Creating a Reusable (& Amazing) Symfony Bundle

With <3 from SymfonyCasts

Chapter 1: Bootstrapping the Bundle & Autoloading

Heeeeey Symfony peeps! I'm excited! Because we're going to dive *deep* in to a *super* interesting topic: how to create your *own* bundles. This is useful if you need to share code between your own projects. Or if you want to share your great new open source library with the *whole* world. Actually, forget that! This tutorial is going to be *awesome* even if you *don't* need to do either of those. Why? Because we *use* third-party bundles *every* day. And by learning how to create one, we're going to become experts in how they work and *really* get a look at Symfony under the hood.

As always, you can earn free high-fives by downloading the source code from this page and coding along with me. After unzipping the file, you'll find a start/ directory with the same code that you see here. Follow the README.md file for steps on how to get your project setup.

The last step will be to open a terminal, move into the project, sip your coffee, and run:

```
$ php bin/console server:run
```

to start the built-in PHP web server.

Introducing KnpUIpsum

Head to your browser and go to http://localhost:8000. Say hello to The Space Bar! This is a Symfony *application* - the one we're building in our beginner Symfony series. Click into one of the articles to see a *bunch* of *delightful*, fake text that we're using to make this page look real. Each time you refresh, you get new random, happy content.

To find out where this is coming from, in the project, open src/Service/KnpUIpsum.php. Yes! This is our *new* creation: it returns "lorem ipsum" dummy text, but with a *little* KnpUniversity flare: the classic latin is replaced with rainbows, unicorns, sunshine and more of our favorite things.

340 lines | src/Service/KnpUIpsum.php

```
... lines 1 - 2
namespace App\Service;
... lines 4 - 9
class KnpUIpsum
{
    private $unicornsAreReal;

    private $minSunshine;

    public function __construct(bool $unicornsAreReal = true, $minSunshine = 3)
    {
        $this->unicornsAreReal = $unicornsAreReal;
        $this->minSunshine = $minSunshine;
    }

    /**
     * Returns several paragraphs of random ipsum text.
     *
     * @param int $count
     * @return string
     */
    public function getParagraphs(int $count = 3): string
    {
        $paragraphs = array();
        for ($i = 0; $i < $count; $i++) {
            $paragraphs[] = $this->addJoy($this->getSentences($this->gauss(5.8, 1.93)));
        }

        return implode("\n\n", $paragraphs);
    }
... lines 37 - 338
}
```

And, you know what? I think we *all* deserve more cupcakes, kittens & baguettes in our life. So I want to share this functionality with the world, by creating the KnpULoremIpsumBundle! Yep, we're going to extract this class into its own bundle, handle configuration, add tests, and do a bunch of other cool stuff.

Right now, we're using this code inside of ArticleController: it's being passed to the constructor. Below, we use that to generate the content.

```
77 lines | src/Controller/ArticleController.php
... lines 1 - 2
namespace App\Controller;
... lines 4 - 14
class ArticleController extends AbstractController
{
    /**
     * Currently unused: just showing a controller with a constructor!
     */
    private $isDebug;

    private $knpUIpsum;

    public function __construct(bool $isDebug, KnpUIpsum $knpUIpsum)
    {
        $this->isDebug = $isDebug;
        $this->knpUIpsum = $knpUIpsum;
    }

    /**
     * @Route("/", name="app_homepage")
     */
    public function homepage()
    {
        return $this->render('article/homepage.html.twig');
    }
... lines 37 - 75
}
```

Isolating into a new Bundle Directory

Ok, the *first* step to creating a new bundle is to move this code into its own location. *Eventually*, all the code for the bundle will live in its own *completely* separate directory & repository. But, sometimes, when you *first* start building, it's a bit easier to *keep* the code in your project: it let's you hack on things really quickly & test them in your app.

So let's keep the code here for now, but isolate it from the app's code. To do that, create a new lib/ directory. And then, another called LoremIpsumBundle: this will be the temporary home for our shiny bundle. Inside, there are a few valid ways to organize things, but I like to create a src/ directory.

```
$ mkdir lib
$ mkdir lib/LoremIpsumBundle
$ mkdir lib/LoremIpsumBundle/src
```

Perfect! Now, move the KnpUIpsum class into that directory. And yea, you could put this into a src/Service directory, or anywhere else you want.

New Vendor Namespace

Oh, but this namespace will *not* work anymore. We need a namespace that's *custom* to our bundle. It could be anything, but usually it has a vendor part - like KnpU and then the name of the library or bundle - LoremIpsumBundle.

```
340 lines | lib/LoremIpsumBundle/src/KnpUIpsum.php
... lines 1 - 2
namespace KnpU\LoremIpsumBundle;
... lines 4 - 9
class KnpUIpsum
... lines 11 - 340
```

And, that's it! If we had decided to put KnpUIpsum into a sub-directory, like Service, then we would of course also add Service to the end of the namespace like normal.

Next, back in ArticleController, go up to the top, remove the use statement, and re-type it to get the new one.

```
77 lines | src/Controller/ArticleController.php
... lines 1 - 2
namespace App\Controller;
... line 4
use KnpU\LoremIpsumBundle\KnpUIpsum;
... lines 6 - 77
```

Handling Autoloading

So... will it work! Yea... probably not - but let's try it! Nope! But I *do* love error messages:

> Cannot autowire ArticleController argument $knpUIpsum... because the KnpUIpsum class was not found.

Of course! After creating the new lib/ directory, we need to tell Composer's autoloader to look for the new classes there. Open composer.json, find the autoload section, and add a new entry: the KnpU\\LoremIpsumBundle\\ namespace will live in lib/LoremIpsumBundle/src/.

```
77 lines | composer.json
... lines 1 - 36
    "autoload": {
        "psr-4": {
            "KnpU\\LoremIpsumBundle\\": "lib/LoremIpsumBundle/src/",
... line 40
        }
    },
... lines 43 - 77
```

Then, open a new terminal tab. To make the autoload changes take effect, run:

```
$ composer dump-autoload
```

Registering the Service

Will it work *now*? Try it! Bah, not yet: but we're closer. The error changed: instead of "class not found", now it says that no KnpUIpsum service exists. To solve this, open config/services.yaml.

Thanks to the auto-registration code in here, we don't normally need to register our classes as services: that's automatic. But, it's only automatic for classes that live in src/. Yep, as *soon* as we moved the class from src/ to lib/, that service disappeared.

And that's ok! When you create a re-usable bundle, you actually *don't* want to rely on auto-registration or autowiring. Instead, as a best-practice, you should configure everything *explicitly* to avoid any surprises.

To do that, at the bottom of this file, add KnpU\LoremIpsumBundle\KnpUIpsum: ~.

```
39 lines | config/services.yaml
... lines 1 - 5
services:
... lines 7 - 37
    KnpU\LoremIpsumBundle\KnpUIpsum: ~
```

This adds a new service for that class. And because we don't need to pass any options or arguments, we can just set this to ~. The class *does* have constructor arguments, but they have default values.

Ok, try it again! Yes! It *finally* works! We've successfully isolated our code into its own directory and we are ready to hack! Next, let's make this a bundle with a bundle class and start digging into how bundles can automatically register services.

Chapter 2: Auto-Adding Services

At this point... we have a *directory* with a PHP class inside. And, honestly, we *could* just move this into its own repository, put it on Packagist and be done! But in that case, it wouldn't be a *bundle*, it would simply be a *library*, which is more or less defined as: a directory full of PHP classes.

So what *is* the difference between a library and a bundle? What does a bundle give is that a library does not? The "mostly-accurate" answer is simple: *services*. If we *only* created a library, people could use our classes, but it would be up to *them* to add configuration to register them as *services* in Symfony's container. But if we make a bundle, *we* can *automatically* add services to the container as *soon* as our bundle is installed. Sure, bundles can also do a few other things - like provide translations and other config - but providing services is their main super power.

So, we're going to create a bundle. Actually, the *perfect* solution would be to create a *library* with only the KnpUIpsum class, and then *also* a bundle that *requires* that library and adds the Symfony service configuration. A good example of this is KnpMenu and KnpMenuBundle.

Creating the Bundle Class

To make this a bundle, create a new class called KnpULoremIpsumBundle. This could be called anything... but usually it's the vendor namespace plus the directory name.

Make this extend Bundle and... that's it! You almost *never* need to have any logic in here.

11 lines | lib/LoremIpsumBundle/src/KnpULoremIpsumBundle.php

```
// ... lines 1 - 2
namespace KnpU\LoremIpsumBundle;
// ... lines 4 - 6
class KnpULoremIpsumBundle extends Bundle
// ... lines 8 - 11
```

To enable this in our app, open bundles.php and configure it for all environments. I'll remove the use statement for consistency. Normally, this happens automatically when we install a bundle... but since we just added the bundle manually, we gotta do it by hand.

17 lines | config/bundles.php

```
// ... lines 1 - 2
return [
// ... lines 4 - 14
    KnpU\LoremIpsumBundle\KnpULoremIpsumBundle::class => ['all' => true],
];
```

And, congratulations! We now have a bundle!

Creating the Extension Class

So.... what the heck does that give us? Remember: the super-power of a bundle is that it can *automatically* add services to the container, without the user needing to configure *anything*. How does that work? Let me show you.

Next to the bundle class, create a new directory called DependencyInjection. Then, add a new class inside with the same name of the bundle, except ending in Extension. So, KnpULoremIpsumExtension. Make this extend Extension from HttpKernel. This forces us to implement *one* method. I'll go to the Code -> Generate menu, or Cmd+N on a Mac, choose "Implement Methods" and select the one we need. Inside, just var_dump that we're alive and... die!

15 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 2
namespace KnpU\LoremIpsumBundle\DependencyInjection;
... lines 4 - 7
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        var_dump('We\'re alive!');die;
    }
}
```

Now move over and refresh. Yes! It hits our new code!

This is *really* important. Whenever Symfony builds the container, it loops over all the bundles and, inside of each, looks for a DependencyInjection directory and then inside of that, a class with the same name of the bundle, but ending in Extension. Woh. *If* that class exists, it instantiates it and calls load(). This is *our* big chance to *add* any services we want! We can go *crazy*!

See this $container variable? It's not *really* a container, it's a container *builder*: something we can add services to.

Adding services.xml

Right now, our service is defined in the config/services.yaml file of the application. Delete that! We're going to put a service configuration file *inside* the bundle instead. Create a Resources/ directory and another config/ directory inside: this is the best-practice location for service config. Then, add services.xml. Yep, I said *XML*. Wait, don't run away!

You *can* use YAML to configure your services, but XML is the best-practice for re-usable bundles... though it doesn't matter much. Using XML *does* have one tiny advantage: it doesn't require the symfony/yaml component, which, at least in theory, makes your bundle feel a bit lighter.

To fill this in... um, I cheat. Google for "Symfony Services", open the documentation, search for XML, and stop when you find a code block that *defines* a service. Click the XML tab and steal this! Paste it into our code. The only thing *we* need to do is configure a single service whose id is the class of the service. So, use KnpU\LoremIpsumBundle\KnpUIpsum. We're not passing any arguments, so we can use the short XML syntax for now.

11 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services
        http://symfony.com/schema/dic/services/services-1.0.xsd">

    <services>
        <service id="KnpU\LoremIpsumBundle\KnpUIpsum" />
    </services>
</container>
```

But this file isn't processed automatically. Go to the extension class and remove the var_dump(). The code to load the config file looks a little funny: $loader = new XmlFileLoader() from the DependencyInjection component. Pass this a new FileLocator - the one from the Config component - with the path to that directory: ../Resources/config. Below that, add $loader->load('services.xml').

18 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 9
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
        $loader->load('services.xml');
    }
}
```

Voilà! Refresh the page. It works! When the container builds, the load() method is called and our bundle adds its service.

Next, let's talk about service id best-practices, how to support autowiring and public versus private services.

Chapter 3: Autowiring & Public/Private Services

Head back to services.xml: there are a *few* really important details we need to get straight.

Best-Practice Service IDs

First, in our *applications*, we usually make the service id match the class name for simplicity: and that's what we've done here. But, when you create a re-usable bundle, the best practice is to use *snake-case* service id's. Change the key to class and add id="knpu_lorem_ipsum.knpu_ipsum".

11 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 6
    <services>
        <service id="knpu_lorem_ipsum.knpu_ipsum" class="KnpU\LoremIpsumBundle\KnpUIpsum" />
    </services>
... lines 10 - 11
```

Why is this the best practice? Well, the user *could* in theory change the class of this service to one of their *own* classes. And, it would be pretty weird to have a service called KnpU\LoremIpsumBundle\KnpUIpsum... when that's *not* actually the class of the service.

Supporting Autowiring

Anyways, this *simple* change, totally borks our app! Woohoo! Refresh!

It *once* again says that no service exists for KnpUIpsum. Remember: we're *autowiring* that class into our controller. And in order for autowiring to work, there *must* be a service whose id *matches* the class used in the type-hint. By changing the id from the class to that weird, snake-case string, we just broke autowiring!

