21 - 23
        if ($shouldRoar) {
            $channel = $this->loggerChannel->get('default')
                ->debug('ROOOOOOOOAR');
        }
    }
... lines 29 - 36
}
```

That's it guys! We didn't touch any config, so just refresh. Oh no, an error:

> Argument 1 passed to DinoListener::__construct() must implement LoggerChannelFactoryInterface, none given.

Can you spot the problem? We saw this once before. We forgot to *tell* the container about the new argument. In dino_roar.services.yml, add @logger.factory:

16 lines | modules/dino_roar/dino_roar.services.yml

```
... lines 1 - 3
services:
... lines 5 - 10
    dino_roar.dino_listener:
... line 12
        arguments: ['@logger.factory']
... lines 14 - 16
```

This is a single-line YAML-equivalent to what I did for the other service.

Rebuild the cache:

```
$ drupal cache:rebuild
```

Refresh. No error! Head into "Reports" and then "Recent Log Messages". There's the "Roar Requested".

You're now an event listener pro. We listened to the kernel.reqeust event, but you can add a listener to *any* event that Drupal *or* a third-party module exposes.

Chapter 18: The Render Array... and Event Listeners

While we're on the topic, the render array works via a listener. When I told you earlier that a controller *always* returns a Response object, that was a bloody lie! Return an array instead, and set #title to $roar:

47 lines | modules/dino_roar/src/Controller/RoarController.php

```
... lines 1 - 10
class RoarController extends ControllerBase
{
... lines 13 - 35
    public function roar($count)
    {
... lines 38 - 41
        return [
            '#title' => $roar
        ];
    }
}
```

Head to /the/dino/says/50 in the browser. As expected, here is a fully-themed page. And I say "as expected", but is it expected? One of the cardinal rules of a Symfony controller is that it must return a Symfony Response object. But this is most certainly *not* a Response: it's an array.

In truth, there's an exception to that rule: if you don't return an array, Drupal, well, Symfony, dispatches an event called kernel.view. There it is! It runs *after* the controller and has one job: try to convert the controller return value into a Response.

Check out this MainContentViewSubscriber. Surprise! This is the listener responsible for handling the render array, via its onViewRenderArray() method. I *love* how the magic of Drupal gets more transparency via events.

So if you want to see how the render array system works, you just need to open that class. In fact, I'll give you a little preview. Find core.services.yml again. Inside, there's a section that includes some services, main_content_renderer_html and main_content_renderer_ajax:

1576 lines | core/core.services.yml

```
... lines 1 - 983
  main_content_renderer.html:
    class: Drupal\Core\Render\MainContent\HtmlRenderer
    arguments: ['@title_resolver', '@plugin.manager.display_variant', '@event_dispatcher', '@module_handler', '@renderer', '@render
    tags:
      - { name: render.main_content_renderer, format: html }
  main_content_renderer.ajax:
    class: Drupal\Core\Render\MainContent\AjaxRenderer
    arguments: ['@element_info']
    tags:
      - { name: render.main_content_renderer, format: drupal_ajax }
      - { name: render.main_content_renderer, format: iframeupload }
... lines 995 - 1576
```

If you looked into MainContentViewSubscriber far enough, you'd see it uses these classes behind the scenes to figure out *how* to render the page. What's really interesting - at least for me, admittedly, I *love* this stuff - is this tag: main_content_renderer with format: html. The AJAX service also has one, with format: drupal_ajax.

I'm getting really, really advanced on you guys, but this is the power of the new way things are done in Drupal. If you wanted to have your own "main_content_renderer" for some other format, like JSON, all you need to do is create a service, and give

it this tag with the new format. You may never need to do this, but as you dig, Drupal's layers start to open up.

That's if for now. If there's something that's confusing you or mysterious in D8, let me know and we'll dive in together. Ok, see ya guys next time!