No worries: we can solve this with a service *alias*. First, identify each service in your app that you *intend* to be used directly by the user. Yea, I know, we only have *one* service. But often, a bundle will have *several* services, but only *some* of them are meant to be accessed by the user: the others are just meant to support things internally.

For each "important" service, define an alias: <service id="" ...> and paste in the class name. Then, alias="" and type the first service's id: knpu_lorem_ipsum.knpu_ipsum.

13 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 6
    <services>
... lines 8 - 9
        <service id="KnpU\LoremIpsumBundle\KnpUIpsum" alias="knpu_lorem_ipsum.knpu_ipsum" />
    </services>
... lines 12 - 13
```

To see what this did, move over to your terminal and run:

```
$ php bin/console debug:container --show-private knpu
```

Ok, there are *two* services: one has the snake-case id and the other is the full class name. If you choose the second, *it's* just an *alias* to the snake-case service. But now that there *is* a service whose id is the class name, anyone can once again autowire using that type-hint. This fixes our page.

Yep, in ArticleController, the KnpUIpsum class is once-again autowired.

Public versus Private Services

Ok, there is *one* last thing you need to think about when setting up your services: whether or not each service should be *public* or *private*. In Symfony 4.0, services are *private* by default, which means that a user cannot fetch a service directly from the container with $container->get() and then the service's id. Instead, you need to use dependency injection, which includes autowiring.

And this is *really* the way people should code going forward: we really should *not* need services to be public. But, since some people *still* do fetch services directly, you *may* want to make your important services public. Let's do this: public="true".

```
13 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml
    ... lines 1 - 7
        <service id="knpu_lorem_ipsum.knpu_ipsum" class="KnpU\LoremIpsumBundle\KnpUIpsum" public="true" />
    ... lines 9 - 13
```

And even though services are private by default, you should also add public="false" to the others. This will make your services *also* behave the same on Symfony 3, where they are *public* by default.

```
13 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml
    ... lines 1 - 9
        <service id="KnpU\LoremIpsumBundle\KnpUIpsum" alias="knpu_lorem_ipsum.knpu_ipsum" public="false" />
    ... lines 11 - 13
```

This makes no difference in our app - it all still works.

Alright! With our services configured, let's talk about how we can allow the user to control the *behavior* of those services via configuration.

Chapter 4: All about the Bundle Extension Config System

We're not passing *any* arguments to the service... but this class *does* have two *very* important arguments: whether or not unicorns are real and the minimum times the word sunshine should appear in each paragraph. But what if a user of our bundle wants *more* sunshine or - gasp - they don't believe in unicorns? Right now, there's *no* way for them to control these arguments.

So if the *bundle* is responsible for registering the services & passing its arguments, how can the *user* of that bundle *control* those arguments? The answer lives in the config/packages directory.

Some important notes: first, our app automatically loads & processes *all* .yaml files it finds in this directory. Second, the *names* of these files are *not* important: you could rename them to *anything* else, .yaml. And *third*, the *entire* purpose of these files is to control the services that are provided by different bundles. When Symfony sees the framework key, it passes this configuration to the FrameworkBundle, which uses it to modify the services it provides.

The same for monolog: this config is passed to MonologBundle and it uses that when registering its services.

Creating a New Config File

Create a new file: knpu_lorem_ipsum.yaml - but, we could call this anything. And just to see what will happen, add some fake config: foo:, then bar: true.

3 lines | config/packages/knpu_lorem_ipsum.yaml

```yaml
foo:
    bar: true
```

Find your browser and refresh! Error! Check out the language carefully. It says that there is no *extension* able to load the configuration for "foo". We *know* that word extension: we just created our *own* extension: KnpULoremIpsumExtension.

Then, since foo is apparently invalid, it lists a bunch of *valid* keys, like framework, web_server, twig, etc. Here's the deal: when Symfony sees a root key like framework, it looks at *all* of the bundles, well, really, the *extension* class for each bundle, to see if there is one called FrameworkExtension. If there *is*, it passes the config to it. If there is *not*, it throws this big, hairy, ugly exception.

Passing Config to our Extension

But check this out: go back to the list of valid keys. Thanks to our KnpULoremIpsumExtension class, there's one called knp_u_lorem_ipsum! Change the root key to use *that* instead.

3 lines | config/packages/knpu_lorem_ipsum.yaml

```yaml
knp_u_lorem_ipsum:
// ... lines 2 - 3
```

Next, open our extension class, var_dump($configs) and die.

19 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```php
// ... lines 1 - 9
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        var_dump($configs);die;
// ... lines 15 - 16
    }
}
```

Try it out! No error! And cool! That bar: true value is passed to the load method! We're one step closer to *using* that config to tweak our service.

But, there are two weird things. First, the root key is... uh... not perfect. The knp_u_ is.. weird - I want it be knpu_... but apparently Symfony disagrees: our extra capital "U" is confusing things. We'll fix this in a bit.

The second weird thing is that the $configs value that's passed to load() is *not* just a simple array with bar=true. Nope, it's an *array* of arrays. Inception. Why? Well, it's possible that the user could add configuration for our bundle in *multiple* files. Like, we could have a dev environment-specific YAML file. When that happens, instead of merging that config together, it would pass us the configuration from *both* files. For example, if knp_u_lorem_ipsum existed in *three* different files, this array would have *three* different arrays inside. And, yep! It will be our job to merge them together. But, that's actually going to be *really* cool.

But before we do that, let's fix our alias to be knpu_lorem_ipsum. It's not something you *often* need to worry about, but the fix is *super* interesting.

Chapter 5: Custom Extension Alias

When you create an extension class, Symfony automatically calculates a "root" config key for it. In our case, it calculated knp_u_lorem_ipsum... it generated this based on our class name. I'd *rather* have knpu_lorem_ipsum. But of course, that doesn't work... yet.

This root key is called the extension *alias*. And *we* can *totally* control it. How? In our extension class, go to the Code->Generate menu, or Cmd+N on a mac, select "Override" methods, and choose getAlias(). Then, return knpu_lorem_ipsum.

24 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 9
class KnpULoremIpsumExtension extends Extension
{
... lines 12 - 18
    public function getAlias()
    {
        return 'knpu_lorem_ipsum';
    }
}
```

Here's how things *really* work. When Symfony boots, it loops over all the extension classes in the system, calls getAlias() on each, and *this* becomes their config key. In the parent class, well, the *parent's* parent class, there is a *default* getAlias() method which... surprise! Removes the Extension suffix, and "underscores" what's left.

Cool! Easy fix! Find your browser and refresh to celebrate! Boo! Another error:

> Users will expect the alias of the default extension of a bundle to be the underscored version of the bundle name. You can override some method if you want to use another alias.

How Bundles Load Extensions

Ok. This is a bit odd, but, on the bright side, it'll give us a chance to do some exploring! Open up our bundle class. It's empty... but it actually does a *bunch* of cool things. Hold Command or Ctrl and click to open the base class. One of the methods is called getContainerExtension().

When Symfony builds the container, it loops over all bundle classes and calls this method, which returns the extension object. Check out the createContainerExtension() method, well, actually, the getContainerExtensionClass() method. Ah! *This* is the reason why Symfony expects our extension to live in the DependencyInjection directory and to end in the word Extension. *All* that magic comes from overrideable methods on our bundle class.

Scroll back up to getContainerExtension(). After it creates the container extension, it does a sanity check: if the alias is different than it expected, it throws an exception. This was originally added to prevent bundle authors from going *crazy* and creating custom aliases like delicious_pizza or beam_me_up_scotty.

But, it's kind of annoying. The fix is easy. In our bundle class, go to the Code -> Generate menu, or Cmd + N on a Mac, select Override Methods and choose getContainerExtension.

Then, if null === $this->extension, set $this->extension to a new KnpULoremIpsumExtension. Return $this->extension at the bottom.

22 lines | lib/LoremIpsumBundle/src/KnpULoremIpsumBundle.php

```
... lines 1 - 7
class KnpULoremIpsumBundle extends Bundle
{
    /**
     * Overridden to allow for the custom extension alias.
     */
    public function getContainerExtension()
    {
        if (null === $this->extension) {
            $this->extension = new KnpULoremIpsumExtension();
        }

        return $this->extension;
    }
}
```

This does the same thing as the parent method, but without that sanity check.

Let's do it... refresh! Our custom alias is alive!!!

Now, it's time to use this $configs array to start allowing our end-users to modify our service. *This* is one of my *favorite* parts.

Chapter 6: Bundle Configuration Class

The KnpUIpsum class *has* two constructor args, but the user can't control these... yet. In knpu_lorem_ipsum.yaml, here's my idea: allow the user to use two new config keys, like unicorns_are_real and min_sunshine, and pass those values to our service as arguments.

Comment-out the var_dump. Symfony's configuration system is *smart*: all the keys are *validated*. If you typo a key - like secret2 under framework, when you refresh, you get a big ol' error! Yep, *each* bundle creates its own "tree" of *all* the valid config keys.

In fact, find your terminal. Run:

```
$ php bin/console config:dump framework
```

This is an example of the *entire* tree of valid configuration for framework! This is *amazing*, and it's made possible by a special Configuration class. It's time to create our own!

Creating the Configuration Class

Inside the DependencyInjection directory, create a new class called Configuration. Make this implement ConfigurationInterface: the one from the Config component. We'll need to implement one method: go to the Code -> Generate menu, or Cmd+N on a Mac, select "Implement Methods" and choose getConfigBuilder().

12 lines | lib/LoremIpsumBundle/src/DependencyInjection/Configuration.php

```
... lines 1 - 6
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
    }
}
```

This is one of the *strangest* classes you'll ever see. By using PHP code, we're going to define the entire *tree* of valid config that can be passed to our bundle.

A *great* way to see how this class works is to look at an existing one! Type Shift+Shift to open a class called FrameworkExtension, deep in the core of Symfony. Yep, this is the extension class for FrameworkBundle! It has the same load() method as *our* extension.

In the same directory, if you click on the top tree, you'll find a class called Configuration. Inside, it defines *all* of the valid config keys with a, sort of, nested tree format. This is a super powerful and, honestly, super complex system. We're only going to use a few basic features. If you need to define a more complex config tree, *definitely* steal, um, borrow, from these core classes.

Building the Config Tree

Back in *our* class, start with $treeBuilder = new TreeBuilder(). Then, $rootNode = $treeBuilder->root() and pass the name of our key: knpu_lorem_ipsum.

```
23 lines | lib/LoremIpsumBundle/src/DependencyInjection/Configuration.php
    ... lines 1 - 9
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('knpu_lorem_ipsum');
    ... lines 14 - 21
    }
```

Tip

Since Symfony 4.3 you should pass the root node name to the TreeBuilder instead:

```
$treeBuilder = new TreeBuilder('knpu_lorem_ipsum');
$rootNode = $treeBuilder->getRootNode();
// ...
```

Now... just start building the config tree! $rootNode->children(), and below, let's create two keys. The first will be for the "unicorns are real" value, and it should be a boolean. To add that, say ->booleanNode('unicorns_are_real'), ->defaultTrue() and to finish configuring this node, ->end().

```
23 lines | lib/LoremIpsumBundle/src/DependencyInjection/Configuration.php
    ... lines 1 - 9
    public function getConfigTreeBuilder()
    {
    ... lines 12 - 13
        $rootNode
            ->children()
                ->booleanNode('unicorns_are_real')->defaultTrue()->end()
    ... lines 17 - 21
    }
```

The other option will an integer: ->integerNode('min_sunshine'), default it to 3, then ->end(). Call ->end() one more time to finish the children().

```
23 lines | lib/LoremIpsumBundle/src/DependencyInjection/Configuration.php
    ... lines 1 - 13
        $rootNode
            ->children()
    ... line 16
                ->integerNode('min_sunshine')->defaultValue(3)->end()
            ->end()
        ;
    ... lines 20 - 23
```

Weird, right!? Return the $treeBuilder at the bottom.

```
23 lines | lib/LoremIpsumBundle/src/DependencyInjection/Configuration.php
    ... lines 1 - 9
    public function getConfigTreeBuilder()
    {
    ... lines 12 - 20
        return $treeBuilder;
    }
```

Using the Configuration Class

In our extension, we can use this to validate and merge all the config together. Start with $configuration = $this->getConfiguration() and pass this $configs and the container. This simply instantiates the Configuration class.

27 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 9
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 14 - 16
        $configuration = $this->getConfiguration($configs, $container);
... lines 18 - 19
    }
... lines 21 - 25
}
```

Here's the *really* important part: $config = $this->processConfiguration(): pass the configuration object and the original, raw array of $configs. var_dump() that final config and die!

27 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 11
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 14 - 17
        $config = $this->processConfiguration($configuration, $configs);
        var_dump($config);die;
    }
... lines 21 - 27
```

Let's see what happens! Find your browser and... refresh! We get an error... which is awesome! It says:

> Unrecognized option "bar" under "knpu_lorem_ipsum"

This is telling us:

> Yo! "bar" is not one of the valid config keys!

Back in knpu_lorem_ipsum.yaml, temporarily comment-out *all* of our config. And, refresh again. Yes! No error! Instead, we see the final, validated & normalized config, with the *two* keys we created in the Configuration class.

3 lines | config/packages/knpu_lorem_ipsum.yaml

```
#knpu_lorem_ipsum:
#    bar: true
```

Put *back* the config, but use a real value: min_sunshine set to 5.

3 lines | config/packages/knpu_lorem_ipsum.yaml

```
knpu_lorem_ipsum:
    min_sunshine: 5
```

Refresh one last time. Woohoo! min_sunshine equals 5. These Configuration classes are strange... but they take care of everything: validating, merging and applying default values.

Dynamically Setting the Arguments

We are *finally* ready to *use* this config. But... how? The service & its arguments are defined in services.xml... so we can't just magically reference those dynamic config values here.

Copy the service id and go back to the extension class. That container builder holds the *instructions* on how to instantiate our service - like its class and what constructor arguments to pass to it. And we - right here in PHP - can *change* those.

Check it out: start with $definition = $container->getDefinition() and pass the service id. This returns a Definition object, which holds the service's class name, arguments and a bunch of other stuff. *Now* we can say $definition->setArgument(): set the first argument - which is index 0 - to $config['']. The first argument is $unicornsAreReal. So use the unicorns_are_real key. Set the second argument - index one - to min_sunshine.

30 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 9
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 14 - 19
        $definition = $container->getDefinition('knpu_lorem_ipsum.knpu_ipsum');
        $definition->setArgument(0, $config['unicorns_are_real']);
        $definition->setArgument(1, $config['min_sunshine']);
    }
... lines 24 - 28
}
```

That's it! Go back and refresh! It works! Sunshine now appears at least 5 times in every paragraph. Our dynamic value *is* being passed!

Oh, and, bonus! In your terminal, run config:dump again, but *this* time pass it knpu_lorem_ipsum:

```
$ php bin/console config:dump knpu_lorem_ipsum
```

Yes! Our bundle now prints its config thanks to the Configuration class. If you want to get *really* fancy - which of course we *do* - you can add documentation there as well. Add ->info() and pass a short description about why you would set this. Do the same for min_sunshine.

23 lines | lib/LoremIpsumBundle/src/DependencyInjection/Configuration.php

```
... lines 1 - 7
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
... lines 12 - 13
        $rootNode
            ->children()
                ->booleanNode('unicorns_are_real')->defaultTrue()->info('Whether or not you believe in unicorns')->end()
                ->integerNode('min_sunshine')->defaultValue(3)->info('How much do you like sunshine?')->end()
... lines 18 - 21
    }
}
```

Run config:dump again:

```
$ php bin/console config:dump knpu_lorem_ipsum
```

Pretty, freakin' cool.

Next, let's get fancier with our config and allow entire *services* to be swapped out.

Chapter 7: Allowing Entire Services to be Overridden

When you create a reusable library, you gotta think about what *extension* points you want to offer your users. Right now, the user can control the two arguments to this class... but they can't control anything else, like the actual *words* that are used in our fake text. These are hardcoded at the bottom.

So... how *could* we allow the user to *override* these? One option that I like is to *extract* this code into its own class, and allow the user to *override* that class *entirely*.

Check this out: in the bundle, create a new class called KnpUWordProvider. Give it a public function called getWordList() that will return an array. Back in KnpUIpsum, steal the big word list array and... return that from the new function.

143 lines | lib/LoremIpsumBundle/src/KnpUWordProvider.php

```
... lines 1 - 4
class KnpUWordProvider
{
    public function getWordList(): array
    {
        return [
            'adorable',
            'active',
            'admire',
            'adventurous',
... lines 14 - 140
        ];
    }
}
```

Perfect! In KnpUIpsum, add a new constructor argument and type-hint it with KnpUWordProvider. Make it the first argument, because it's required. Create a new property for this - $wordProvider - then set it below: $this->wordProvider = $wordProvider.

211 lines | lib/LoremIpsumBundle/src/KnpUIpsum.php

```
... lines 1 - 9
class KnpUIpsum
{
    private $wordProvider;
... lines 13 - 17
    public function __construct(KnpUWordProvider $wordProvider, bool $unicornsAreReal = true, $minSunshine = 3)
    {
        $this->wordProvider = $wordProvider;
... lines 21 - 22
    }
... lines 24 - 209
}
```

With all that setup, down below in the original method, just return $this->wordProvider->getWordList().

```
211 lines | lib/LoremIpsumBundle/src/KnpUIpsum.php
... lines 1 - 205
    private function getWordList(): array
    {
        return $this->wordProvider->getWordList();
    }
... lines 210 - 211
```

Our class is now *more* flexible than before. Of course, in services.xml, we need to tell Symfony to pass in that new argument! Copy the existing service node so that we can register the new provider as a service first. Call this one knpu_lorem_ipsum.knpu_word_provider and set the class to KnpUWordProvider. Oh, but this service does *not* need to be public: no one should need to use this service directly.

```
17 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml
... lines 1 - 6
    <services>
... lines 8 - 11
        <service id="knpu_lorem_ipsum.knpu_word_provider" class="KnpU\LoremIpsumBundle\KnpUWordProvider" />
... lines 13 - 14
    </services>
... lines 16 - 17
```

Above, we need to *stop* using the short service syntax. Instead, add a closing service tag. Then, add an argument with type="service" and id="knpu_lorem_ipsum.knpu_word_provider".

```
17 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml
... lines 1 - 6
    <services>
        <service id="knpu_lorem_ipsum.knpu_ipsum" class="KnpU\LoremIpsumBundle\KnpUIpsum" public="true">
            <argument type="service" id="knpu_lorem_ipsum.knpu_word_provider" />
        </service>
... lines 11 - 14
    </services>
... lines 16 - 17
```

If you're used to configuring services in YAML, the type="service" is equivalent to putting an @ symbol before the service id. The *last* change we need to make is in the extension class. These are now the second and third arguments, so use the indexes one and two.

```
30 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php
... lines 1 - 9
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 14 - 20
        $definition->setArgument(1, $config['unicorns_are_real']);
        $definition->setArgument(2, $config['min_sunshine']);
    }
... lines 24 - 28
}
```

Phew! Unless we messed something up, it should work! Try it! Yes! We *still* get fresh words each time.

Making the Word Provider Configurable

So... we refactored our code to be more flexible... but it's *still* not possible for the user to override the word provider. Here's my idea: in the Configuration class, add a new *scalar* node - in other words, a *string* option - called word_provider. Default this to null, and you can add some documentation to be *super* cool. If the user wants to customize the word list, they will set this to the service *id* of their *own* word provider.

24 lines | lib/LoremIpsumBundle/src/DependencyInjection/Configuration.php

```
... lines 1 - 7
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
... lines 12 - 13
        $rootNode
            ->children()
... lines 16 - 17
                ->scalarNode('word_provider')->defaultNull()->end()
... lines 19 - 22
    }
}
```

So, in the extension class, if the that value is *not* set to null, let's *replace* the first argument entirely: $definition->setArgument() with 0 and $config['word_provider'].

33 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 9
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 14 - 20
        if (null !== $config['word_provider']) {
            $definition->setArgument(0, $config['word_provider']);
        }
... lines 24 - 25
    }
... lines 27 - 31
}
```

Creating our Custom Word Provider

We're *not* setting this config value yet, but when we refresh, great! We at least didn't *break* anything... though we *do* have a small mistake...

Anyways, let's test the system properly by creating our own, new word provider. In src/Service, create a class called CustomWordProvider. Make this extend the KnpUWordProvider because I just want to *add* something to the core list. To override the method, go to the Code -> Generate menu, or Cmd+N on a Mac - choose "Override methods" and select getWordList().

17 lines | src/Service/CustomWordProvider.php

```
... lines 1 - 6
class CustomWordProvider extends KnpUWordProvider
{
    public function getWordList(): array
    {
... lines 11 - 14
    }
}
```

Inside, set $words = parent::getWordList(). Then, add the word "beach"... because we all deserve a little bit more beach in our lives. Return $words at the bottom.

17 lines | src/Service/CustomWordProvider.php

```
... lines 1 - 8
    public function getWordList(): array
    {
        $words = parent::getWordList();
        $words[] = 'beach';

        return $words;
    }
... lines 16 - 17
```

Thanks to the standard service configuration in our app, this class is already registered as a service. So all *we* need to do is go into the config/packages directory, open knpu_lorem_ipsum.yaml, and set word_provider to App\Service\CustomWordProvider.

4 lines | config/packages/knpu_lorem_ipsum.yaml

```
knpu_lorem_ipsum:
... line 2
    word_provider: App\Service\CustomWordProvider
```

Let's see if this thing works! Move over and refresh! Boooo!

> Argument 1 passed to KnpUIpsum::__construct() must be an instance of KnpUWordProvider - because that's our type-hint - *string* given.

Look below in the stack-trace: this is pretty deep code, but you can actually see that something is creating a new KnpUIpsum, but passing the string *class* name of our provider as the first argument... not the service!

Go back to our extension class. Here's the fix: when we set the argument to $config['word_provider'], this *of course* sets that argument to the *string* value! To fix this in YAML, we would prefix the service id with the @ symbol. In PHP, wrap the value in a new Reference() object. *This* tells Symfony that we're referring to a *service*.

34 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 10
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 15 - 22
        $definition->setArgument(0, new Reference($config['word_provider']));
... lines 24 - 26
    }
... lines 28 - 32
}
```

Deep breath and, refresh! It works! And if you search for "beach"... yes! Let's go to the beach!

This is a great step! But there are two other nice improvements we can make: using a service alias & introducing an interface. Let's add those next.

Chapter 8: Extensibility with Interfaces & Aliases

I want to make two other changes to the new "word provider" setup. The first is optional: it's another common method for making the word provider configurable.

Go back into our services.xml file. Right now, we set the first argument inside of the XML file, then override that argument in the extension class, if a different value is provided. Another option - and we'll talk about the advantages later - is to use a service *alias*.

Copy the alias we created earlier in order to enable autowiring. Create a new alias whose id is knpu_lorem_ipsum.word_provider and set the alias to the knp_word_provider service id above.

18 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 6
    <services>
... lines 8 - 13
        <service id="knpu_lorem_ipsum.word_provider" alias="knpu_lorem_ipsum.knpu_word_provider" public="false" />
... line 15
    </services>
... lines 17 - 18
```

Thanks to this, there is now a *new* service in the container called knpu_lorem_ipsum.word_provider. But when someone references it, it actually just points to our knpu_lorem_ipsum.knpu_word_provider. Now, for the argument to KnpUIpsum, pass the *alias* id instead.

18 lines | lib/LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 7
        <service id="knpu_lorem_ipsum.knpu_ipsum" class="KnpU\LoremIpsumBundle\KnpUIpsum" public="true">
            <argument type="service" id="knpu_lorem_ipsum.word_provider" />
        </service>
... lines 11 - 18
```

So far, this won't change *anything*. But open the extension class. Instead of changing the argument, we can override the *alias* to point to *their* service id. Do this with $container->setAlias(). First pass knpu_lorem_ipsum.word_provider and set this alias to $config['word_provider']. We don't need the new Reference() here because the setAlias() method expects this to be a service ID.

34 lines | lib/LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 10
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 15 - 21
        if (null !== $config['word_provider']) {
            $container->setAlias('knpu_lorem_ipsum.word_provider', $config['word_provider']);
        }
... lines 25 - 26
    }
... lines 28 - 32
}
```

And before even trying it, copy the service alias, find your terminal, and run:

```
$ php bin/console debug:container --show-private knpu_lorem_ipsum.word_provider
```

Yes! This is an alias to our CustomWordProvider. And *that* means that the first argument to KnpUIpsum will use that. Refresh to make sure it still works. It does!

There's no *amazing* reason to use this alias strategy versus what we had before, but there are two minor advantages. First, *if* we needed to reference the word provider service in multiple places - probably in services.xml - using an alias is *easier*, because you don't need to remember to, for example, replace 5 different arguments where the service is used. And second, *if* we wanted this service to be used *directly* by our users, creating an alias is the *only* way to give them a service id they can reference, *even* if they override the word provider to be something else.

Creating a WordProviderInterface

Ok, our setup is really, really nice. But there is *one* restriction we're putting on our user that we really do *not* need to! Open KnpUIpsum and scroll all the way to the constructor. The first argument is type-hinted with KnpUWordProvider. This means that if the user wants to create their *own* word provider, they *must* extend our *original* KnpUWordProvider. We *are* doing this... because we just want to add a new word to the list, but this should *not* be required! All *we* care about is that the service has a getWordList() method that returns an array.

In other words, this is the *perfect* use-case for an interface! Wooo! In the bundle, create a new PHP class. Call it WordProviderInterface and change the "kind" from class to interface.

Inside, add the getWordList() method and make it return an array. This is *also* the perfect place to add some documentation about what this method should do.

14 lines | lib/LoremIpsumBundle/src/WordProviderInterface.php

```
... lines 1 - 4
interface WordProviderInterface
{
    /**
     * Return an array of words to use for the fake text.
     *
     * @return array
     */
    public function getWordList(): array;
}
```

With the interface done, go back to KnpUIpsum, change the type-hint to WordProviderInterface. The user can now pass *anything* they want, as long as it has this getWordList() method... because *that* is what we're using at the bottom of KnpUIpsum.

211 lines | lib/LoremIpsumBundle/src/KnpUIpsum.php

```
... lines 1 - 9
class KnpUIpsum
{
... lines 12 - 17
    public function __construct(WordProviderInterface $wordProvider, bool $unicornsAreReal = true, $minSunshine = 3)
... lines 19 - 209
}
```

Then, of course, we also need to go open *our* provider and make sure it implements this interface: implements WordProviderInterface.

```
143 lines | lib/LoremIpsumBundle/src/KnpUWordProvider.php
... lines 1 - 4
class KnpUWordProvider implements WordProviderInterface
{
... lines 7 - 142
}
```

If you try it now... *not* broken! And yea, *our* CustomWordProvider will *still* extend KnpUWordProvider, but that's now optional - we could just implement the interface directly.

Next, let's take a big step and move our bundle *out* of our code and give it it's *own* composer.json file!

Chapter 9: Proper Bundle composer.json File

We put the bundle into our app temporarily because it made it *really* easy to hack on the bundle, test in the app and repeat.

But now that it's getting kinda stable, it's time to move the bundle into its own directory with its *own* repository. It's like watching your kid grow up, and *finally* move into their own apartment.

Find your terminal, and kick that lazy bundle out of your house and into a new directory next door:

```
$ mv lib/LoremIpsumBundle ../LoremIpsumBundle
```

In PhpStorm, let's open that second directory inside a new window, and re-decorate things a little bit. Ok, a lot to keep track of: application code, bundle code and terminal. To confuse things more, open a *third* terminal tab and move it into the bundle, which, sadly, does *not* have a git repository yet!

Let's add one!

```
$ git init
$ git status
```

Add everything and commit!

```
$ git add .
$ git commit -m "Unicorns"
```

Bootstrapping composer.json

To make this a shareable package, it needs its very-own composer.json file. To create it, run:

```
$ composer init
```

Let's call it knpuniversity/lorem-ipsum-bundle, give it a description, make sure the author is correct, leave minimum-stability alone and, for "Package Type" - this is important! - use symfony-bundle. That's needed so that Flex will automatically enable the bundle when it's installed. For License, I'll use MIT - but more on that later. And finally, let's *not* add any dependencies yet. And, generate! Let's *definitely* ignore the vendor/ directory.

14 lines | LoremIpsumBundle/composer.json

```
{
    "name": "knpuniversity/lorem-ipsum-bundle",
    "description": "Happy lorem ipsum",
    "type": "symfony-bundle",
    "license": "MIT",
    "authors": [
        {
            "name": "Ryan Weaver",
            "email": "ryan@knpuniversity.com"
        }
    ],
    "require": {}
}
```

Hello .gitignore file and hello composer.json! This file has a few purposes. First, of course, it's where we will eventually require any packages the bundle needs. We'll do that later. But I am going to start at least by saying that we require php 7.1.3. That's the version that Symfony 4.0 requires.

16 lines | LoremIpsumBundle/composer.json

```
    ... lines 1 - 11
    "require": {
        "php": "^7.1.3"
    }
    ... lines 15 - 16
```

Autoloading Rules

Second, the composer.json file is where *we* define our autoloading rules: Composer needs to know what namespace our bundle uses and where those classes live.

Up until now, we put those autoload rules inside the main project. Let's steal that section and remove the line for our bundle. Paste that into the bundle and remove the App line. The KnpU\\LoremIpsumBundle\\ namespace lives in just, src/.

21 lines | LoremIpsumBundle/composer.json

```
    ... lines 1 - 14
    "autoload": {
        "psr-4": {
            "KnpU\\LoremIpsumBundle\\": "src/"
        }
    }
    ... lines 20 - 21
```

Using a "path" Repository

So... yay! We have a standalone bundle with its own repository! But, I'm not *quite* ready to push this to Packagist yet... and I kinda want to keep testing it inside my app. But, how? We can't composer require it until it lives on Packagist, right?

Well, there *is* one trick. Google for "composer path package".

Click on the "Repositories" documentation and... *all* the way at the bottom... there's a path option! This allows us to point to any directory on our computer that contains a composer.json file. Then, suddenly, *that* library becomes available to composer require.

Copy the repositories section, find our application's composer.json and, at the bottom, paste this. The library lives at ../LoremIpsumBundle.

Tip

The course code contains LoremIpsumBundle project inside itself, hence you won't see ../ on the repository URL in the code blocks.

```
83 lines | composer.json
... lines 1 - 75
    "repositories": [
        {
            "type": "path",
            "url": "LoremIpsumBundle"
        }
    ]
... lines 82 - 83
```

Thanks to that, our application *now* knows that there is a package called knpuniversity/lorem-ipsum-bundle available. Back at the terminal, find the tab for our application and composer require knpuniversity/lorem-ipsum-bundle, with a :*@dev at the end.

```
$ composer require knpuniversity/lorem-ipsum-bundle:*@dev
```

A path package isn't quite as smart as a normal package: you don't have versions or anything like that: it just uses whatever code is in that directory. This tells Composer to require that package, but not worry about the version.

And, cool! On my system, it installed with a symlink, which means we can keep hacking on the bundle and testing it live in the app.

Oh, and since Symfony flex noticed that our package has a symfony-bundle type, it actually tried to configure a recipe, which would normally enable the bundle for us in bundles.php. It didn't this time, only because we already have that code.

Now that everything is reconnected, it should work! Refresh the page. Yes! That bundle is properly living on its own.

Next, we actually already have some tests for our bundle... but they still live in the app. Let's move these into the bundle and start talking about properly adding the dependencies that it needs.

Chapter 10: Testing the Bundle

Hey! Someone already made some tests for our bundle! *So* nice! Right now, they live in the *app*, but moving them *into* the bundle is our next job! But first... let's make sure they're still working.

66 lines | tests/Service/KnpUIpsumTest.php

```
... lines 1 - 2
namespace App\Tests\Service;
... lines 4 - 7
class KnpUIpsumTest extends TestCase
{
    public function testGetWords()
    {
        $ipsum = new KnpUIpsum();

        $words = $ipsum->getWords(1);
        $this->assertInternalType('string', $words);
        $this->assertCount(1, explode(' ', $words));

        $words = $ipsum->getWords(10);
        $this->assertCount(10, explode(' ', $words));

        $words = $ipsum->getWords(10, true);
        $this->assertCount(10, $words);
    }
... lines 24 - 65
}
```

Find the terminal tab for the application and run:

```
$ ./vendor/bin/simple-phpunit
```

The first time you run this, it'll download PHPUnit behind the scenes. Then... it does *not* pass!

> Class App\Service\KnpUIpsum not found

Of course! When we moved this class into the new namespace, we did *not* update the test! No problem - just re-type KnpUIpsum and hit tab to auto-complete and get the new use statement.

67 lines | tests/Service/KnpUIpsumTest.php

```
... lines 1 - 4
use KnpU\LoremIpsumBundle\KnpUIpsum;
... lines 6 - 67
```

Perfect! But... I can already see another problem! When we added the first constructor argument to KnpUIpsum, we *also* didn't update the test. I could use mocking here, but it's just as easy to say new KnpUWordProvider. Repeat that in the two other places.

```
67 lines | tests/Service/KnpUIpsumTest.php
... lines 1 - 5
use KnpU\LoremIpsumBundle\KnpUWordProvider;
... lines 7 - 8
class KnpUIpsumTest extends TestCase
{
    public function testGetWords()
    {
        $ipsum = new KnpUIpsum(new KnpUWordProvider());
... lines 14 - 23
    }
... line 25
    public function testGetSentences()
    {
        $ipsum = new KnpUIpsum(new KnpUWordProvider());
... lines 29 - 37
    }
... line 39
    public function testGetParagraphs()
    {
... lines 42 - 43
        for ($i = 0; $i < 100; $i++) {
            $ipsum = new KnpUIpsum(new KnpUWordProvider());
... lines 46 - 64
        }
    }
}
```

Ok, try those tests again!

```
$ ./vendor/bin/simple-phpunit
```

Got it!

Adding Tests to your Bundle & autoload-dev

Time to move this into our bundle. We already have a src/ directory. Now create a new directory next to that called tests/. Copy the KnpUIpsumTest and put that directly in this new folder. I'm putting it *directly* in tests/ because the KnpUIpsum class itself lives directly in src/.

And the test file is now gone from the app.

But really... we shouldn't need to update much... or *anything* in the test class itself. In fact, the *only* thing we need to change is the namespace. Instead of App\Tests\Services, start with the same namespace as the rest of the bundle. So, KnpU\LoremIpsumBundle\Tests.

```
67 lines | LoremIpsumBundle/tests/KnpUIpsumTest.php
... lines 1 - 2
namespace KnpU\LoremIpsumBundle\Tests;
... lines 4 - 67
```

But, if we're going to start putting classes in the tests/ directory, we need to make sure that Composer can autoload these files. This isn't strictly required to make PHPUnit work, but it *will* be needed if you add any helper or dummy classes to the directory and want to use them in your tests.

And, it's easy! We basically want to add a second PSR-4 rule that says that the KnpU\LoremIpsumBundle\Tests namespace lives in the tests/ directory. But... don't! Instead, copy the entire section, paste and rename it to autoload-dev. Change the namespace to end in Tests\\ and point this at the tests/ directory.

26 lines | LoremIpsumBundle/composer.json

```
... lines 1 - 19
    "autoload-dev": {
        "psr-4": {
            "KnpU\\LoremIpsumBundle\\Tests\\": "tests/"
        }
    }
... lines 25 - 26
```

Why autoload-dev? The issue is that our end users will *not* be using anything in the tests/ directory: this config exists *just* to help us when we are working directly on the bundle. By putting it in autoload-dev, the autoload rules for the tests/ directory will *not* be added to the autoload matrix of our users' applications, which will give them a slight performance boost.

Installing symfony/phpunit-bridge

Ok: our test is ready. So let's run it! Move over to the terminal for the bundle and run... uh... wait a second. Run, what? We haven't installed PHPUnit! Heck, we don't even have a vendor/ directory yet. Sure, you *can* run composer install to get a vendor/ directory... but with nothing inside.

This should be no surprise: if we want to test our bundle, the bundle *itself* needs to require PHPUnit. Go back to the terminal and run:

```
$ composer require symfony/phpunit-bridge --dev
```

Two important things. First, we're using Symfony's PHPUnit bridge because it has a few extra features... and ultimately uses PHPUnit behind-the-scenes. Second, just like with autoloading, our end users do *not* need to have symfony/phpunit-bridge installed in *their* vendor directory. We *only* need this when we're working on the bundle itself. By adding it to require-dev, when a user installs our bundle, it will not *also* install symfony/phpunit-bridge.

Ignoring composer.lock

Now that we've run composer install, we have a composer.lock file! So, commit it! Wait, don't! Libraries and bundles should actually *not* commit this file - there's just no purpose to lock the dependencies: it doesn't affect our end-users at all. Instead, open the .gitignore file and ignore composer.lock. Now when we run git status, yep! It's gone.

3 lines | LoremIpsumBundle/.gitignore

```
... line 1
composer.lock
```

phpunit.xml.dist

Ok, let's *finally* run the tests!

```
$ ./vendor/bin/simple-phpunit
```

It - of course - downloads PHPUnit behind the scenes the first time and then... nothing! It... just prints out the options??? What the heck? Well... our bundle doesn't have a phpunit.xml.dist file yet... so it has *no* idea *where* our test files live or anything else!

A good phpunit.xml.dist file is pretty simple... and I usually steal one from a bundle I trust. For example, Go to github.com/knpuniversity/oauth2-client-bundle. Find the phpunit.xml.dist file, view the raw version and copy it. Back at our bundle, create that file and paste it in.

28 lines | LoremIpsumBundle/phpunit.xml.dist

```
<?xml version="1.0" encoding="UTF-8"?>

<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://schema.phpunit.de/4.1/phpunit.xsd"
         backupGlobals="false"
         colors="true"
         bootstrap="./vendor/autoload.php"
        >
... lines 9 - 26
</phpunit>
```

Oh, and before I forget, in .gitignore, *also* ignore phpunit.xml. The .dist version *is* committed, but this allows anyone to have a custom version on their local copy that they do not commit.

4 lines | LoremIpsumBundle/.gitignore

```
... lines 1 - 2
phpunit.xml
```

Check out the new file: the really important thing is that we set the bootstrap key to vendor/autoload.php so that we get Composer's autoloading. This also sets a few php.ini settings and... yes: we tell PHPUnit *where* our test files live.

Now I think it *will* work. Find your terminal and try it again:

```
$ ./vendor/bin/simple-phpunit
```

It passes! Woo!

After seeing these fancy green colors, you *might* be thinking that our bundle is working! And if you did... you'd be *half* right. Next, we'll build a functional test... which is *totally* going to fail.

Chapter 11: Service Integration Test

Thanks to the unit test, we can confidently say that the KnpUIpsum class works correctly. But... that's only like 10% of our bundle's code! *Most* of the bundle is related to service configuration. So what guarantees that the bundle, extension class, Configuration class and services.xml files are all correct? Nothing! Yay!

And it's not that we need to test *everything*, but it would be great to *at least* have a "smoke" test that made sure that the bundle correctly sets up a knpu_lorem_ipsum.knpu_ipsum service.

Bootstrapping the Integration Test

We're going to do that with a functional test! Or, depending on how you name things, this is really more of an integration test. Details. Anyways, in the tests/ directory, create a new class called FunctionalTest.

Make this extend the normal TestCase from PHPUnit, and add a public function testServiceWiring().

27 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 8
class FunctionalTest extends TestCase
{
    public function testServiceWiring()
    {

    }
}
... lines 16 - 27
```

And here is where things get interesting. We basically want to initialize our bundle into a real app, and check that the container has that service. But... we do *not* have a Symfony app lying around! So... let's make the *smallest* possible Symfony app ever.

To do this, we just need a Kernel class. And instead of creating a new *file* with a new class, we can hide the class right inside *this* file, because it's only needed here.

Add class KnpULoremIpsumTestingKernel extends Kernel from... wait... why is this not auto-completing the Kernel class? There *should* be one in Symfony's HttpKernel component! What's going on?

Dependencies: symfony/framework-bunde?

Remember! In our composer.json, other than the PHP version, the require key is empty! We're *literally* saying that someone is allowed to use this bundle even if they use *zero* parts of Symfony. That's not OK. We need to be explicit about what dependencies are *actually* required to use this bundle.

But... what dependencies are required, exactly? Honestly... most bundles simply require symfony/framework-bundle. FrameworkBundle provides all of the core services, like the router, session, etc. It *also* requires the http-kernel component, event-dispatcher and probably anything else that your bundle relies on.

Requiring FrameworkBundle is *not* a horrible thing. But, it's *technically* possible to use the Symfony framework *without* the FrameworkBundle, and some people *do* do this.

So we're going to take the *tougher*, more interesting road and *not* simply require that bundle. Instead, let's look at the actual components our code uses. For example, open the bundle class. Obviously, we depend on the http-kernel component. And in the extension class, we're using config and dependency-injection. In Configuration, nothing new: just config.

Ok! Our bundle needs the config, dependency-injection and http-kernel components. And by the way, this is *exactly* why we're writing the integration test! Our bundle is not setup correctly right now... but it wasn't very obvious.

Adding our Dependencies

In composer.json, add these: symfony/config at version ^4.0. Copy this and paste it two more times. Require symfony/dependency-injection and symfony/http-kernel.

32 lines | LoremIpsumBundle/composer.json

```
... lines 1 - 11
    "require": {
... line 13
        "symfony/config": "^4.0",
        "symfony/dependency-injection": "^4.0",
        "symfony/http-kernel": "^4.0"
    },
... lines 18 - 32
```

Now, find your terminal, and run:

```
$ composer update
```

Perfect! Once that finishes, we can go back to our functional test. Re-type the "l" on Kernel and... yes! *There* is the Kernel class from http-kernel.

This requires us to implement two methods. Go to the Code -> Generate menu - or Command + N on a Mac - click "Implement Methods" and choose the two.

27 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 2
namespace KnpU\LoremIpsumBundle\Tests;
... lines 4 - 16
class KnpULoremIpsumTestingKernel extends Kernel
{
    public function registerBundles()
    {
    }

    public function registerContainerConfiguration(LoaderInterface $loader)
    {
    }
}
```

Inside registerBundles, return an array and *only* enable *our* bundle: new KnpULoremIpsumBundle(). Since we're not dependent on any other bundles - like FrameworkBundle - we should, in theory, be able to boot an app with only this. Kinda cool!

38 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 26
    public function registerBundles()
    {
        return [
            new KnpULoremIpsumBundle(),
        ];
    }
... lines 33 - 38
```

And... that's it! Our app is ready. Back in testServiceWiring, add $kernel = new KnpULoremIpsumTestingKernel() and pass this test for the environment, thought that doesn't matter, and true for debug. Next, *boot* the kernel, and say $container = $kernel->getContainer().

38 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 10
class FunctionalTest extends TestCase
{
    public function testServiceWiring()
    {
        $kernel = new KnpULoremIpsumTestingKernel('test', true);
        $kernel->boot();
        $container = $kernel->getContainer();
... lines 18 - 21
    }
}
... lines 24 - 38
```

This is *great*! We just booted a *real* Symfony app. And now, we can makes sure our service exists. Add $ipsum = $container->get(), copy the id of our service, and paste it here. We can do this because the service is public.

Let's add some very basic checks, like $this->assertInstanceOf() that KnpUIpsum::class is the type of $ipsum. And also, $this->assertInternalType() that a string is what we get back when we call $ipsum->getParagraphs().

38 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 12
    public function testServiceWiring()
    {
... lines 15 - 18
        $ipsum = $container->get('knpu_lorem_ipsum.knpu_ipsum');
        $this->assertInstanceOf(KnpUIpsum::class, $ipsum);
        $this->assertInternalType('string', $ipsum->getParagraphs());
    }
... lines 23 - 38
```

The unit test *truly* tests this class - so we really only need a sanity check. I think it's time to try this! Find your terminal, and run:

```
$ ./vendor/bin/simple-phpunit
```

Yes! We're now *sure* that our service is wired correctly! So, this functional test didn't *fail* like I promised in the last chapter. But the point is this: before we added our dependencies, our bundle was *not* actually setup correctly.

And, woh! In the tests/ directory, we suddenly have a cache/ folder! That comes from our kernel: it caches files just like a normal app. To make sure this doesn't get committed, open .gitignore and ignore /tests/cache.

5 lines | LoremIpsumBundle/.gitignore

```
... lines 1 - 3
/tests/cache
```

Next, let's get a little more complex by testing that some of our configuration options work.

Chapter 12: Complex Config Test

There is *one* important part of the bundle that is *not* tested yet: our configuration. If the user sets the min_sunshine option, there's no test that this is correctly passed to the service.

And yea, again, you do *not* need to have a test for *everything*: use your best judgment. For configuration like this, there are *three* different ways to test it. First, you can test the Configuration class itself. That's a nice idea if you have some really complex rules. Second, you can test the extension class directly. In this case, you would pass different config arrays to the load() method and assert that the arguments on the service Definition objects are set correctly. It's a really low-level test, but it works.

And *third*, you can test your configuration with an integration test like we created, where you boot a real application with some config, and check the behavior of the final services.

If you *do* want to test the configuration class or the extension class, like always, a great way to do this is by looking at the core code. Press Shift+Shift to open FrameworkExtensionTest. If you did some digging, you'd find out that this test parses YAML files full of framework configuration, parses them, then checks to make sure the Definition objects are correct based on that configuration.

Try Shift + Shift again to open ConfigurationTest. There are a bunch of these, but the one from FrameworkBundle is a pretty good example.

Dummy Test Word Provider

We're going to use the third option: boot a *real* app with some config, and test the final services. Specifically, I want to test that the custom word_provider config works.

Let's think about this: to create a custom word provider, you need the class, like CustomWordProvider, you need to register it as a service - which is automatic in our app - and *then* you need to pass the service id to the word_provider option. We're going to do *all* of that, right here at the bottom of this test class. It's a little nuts, and that's exactly why we're talking about it!

Create a new class called StubWordList and make it implement WordProviderInterface. This will be our fake word provider. Go to the Code -> Generate menu, or Command + N on a Mac, and implement the getWordList() method. Just return an array with two words: stub and stub2.

74 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 2
namespace KnpU\LoremIpsumBundle\Tests;
... lines 4 - 66
class StubWordList implements WordProviderInterface
{
    public function getWordList(): array
    {
        return ['stub', 'stub2'];
    }
}
```

Next, copy the testServiceWiring() method, paste it, and rename it to testServiceWiringWithConfiguration(). Remove the last two asserts: we're going to work more on this in a minute.

74 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 12
class FunctionalTest extends TestCase
{
... lines 15 - 25
    public function testServiceWiringWithConfiguration()
    {
        $kernel = new KnpULoremIpsumTestingKernel([
            'word_provider' => 'stub_word_list'
        ]);
        $kernel->boot();
        $container = $kernel->getContainer();

        $ipsum = $container->get('knpu_lorem_ipsum.knpu_ipsum');
... line 35
    }
}
... lines 38 - 74
```

Configuring Bundles in the Kernel

Here's the tricky part: we're using the same kernel in two different tests... but we want them to *behave* differently. In the second test, I need to pass some extra configuration. This will look a bit technical, but just follow me through this.

First, inside the kernel, go back to the Code -> Generate menu, or Command + N on a Mac, and override the constructor. To simplify, instead of passing the environment and debug flag, just hard-code those when we call the parent constructor.

70 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 38
class KnpULoremIpsumTestingKernel extends Kernel
{
    public function __construct()
    {
        parent::__construct('test', true);
    }
... lines 45 - 60
}
... lines 62 - 70
```

Thanks to that, we can remove those arguments in our two test functions above. But *now*, add an optional array argument called $knpUIpsumConfig. This will be the configuration we want to pass under the knpu_lorem_ipsum key.

At the top of the kernel, create a new private variable called $knpUIpsumConfig, and then assign that in the constructor to the argument.

```
74 lines | LoremIpsumBundle/tests/FunctionalTest.php
... lines 1 - 38
class KnpULoremIpsumTestingKernel extends Kernel
{
    private $knpUIpsumConfig;

    public function __construct(array $knpUIpsumConfig = [])
    {
        $this->knpUIpsumConfig = $knpUIpsumConfig;
... lines 46 - 47
    }
... lines 49 - 64
}
... lines 66 - 74
```

Next, find the registerContainerConfiguration() method. In a normal Symfony app, *this* is the method that's responsible for parsing all the YAML files in the config/packages directory and the services.yaml file.

Instead of parsing YAML files, we can instead put all that logic into PHP with $loader->load() passing it a callback function with a ContainerBuilder argument. Inside of *here*, we can start registering services and passing bundle extension configuration.

```
74 lines | LoremIpsumBundle/tests/FunctionalTest.php
... lines 1 - 56
    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(function(ContainerBuilder $container) {
... lines 60 - 62
        });
    }
... lines 65 - 74
```

First, in all cases, let's register our StubWordList as a service: $container->register(), pass it any id - like stub_word_list - and pass the class: StubWordList::class. It doesn't need any arguments.

```
74 lines | LoremIpsumBundle/tests/FunctionalTest.php
... lines 1 - 58
        $loader->load(function(ContainerBuilder $container) {
            $container->register('stub_word_list', StubWordList::class);
... lines 61 - 62
        });
... lines 64 - 74
```

Next, we need to pass any custom knpu_lorem_ipsum bundle extension configuration. Do this with $container->loadFromExtension() with knpu_lorem_ipsum and, for the second argument, the array of config you want: $this->knpUIpsumConfig.

```
74 lines | LoremIpsumBundle/tests/FunctionalTest.php
... lines 1 - 58
        $loader->load(function(ContainerBuilder $container) {
... lines 60 - 61
            $container->loadFromExtension('knpu_lorem_ipsum', $this->knpUIpsumConfig);
        });
... lines 64 - 74
```

Basically, each test case can *now* pass whatever custom config they want. The first won't pass any, but the second will pass the word_provider key set to the service id: stub_word_list.

```
74 lines | LoremIpsumBundle/tests/FunctionalTest.php
... lines 1 - 12
class FunctionalTest extends TestCase
{
... lines 15 - 25
    public function testServiceWiringWithConfiguration()
    {
        $kernel = new KnpULoremIpsumTestingKernel([
            'word_provider' => 'stub_word_list'
        ]);
... lines 31 - 35
    }
}
... lines 38 - 74
```

The *downside* of an integration test is that we can't assert *exactly* that the StubWordList was passed into KnpUIpsum. We can only test the *behavior* of the services. But since that stub word list only uses two different words, we can reasonably test this with $this->assertContains('stub', $ipsum->getWords(2)).

```
74 lines | LoremIpsumBundle/tests/FunctionalTest.php
... lines 1 - 25
    public function testServiceWiringWithConfiguration()
    {
... lines 28 - 34
        $this->assertContains('stub', $ipsum->getWords(2));
    }
... lines 37 - 74
```

Ready to try this? Find your terminal and... run those tests!

```
$ ./vendor/bin/simple-phpunit
```

Ah man! Our new test *fails*! Hmm... it looks like it's *not* using our custom word provider. Weird!

It's probably weirder than you think. Re-run *just* that test by passing --filter testServiceWiringWithConfiguration:

```
$ ./vendor/bin/simple-phpunit --filter testServiceWiringWithConfiguration
```

It still fails. But now, clear the cache directory:

```
$ rm -rf tests/cache
```

And try the test again:

```
$ ./vendor/bin/simple-phpunit --filter testServiceWiringWithConfiguration
```

Holy Houdini Batman! It *passed*! In fact, try *all* the tests:

```
$ ./vendor/bin/simple-phpunit
```

They *all* pass! Black magic! What the heck just happened?

When you boot a kernel, it creates a tests/cache directory that includes the cached container. The *problem* is that it's using the same cache directory for *both* tests. Once the cache directory is populated the first time, *all* future tests re-use the same container from the *first* test, instead of building their own.

It's a subtle problem, but has an easy fix: we need to make the Kernel use a different cache directory each time it's instantiated. There are tons of ways to do this, but here's an easy one. Go back to the Code -> Generate menu, or Command + N on a Mac, and override a method called getCacheDir(). Return __DIR__.'/cache/' then spl_object_hash($this). So, we will *still* use that cache directory, but each time you create a new Kernel, it will use a different subdirectory.

79 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 38
class KnpULoremIpsumTestingKernel extends Kernel
{
... lines 41 - 65
    public function getCacheDir()
    {
        return __DIR__.'/cache/'.spl_object_hash($this);
    }
}
... lines 71 - 79
```

Clear out the cache directory one last time. Then, run the tests!

```
$ ./vendor/bin/simple-phpunit
```

They pass! Run them again:

```
$ ./vendor/bin/simple-phpunit
```

You should now see *four* unique sub-directories inside cache/. I won't do it, but to make things even better, you could clear the cache/ directory between tests with a teardown() method in the test class.

Chapter 13: Adding Routes & Controllers

If you watch a lot of KnpU tutorials, you know that I *love* to talk about how the *whole* point of a bundle is that it adds *services* to the container. But, even I have to admit that a bundle can do a lot more than that: it can add routes, controllers, translations, public assets, validation config and a bunch more!

Find your browser and Google for "Symfony bundle best practices". This is a really nice document that talks about how you're *supposed* to build re-usable bundles. We're following, um, *most* of the recommendations. It tells you the different directories where you should put different things. Some of these directories are just convention, but some are required. For example, if your bundle provides translations, they need to live in the Resources/translations directory next to the bundle class. If you follow that rule, Symfony will automatically load them.

Adding a Route + Controller

Here's our *new* goal: I want to add a route & controller to our bundle. We're going to create an optional API endpoint that returns some delightful lorem ipsum text.

Before we start, I'll open my PhpStorm preferences and, just to make this more fun, search for "Symfony" and enable the Symfony plugin. Also search for "Composer" and select the composer.json file so that PhpStorm knows about our autoload namespaces.

Back to work! In src/, create a Controller directory and inside of that, a new PHP class called IpsumApiController. We don't need to make this extend anything, but it's OK to extend AbstractController to get some shortcuts... except what!? AbstractController doesn't exist!

That's because the class lives in FrameworkBundle and... remember! Our bundle does *not* require that! Ignore this problem for now. Instead, find our app code, open AbstractController, copy its namespace, and use it to add the use statement manually to the controller.

25 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 5
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
... line 7
class IpsumApiController extends AbstractController
... lines 9 - 25
```

Next, add a public function called index. Here, we're going to use the KnpUIpsum class to return a JSON response with some dummy text. When you create a controller in a reusable bundle, the best practice is to register your controller as a proper service and use dependency injection to get anything you need.

25 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 16
    public function index()
    {
... lines 19 - 22
    }
... lines 24 - 25
```

Add public function __construct() and type-hint the first argument with KnpUIpsum. I'll press Alt+Enter and choose Initialize Fields so that PhpStorm creates and sets a property for that.

25 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 9
    private $knpUIpsum;
... line 11
    public function __construct(KnpUIpsum $knpUIpsum)
    {
        $this->knpUIpsum = $knpUIpsum;
    }
... lines 16 - 25
```

Down below, return $this->json() - we will *not* have auto-complete for that method because of the missing AbstractController - with a paragraphs key set to $this->knpUIpsum->getParagraphs() and a sentences key set to $this->knpUIpsum->getSentences()

25 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 16
    public function index()
    {
        return $this->json([
            'paragraphs' => $this->knpUIpsum->getParagraphs(),
            'sentences' => $this->knpUIpsum->getSentences(),
        ]);
    }
... lines 24 - 25
```

Excellent!

Registering your Controller as a Service

Next, we need to register this as a service. In services.xml, copy the first service, call this one ipsum_api_controller, and set its class name. For now, *don't* add public="true" or false: we'll learn more about this in a minute. Pass one argument: the main knpu_lorem_ipsum.knpu_ipsum service.

22 lines | LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 6
    <services>
... lines 8 - 13
        <service id="knpu_lorem_ipsum.controller.ipsum_api_controller" class="KnpU\LoremIpsumBundle\Controller\IpsumApiController"
            <argument type="service" id="knpu_lorem_ipsum.knpu_ipsum" />
        </service>
... lines 17 - 19
    </services>
... lines 21 - 22
```

Perfect!

Routing

Finally, let's add some routing! In Resources/config, create a new routes.xml file. This could be called anything because the user will import this file manually from their app.

To fill this in, as usual, we'll cheat! Google for "Symfony Routing" and, just like we did with services, search for "XML" until you find a good example.

Copy that code and paste it into our file. Let's call the one route knpu_lorem_ipsum_api. For controller, copy the service id, paste, and add a single colon then index.

10 lines | LoremIpsumBundle/src/Resources/config/routes.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing
        http://symfony.com/schema/routing/routing-1.0.xsd">

    <route id="knpu_lorem_ipsum_api" controller="knpu_lorem_ipsum.controller.ipsum_api_controller:index" path="/" >
        <!-- settings -->
    </route>
</routes>
```

Fun fact: in Symfony 4.1, the syntax changes to a double :: and using a single colon is deprecated. Keep a single : for now if you want your bundle to work in Symfony 4.0.

Finally, for path, the user will probably want something like /api/lorem-ipsum. But instead of *guessing* what they want, just set this to /, or at least, something short. We'll allow the user to *choose* the path *prefix*.

And that's it! But... how can we make sure it works? In a few minutes, we're going to write a *legitimate* functional test for this. But, for now, let's just test it in our app!

In the config directory, we have a routes.yaml file, and we *could* import the routes.xml file from here. But, it's more common to go into the routes/ *directory* and create a separate file: knpu_lorem_ipsum.yaml.

Add a root key - _lorem_ipsum - this is meaningless, then resources set to @KnpULoremIpsumBundle and then the path to the file: /Resources/config/routes.xml. *Then*, give this a prefix! How about /api/ipsum.

4 lines | config/routes/knpu_lorem_ipsum.yaml

```
_lorem_ipsum:
    resource: '@KnpULoremIpsumBundle/Resources/config/routes.xml'
    prefix: /api/ipsum
```

Did it work? Let's find out: find your terminal tab for the application, and use the trusty old:

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

There it is! /api/ipsum/. Copy that, find our browser, paste and.... nope. Error!

> Controller ipsum_api_controller cannot be fetched from the container because it is private. Did you forget to tag the service with controller.service_arguments.

The error is not *entirely* correct for *our* circumstance. First, yes, at this time, controllers are the *one* type of service that *must* be public. If you're building an *app*, you can give it this tag, which will automatically make it public. But for a reusable bundle, in services.xml, we need to set public="true".

22 lines | LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 6
    <services>
... lines 8 - 13
        <service id="knpu_lorem_ipsum.controller.ipsum_api_controller" class="KnpU\LoremIpsumBundle\Controller\IpsumApiController"
... line 15
        </service>
... lines 17 - 19
    </services>
... lines 21 - 22
```

Try that again! *Now* it works. And... you *might* be surprised! After all, our bundle references a class that does *not* exist! This *is* a problem... at least, a minor problem. But, because FrameworkBundle *is* included in our app, it *does* work.

But to *really* make things solid, let's add a proper functional test to the bundle that guarantees that this route and controller work. And when we do that, it'll become *profoundly* obvious that we are, yet again, *not* properly requiring all the dependencies we need.

Chapter 14: Controller Functional Test

We just added a route and controller, and since this bundle is going to be used by, probably, *billions* of people, I want to make sure they work! How? By writing a good old-fashioned functional test that surfs to the new URL and checks the result.

In the tests/ directory, create a new Controller directory and a new PHP class inside called IpsumApiControllerTest. As always, make this extend TestCase from PHPUnit, and add a public function testIndex().

18 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php

```
... lines 1 - 10
class IpsumApiControllerTest extends TestCase
{
    public function testIndex()
    {

    }
}
```

How to Boot a Fake App?

The setup for a functional test is pretty similar to an integration test: create a custom test kernel, but this time, import routes.xml inside. Then, we can use Symfony's BrowserKit to make requests into that kernel and check that we get a 200 status code back.

Start by stealing the testing kernel from the FunctionalTest class. Paste this at the bottom, and, just to avoid confusion, give it a different name: KnpULoremIpsumControllerKernel. Re-type the l and hit tab to add the use statement for the Kernel class.

43 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php

```
... lines 1 - 18
class KnpULoremIpsumControllerKernel extends Kernel
{
    public function __construct()
    {
        parent::__construct('test', true);
    }

    public function registerBundles()
    {
        return [
            new KnpULoremIpsumBundle(),
        ];
    }

    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(function(ContainerBuilder $container) {
        });
    }

    public function getCacheDir()
    {
        return __DIR__.'/../cache/'.spl_object_hash($this);
    }
}
```

Then, we can simplify: we don't need any special configuration: just call the parent constructor. Re-type the bundle name and hit tab to get the use statement, and do this on the other two highlighted classes below. Empty the load() callback for now.

Yep, we're just booting a kernel with one bundle... super boring.

Do we Need FrameworkBundle Now?

And here's where things get confusing. In composer.json, as you know, we do *not* have a dependency on symfony/framework-bundle. But now... we have a route and controller... and... well... the *entire* routing and controller system comes from FrameworkBundle! In other words, while not *impossible*, it's incredibly unlikely that someone will want to import our route, but *not* use FrameworkBundle.

This means that we *now* depend on FrameworkBundle. Well actually, that's not *entirely* true. Our new route & controller are optional features. So, in a perfect world, FrameworkBundle should *still* be an *optional* dependency. In other words, we are *not* going to add it to the require key. In reality, if you did, no big deal - but we're doing things the harder, more interesting way.

This leaves us with a big ugly problem! In order to *test* that the route and controller work, we need the route & controller system! We need FrameworkBundle! This is yet *another* case when we need a dependency, but we *only* need the dependency when we're developing the bundle or running tests. Find your terminal and run:

```
$ composer require symfony/framework-bundle --dev
```

Let this download. Excellent!

Importing Routes from the Kernel

Back in the test, thanks to FrameworkBundle, we can use a *really* cool trait to make life simpler. Full disclosure, I helped created the trait - so of course *I* think it's cool. But really, it makes life easier: use MicroKernelTrait. Remove registerContainerConfiguration() and, instead go back again to the Code -> Generate menu - or Command + N on a Mac - and implement the two missing methods: configureContainer(), and configureRoutes().

```
51 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php
... lines 1 - 20
class KnpULoremIpsumControllerKernel extends Kernel
{
    use MicroKernelTrait;
... lines 24 - 36
    protected function configureRoutes(RouteCollectionBuilder $routes)
    {

    }
... line 41
    protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
    {

    }
... lines 46 - 50
}
```

Cool! So... let's import our route! $routes->import(), then the path to that file: __DIR__.'/../../src/Resources/config/routes.xml'.

```
51 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php
... lines 1 - 36
    protected function configureRoutes(RouteCollectionBuilder $routes)
    {
        $routes->import(__DIR__.'/../../src/Resources/config/routes.xml', '/api');
    }
... lines 41 - 51
```

Setting up the Test Client

Nice! And... that's really all the kernel needs. Back up in testIndex(), create the new kernel: new KnpULoremIpsumControllerKernel().

```
57 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php
... lines 1 - 13
class IpsumApiControllerTest extends TestCase
{
    public function testIndex()
    {
        $kernel = new KnpULoremIpsumControllerKernel();
... lines 19 - 23
    }
}
... lines 26 - 57
```

Now, you can almost pretend like this a normal functional test in a normal Symfony app. Create a test client: $client = new Client() - the one from FrameworkBundle - and pass it the $kernel.

Use this to make requests into the app with $client->request(). You will *not* get auto-completion for this method - we'll find out why soon. Make a GET request, and for the URL... actually, down in configureRoutes(), ah, I forgot to add a prefix! Add /api as the second argument. Make the request to /api/.

57 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php

```
... lines 1 - 15
    public function testIndex()
    {
... line 18
        $client = new Client($kernel);
        $client->request('GET', '/api/');
... lines 21 - 23
    }
... lines 25 - 57
```

57 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php

```
... lines 1 - 26
class KnpULoremIpsumControllerKernel extends Kernel
{
... lines 29 - 42
    protected function configureRoutes(RouteCollectionBuilder $routes)
    {
        $routes->import(__DIR__.'/../../src/Resources/config/routes.xml', '/api');
    }
... lines 47 - 56
}
```

Cool! Let's dump the response to see what it looks like: var_dump($client->getResponse()->getContent()). Then add an assert that 200 matches $client->getResponse()->getStatusCode().

57 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php

```
... lines 1 - 13
class IpsumApiControllerTest extends TestCase
{
    public function testIndex()
    {
... lines 18 - 21
        var_dump($client->getResponse()->getContent());
        $this->assertSame(200, $client->getResponse()->getStatusCode());
    }
}
... lines 26 - 57
```

Alright! Let's give this a try! Find your terminal, and run those tests!

```
$ ./vendor/bin/simple-phpunit
```

Woh! They are *not* happy:

> Fatal error class BrowserKit\Client does not exist.

Hmm. This comes from the http-kernel\Client class. Here's what's happening: we use the Client class from FrameworkBundle, *that* extends Client from http-kernel, and *that* tries to use a class from a component called browser-kit, which is an *optional* dependency of http-kernel. Geez.

Basically, we're trying to use a class from a library that we don't have installed. You know the drill, find your terminal and run:

```
$ composer require symfony/browser-kit --dev
```

When that finishes, try the test again!

```
$ ./vendor/bin/simple-phpunit
```

Oof. It just looks *awful*:

> LogicException: Container extension "framework" is not registered.

This comes from ContainerBuilder, which is called from somewhere inside MicroKernelTrait. This is a bit tougher to track down. When we use MicroKernelTrait, behind the scenes, it adds some framework configuration to the container in order to configure the router. But... our kernel does *not* enable FrameworkBundle!

No problem: add new FrameworkBundle to our bundles array.

60 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php

```
... lines 1 - 35
    public function registerBundles()
    {
        return [
... line 39
            new FrameworkBundle(),
        ];
    }
... lines 43 - 60
```

Then, go back and try the tests again: hold your breath:

```
$ ./vendor/bin/simple-phpunit
```

No! Hmm:

> The service url_signer has a dependency on a non-existent parameter "kernel.secret".

This is a fancy way of saying that, for *some* reason, there is a missing parameter. It turns out that FrameworkBundle has *one* required piece of configuration. In your application, open config/packages/framework.yaml. Yep, right on top: the secret key.

This is used in various places for security, and, since it needs to be unique and secret, Symfony can't give you a default value. For our testing kernel, it's meaningless, but it needs to exist. In configureContainer(), add $c->loadFromExtension() passing it framework and an array with secret set to anything. The FrameworkExtension uses this value to set that missing parameter.

60 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php

```
... lines 1 - 48
    protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
    {
        $c->loadFromExtension('framework', [
            'secret' => 'F00',
        ]);
    }
... lines 55 - 60
```

Do those tests... one, last time:

```
$ ./vendor/bin/simple-phpunit
```

Phew! They *pass*! The response status code is 200 and you can even see the JSON. Go back to the test and take out the var_dump().

Next, let's get away from tests and talk about events: the *best* way to allow users to hook into your controller logic.

Chapter 15: Dispatching Custom Events

What if a user wants to change the behavior of our controller? Symfony *does* have a way to override controllers from a bundle... but *not* if that controller is registered as a service, like our controller. Well, ok, thanks to Symfony's incredible container, there is *always* a way to override a service. But let's not make our users do crazy things! If someone wants to tweak how our controller behaves, let's make it easy!

How? By dispatching a custom event. Ready for our new goal? I want to allow a user to *change* the data that we return from our API endpoint. Specifically, we're going to add a *third* key to the JSON array from our app.

Custom Event Class

The *first* step to dispatching an event is to create an event class. Create a new Event directory with a PHP class inside: call it FilterApiResponseEvent. I just made that up.

Make this extend a core Event class from Symfony. When you dispatch an event, you have the opportunity to pass an Event object to any listeners. To be as *awesome* as possible, you'll want to make sure that object contains as *much* useful information as you can.

26 lines | LoremIpsumBundle/src/Event/FilterApiResponseEvent.php

```
... lines 1 - 6
class FilterApiResponseEvent extends Event
{
... lines 9 - 24
}
```

In this case, a listener might want to access the data that we're about to turn into JSON. Cool! Add public function __construct() with an array $data argument. I'll press Alt+Enter and choose "Initialize Fields" to create a data property and set it.

26 lines | LoremIpsumBundle/src/Event/FilterApiResponseEvent.php

```
... lines 1 - 6
class FilterApiResponseEvent extends Event
{
    private $data;

    public function __construct(array $data)
    {
        $this->data = $data;
    }
... lines 15 - 24
}
```

Then, we need a way for the listeners to access this. *And*, we *also* want any listeners to be able to *set* this. Go back to the Code -> Generate menu, or Command + N on a Mac, choose "Getter and Setters" and select data.

```
26 lines | LoremIpsumBundle/src/Event/FilterApiResponseEvent.php
... lines 1 - 15
    public function getData(): array
    {
        return $this->data;
    }

    public function setData(array $data)
    {
        $this->data = $data;
    }
... lines 25 - 26
```

It's ready!

Dispatching the Event

Head to your controller: this is where we'll *dispatch* that event. First, set the data to a $data variable and then create the event object: $event = new FilterApiResponseEvent() passing it the data.

```
35 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php
... lines 1 - 9
class IpsumApiController extends AbstractController
{
... lines 12 - 21
    public function index()
    {
... lines 24 - 28
        $event = new FilterApiResponseEvent($data);
... lines 30 - 32
    }
}
```

I'm not going to dispatch the event *quite* yet, but at the end, pass $event->getData() to the json method.

```
35 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php
... lines 1 - 21
    public function index()
    {
... lines 24 - 31
        return $this->json($event->getData());
    }
... lines 34 - 35
```

To dispatch the event, we need... um... the event dispatcher! And of course, we're going to pass this in as an argument: EventDispatcherInterface $eventDispatcher. Press Alt+enter and select "Initialize Fields" to add that as a property and set it in the constructor.

35 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 13
    private $eventDispatcher;
... line 15
    public function __construct(KnpUIpsum $knpUIpsum, EventDispatcherInterface $eventDispatcher)
    {
... line 18
        $this->eventDispatcher = $eventDispatcher;
    }
... lines 21 - 35
```

As *soon* as we do this, we need to also open services.xml and pass a second argument: type="service" and id="event_dispatcher".

23 lines | LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 6
    <services>
... lines 8 - 13
        <service id="knpu_lorem_ipsum.controller.ipsum_api_controller" class="KnpU\LoremIpsumBundle\Controller\IpsumApiController"
... line 15
            <argument type="service" id="event_dispatcher" />
        </service>
... lines 18 - 20
    </services>
... lines 22 - 23
```

Back in the controller, right after you create the event, dispatch it: $this->eventDispatcher->dispatch(). The first argument is the event *name* and we can actually dream up whatever name we want. Let's use: knpu_lorem_ipsum.filter_api. For the second argument, pass the event.

35 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 9
class IpsumApiController extends AbstractController
{
... lines 12 - 21
    public function index()
    {
... lines 24 - 29
        $this->eventDispatcher->dispatch('knpu_lorem_ipsum.filter_api', $event);
... lines 31 - 32
    }
}
```

And... yea, that's it! I mean, we haven't tested it yet, but this should work: our users have a *new* hook point.

Being Careful with Optional Dependencies

But actually there's a *small* surprise. Find your terminal and re-run all the tests:

```
$ ./vendor/bin/simple-phpunit
```

They fail! Check this out: it says that our controller service has a dependency on a non-existent service event_dispatcher. But, the service id *is* event_dispatcher - that's not a typo! The problem is that the event_dispatcher service - like *many* services - comes from FrameworkBundle.

Open up the test that's failing: FunctionalTest. Inside, we're testing with a kernel that does *not* include FrameworkBundle! We did this on purpose: FrameworkBundle is an *optional* dependency.

Let me say it a different way: one of our services depends on another service that may or may not exist. Since we *want* our bundle to work without FrameworkBundle, we need to make the event_dispatcher service optional. To do that, add an on-invalid attribute set to null.

23 lines | LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 13
        <service id="knpu_lorem_ipsum.controller.ipsum_api_controller" class="KnpU\LoremIpsumBundle\Controller\IpsumApiController"
... line 15
            <argument type="service" id="event_dispatcher" on-invalid="null" />
        </service>
... lines 18 - 23
```

Thanks to this, if the event_dispatcher service doesn't exist, instead of an error, it'll just pass null. That means, we need to make that argument optional, with = null, or by adding a ? before the type-hint.

37 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 9
class IpsumApiController extends AbstractController
{
... lines 12 - 15
    public function __construct(KnpUIpsum $knpUIpsum, EventDispatcherInterface $eventDispatcher = null)
    {
... lines 18 - 19
    }
... lines 21 - 35
}
```

Inside the action, be sure to code defensively: *if* there is an event dispatcher, do our magic.

37 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 21
    public function index()
    {
... lines 24 - 29
        if ($this->eventDispatcher) {
            $this->eventDispatcher->dispatch('knpu_lorem_ipsum.filter_api', $event);
        }
... lines 33 - 34
    }
... lines 36 - 37
```

Try the tests again:

```
$ ./vendor/bin/simple-phpunit
```

Aw yea! Next, let's make our event *easier* to use by documenting it with an event *constants* class. Then... let's make sure it works!

Chapter 16: Event Constants & @Event Docs

There's one way we can make this better, and *all* high quality bundles do this: set the event name as a *constant*, instead of just having this random string. It's even a bit cooler than it sounds.

In the Event directory, create a new class: KnpULoremIpsumEvents. *If* your bundle dispatches events, you should typically have *one* class that has a constant for *each* event. It's a one-stop place to find *all* the event hook points.

16 lines | LoremIpsumBundle/src/Event/KnpULoremIpsumEvents.php

```
... lines 1 - 4
final class KnpULoremIpsumEvents
{
... lines 7 - 14
}
```

Make this class final... which isn't too important... but in general, you should considering making *any* class in a shareable library final, unless you *do* want people to be able to sub-class it. Using final is always a safe bet and can be removed later.

Anyways, add const FILTER_API = '', go copy the event name and paste it here.

16 lines | LoremIpsumBundle/src/Event/KnpULoremIpsumEvents.php

```
... lines 1 - 13
    const FILTER_API = 'knpu_lorem_ipsum.filter_api';
... lines 15 - 16
```

Now, of course, *replace* that string in the controller with KnpULoremIpsumEvents::FILTER_API.

38 lines | LoremIpsumBundle/src/Controller/IpsumApiController.php

```
... lines 1 - 10
class IpsumApiController extends AbstractController
{
... lines 13 - 22
    public function index()
    {
... lines 25 - 30
        if ($this->eventDispatcher) {
            $this->eventDispatcher->dispatch(KnpULoremIpsumEvents::FILTER_API, $event);
        }
... lines 34 - 35
    }
}
```

So, this is nice! Though, the reason I *really* like this is that it gives us a proper place to document the *purpose* of this event: why you would listen to it and the types of things you can do.

The Special @Event Documentation

But the *coolest* part is this: add @Event(), and then inside double quotes, put the full class name of the event that listeners will receive. In other words, copy the namespace from the event class, paste it here and add \FilterApiResponseEvent.

```
16 lines | LoremIpsumBundle/src/Event/KnpULoremIpsumEvents.php
... lines 1 - 4
final class KnpULoremIpsumEvents
{
    /**
     * Called directly before the Lorem Ipsum API data is returned.
     *
     * Listeners have the opportunity to change that data.
     *
     * @Event("KnpU\LoremIpsumBundle\Event\FilterApiResponseEvent")
     */
    const FILTER_API = 'knpu_lorem_ipsum.filter_api';
}
```

What the heck does this do? On a technical level, absolutely nothing! This is purely documentation. But! Some systems - like PhpStorm - know to *parse* this and use it to help us when we're building event subscribers. We'll see *exactly* what I'm talking about in a minute. But, it's at least good documentation: if you listen to this event, this is the event object you should expect.

Creating an EventSubscriber

And... we're done! I'm not going to write a test for this, but I *do* at least want to make sure it works in my project. Move back over to the application code. Inside src/, create a new directory called EventSubscriber. Then, a new class called AddMessageToIpsumApiSubscriber.

```
25 lines | src/EventSubscriber/AddMessageToIpsumApiSubscriber.php
... lines 1 - 8
class AddMessageToIpsumApiSubscriber implements EventSubscriberInterface
{
... lines 11 - 23
}
```

Like all subscribers, this needs to implement EventSubscriberInterface. Then I'll go to the Code -> Generate menu, or Command + N on a Mac, select Implement Methods, and add getSubscribedEvents.

```
25 lines | src/EventSubscriber/AddMessageToIpsumApiSubscriber.php
... lines 1 - 10
    public static function getSubscribedEvents()
    {
... lines 13 - 15
    }
... lines 17 - 25
```

Before we fill this in, I want to make sure that PhpStorm is fully synchronized with how our bundle looks - sometimes the symlink gets stale. Right click on the vendor/knpuniversity/lorem-ipsum-bundle directory, and click "Synchronize".

Cool: now it will *definitely* see the new event classes. When it's done indexing, return an array with KnpULoremIpsumEvents::FILTER_API set to, how about, onFilterApi.

```
25 lines | src/EventSubscriber/AddMessageToIpsumApiSubscriber.php
... lines 1 - 10
    public static function getSubscribedEvents()
    {
        return [
            KnpULoremIpsumEvents::FILTER_API => 'onFilterApi',
        ];
    }
... lines 17 - 25
```

Ready for the magic? Thanks to the Symfony plugin, we can hover over the method name, press Alt + Enter and select "Create Method". Woh! It added the onFilterApi method for me *and* type-hinted the first argument with FilterApiResponseEvent! But, how did it know that this was the right event class?

25 lines | src/EventSubscriber/AddMessageToIpsumApiSubscriber.php

```
... lines 1 - 17
    public function onFilterApi(FilterApiResponseEvent $event)
    {
... lines 20 - 22
    }
... lines 24 - 25
```

It knew that thanks to the @Event() documentation we added earlier.

Inside the method, let's say $data = $event->getData() and then add a new key called message set to, the very important, "Have a magical day". Finally, set that data *back* on the event with $event->setData($data).

25 lines | src/EventSubscriber/AddMessageToIpsumApiSubscriber.php

```
... lines 1 - 17
    public function onFilterApi(FilterApiResponseEvent $event)
    {
        $data = $event->getData();
        $data['message'] = 'Have a magical day!';
        $event->setData($data);
    }
... lines 24 - 25
```

That is it! Thanks to Symfony's service auto-configuration, this is already a service and it will already be an event subscriber. In other words, go refresh the API endpoint. It, just, works! Our controller is now extensible, without the user needing to override it. Dispatching events is most commonly done in controllers, but you could dispatch them in any service.

Next, let's improve our word provider setup by making it a true *plugin* system with dependency injection tags and compiler passes. Woh.

Chapter 17: Plugin System with Tags

At this point, the user *can* control the word provider. But, there's only ever *one* word provider. That may be fine, but I want to make this more flexible! And, along the way, learn about one of the most important, but complex systems that is commonly used in bundles: the tag & compiler pass system.

First, let's make our mission clear: instead of allowing just *one* word provider, I want to allow *many* word providers. I also want *other* bundles to be able to automatically add new word providers to the system. Basically, I want a word provider *plugin* system.

Allowing Multiple Word Providers

To get this started, we need to refactor KnpUIpsum: change the first argument to be an *array* of $wordProviders. Rename the property to $wordProviders, and I'll add some PHPDoc above this to help with auto-completion: this will be an array of WordProviderInterface[].

229 lines | LoremIpsumBundle/src/KnpUIpsum.php

```
... lines 1 - 9
class KnpUIpsum
{
    /**
     * @var WordProviderInterface[]
     */
    private $wordProviders;
... lines 16 - 22
    public function __construct(array $wordProviders, bool $unicornsAreReal = true, $minSunshine = 3)
    {
        $this->wordProviders = $wordProviders;
... lines 26 - 27
    }
... lines 29 - 227
}
```

Let's also add a *new* property called wordList: in a moment, we'll use this to store the final word list, so that we only need to calculate it once.

229 lines | LoremIpsumBundle/src/KnpUIpsum.php

```
... lines 1 - 20
    private $wordList;
... lines 22 - 229
```

The big change is down below in the getWordList() method. First, if null === $this->wordList, then we need to loop over all the word providers to *create* that word list.

Once we've done, that, at the bottom, return $this->wordList.

229 lines | LoremIpsumBundle/src/KnpUIpsum.php

```
... lines 1 - 210
    private function getWordList(): array
    {
        if (null === $this->wordList) {
... lines 214 - 223
        }

        return $this->wordList;
    }
... lines 228 - 229
```

Back in the if, create an empty $words array, then loop over $this->wordProviders as $wordProvider. For each word provider, set $words to an array_merge of the words so far and $wordProvider->getWordList().

229 lines | LoremIpsumBundle/src/KnpUIpsum.php

```
... lines 1 - 212
        if (null === $this->wordList) {
            $words = [];
            foreach ($this->wordProviders as $wordProvider) {
                $words = array_merge($words, $wordProvider->getWordList());
            }
... lines 218 - 223
        }
... lines 225 - 229
```

After, we need a sanity check: if the count($words) <= 1, throw an exception: this class only works when there are at least *two* words. Finally, set $this->wordList to $words.

229 lines | LoremIpsumBundle/src/KnpUIpsum.php

```
... lines 1 - 212
        if (null === $this->wordList) {
... lines 214 - 218
            if (count($words) <= 1) {
                throw new \Exception('Word list must contain at least 2 words, yo!');
            }
... line 222
            $this->wordList = $words;
        }
... lines 225 - 229
```

Perfect! This class is now just a *little* bit more flexible. In config/services.xml, instead of passing one word provider, add an <argument with type="collection", them move the word provider argument inside of this.

25 lines | LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 6
    <services>
        <service id="knpu_lorem_ipsum.knpu_ipsum" class="KnpU\LoremIpsumBundle\KnpUIpsum" public="true">
            <argument type="collection">
                <argument type="service" id="knpu_lorem_ipsum.word_provider" />
            </argument>
        </service>
... lines 13 - 22
    </services>
... lines 24 - 25
```

There's no fancy plugin system yet, but things *should* still work. Find your browser and refresh. Great! Even the article page looks fine.

Tagging the Service

Here's the burning question: how can we improve this system so that our application, or even *other* bundles, can add new word providers to this collection? The answer... takes a few steps to explain.

First, I want you to pass an *empty* collection as the first argument. Then, below on the word provider service, change this to use the longer service syntax so that, inside, we can add <tag name="">, and, invent a new tag string. How about: knpu_ipsum_word_provider.

25 lines | LoremIpsumBundle/src/Resources/config/services.xml

```
... lines 1 - 7
        <service id="knpu_lorem_ipsum.knpu_ipsum" class="KnpU\LoremIpsumBundle\KnpUIpsum" public="true">
            <argument type="collection" /> <!-- filled in via a compiler pass -->
        </service>
... line 11
        <service id="knpu_lorem_ipsum.knpu_word_provider" class="KnpU\LoremIpsumBundle\KnpUWordProvider">
            <tag name="knpu_ipsum_word_provider" />
        </service>
... lines 15 - 25
```

If this makes *no* sense to you, no problem. Because, it will *not* work yet: when you refresh, big error! At this moment, there are *zero* word providers.

If you've worked with Symfony for a while, you've probably *used* tags before. At a high-level, the idea is pretty simple. First, you can attach tags to services... which... initially... does nothing. But then, a bundle author - that's us! - can write some code that finds all services in the container with this tag and dynamically add them to the collection argument!

When this is setup, our application - or even *other bundles* - can add services, give them this tag, and they will automatically be "plugged" into the system. This is how Twig Extensions, Event Subscribers, Voters, and many other parts of Symfony work.

The Easy Way

So... how do we hook this all up? Well, if your bundle will only need to support Symfony 3.4 or higher, there's a *super* easy way. Just replace the <argument type="collection"> with <argument type="tagged" tag="knpu_ipsum_word_provider" />. This tells Symfony to find all services with this tag, and pass them as a collection. And... you'd be done!

Tip

You will also need to change the array $wordProviders constructor argument in KnpUIpsum to iterable $wordProviders.

But, if you want to support *earlier* versions of Symfony, or you want to know how the compiler pass system works, keep watching.

Chapter 18: Tags, Compiler Passes & Other Nerdery

Let's review: we gave our service a tag. And now, we want to tell Symfony to find *all* services in the container with this tag, and pass them as the first argument to our KnpUIpsum service. Like I mentioned in the previous chapter, if you only need to support Symfony 3.4 or higher, there's a shortcut. But if you need to support lower versions or want to geek out with me about compiler passes, well, you're in luck!

First question: how can we find *all* services that have the knpu_ipsum_word_provider tag? If you look in the extension class, you might think that we could do some magic here with the $container variable. And... yea! It even has a method called findTaggedServiceIds()!

But... you actually *can't* do this logic here. Why? Well, when this method is called, not *all* of the other bundles and extensions have been loaded yet. So if you tried to find all the services with a certain tag, some of the services might not be in the container yet. And actually, you can't even get *that* far: the ContainerBuilder is *empty* at the beginning of this method: it doesn't contain *any* of the services from *any* other bundles. Symfony passes us an empty container builder, and then merges it into the *real* one later.

Compiler Pass

The *correct* place for any logic that needs to operate on the *entire* container, is a compiler pass. In the DependencyInjection directory - though it doesn't technically matter where this class goes - create a Compiler directory then a new class called WordProviderCompilerPass. Make this, implement a CompilerPassInterface, and then go to the Code -> Generate menu - or Command + N on a Mac - click "Implement Methods" and select process().

```
14 lines | LoremIpsumBundle/src/DependencyInjection/Compiler/WordProviderCompilerPass.php
... lines 1 - 7
class WordProviderCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {

    }
}
```

A compiler pass *also* receives a ContainerBuilder argument. But, instead of being empty, this is full of *all* of the services from *all* of the bundles. That means that we can say foreach ($container->findTaggedServiceIds(), pass this the tag we're using: knpu_ipsum_word_provider, and say as $id => $tags.

```
17 lines | LoremIpsumBundle/src/DependencyInjection/Compiler/WordProviderCompilerPass.php
... lines 1 - 9
    public function process(ContainerBuilder $container)
    {
        foreach ($container->findTaggedServiceIds('knpu_ipsum_word_provider') as $id => $tags) {
... line 13
        }
... line 15
    }
```

This is a little confusing: the $id key is the service ID that was tagged. Then, $tags is an array with extra information about the tag. Sometimes, a tag can have other attributes, like priority. You can also tag the same service with the same *tag*, multiple times.

Anyways, we don't need that info: let's just var_dump($id) to see if it works, then die.

```
17 lines | LoremIpsumBundle/src/DependencyInjection/Compiler/WordProviderCompilerPass.php
... lines 1 - 11
        foreach ($container->findTaggedServiceIds('knpu_ipsum_word_provider') as $id => $tags) {
            var_dump($id);
        }
        die;
... lines 16 - 17
```

Registering the Compiler Pass

To *tell* Symfony about the compiler pass, open your *bundle* class. Here, go back to the Code -> Generate menu - or Command + N on a Mac - choose "Override Methods" and select build(). You don't need to call the parent build() method: it's empty. *All* we need here is $container->addCompilerPass(new WordProviderCompilerPass()).

```
29 lines | LoremIpsumBundle/src/KnpULoremIpsumBundle.php
... lines 1 - 9
class KnpULoremIpsumBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        $container->addCompilerPass(new WordProviderCompilerPass());
    }
... lines 16 - 27
}
```

There are different *types* of compiler passes, which determine when they are executed relative to *other* passes. And, there's also a priority. But unless you're doing something *really* fancy, the standard type and priority work fine.

Thanks to this line, whenever the container is built, it *should* hit our die statement. Let's move over to the browser and, refresh!

Yes! There is the *one* service that has the tag.

And now... it's easy! The code in a compiler pass looks a lot like the code in an extension class. At the top, add $definition = $container->getDefinition('knpu_lorem_ipsum.knpu_ipsum').

Ultimately, we need to modify *this* services's first argument. Create an empty $references array. And, in the foreach, just add stuff to it: $references[] = new Reference() and pass in the $id.

Finish this with $definition->setArgument(), pass it 0 for the first argument, and the array of reference objects.

```
21 lines | LoremIpsumBundle/src/DependencyInjection/Compiler/WordProviderCompilerPass.php
... lines 1 - 10
    public function process(ContainerBuilder $container)
    {
        $definition = $container->getDefinition('knpu_lorem_ipsum.knpu_ipsum');
        $references = [];
        foreach ($container->findTaggedServiceIds('knpu_ipsum_word_provider') as $id => $tags) {
            $references[] = new Reference($id);
        }

        $definition->setArgument(0, $references);
    }
```

We're done! Go back to our browser and try it! Woohoo! We're now passing an *array* of all of the word provider services into the KnpUIpsum class.... which... yea, is just one right now.

Cleanup the Old Configuration

With this in place, we can remove our old config option. In the Configuration class, delete the word_provider option. And in the extension class, remove the code that reads this.

31 lines | LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 10
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
        $loader->load('services.xml');

        $configuration = $this->getConfiguration($configs, $container);
        $config = $this->processConfiguration($configuration, $configs);

        $definition = $container->getDefinition('knpu_lorem_ipsum.knpu_ipsum');
        $definition->setArgument(1, $config['unicorns_are_real']);
        $definition->setArgument(2, $config['min_sunshine']);
    }
... lines 25 - 29
}
```

Tagging the CustomWordProvider

Next, move over to the application code, and in config/packages/knpu_lorem_ipsum.yaml, yep, take out the word_provider key.

3 lines | config/packages/knpu_lorem_ipsum.yaml

```
knpu_lorem_ipsum:
    min_sunshine: 5
```

If you refresh now... it's going to work. But, not surprisingly, the word "beach" will not appear in the text. Remember: "beach" is the word that we're adding with our CustomWordProvider. This class is *not* being used. And... that make sense! We haven't tagged this service with *anything*, so our bundle doesn't know to use it.

Before we do that, now that there are *multiple* providers, I don't need to extend the core provider anymore. Implement the WordProviderInterface directly. Then, just return an array with the one word: beach.

14 lines | src/Service/CustomWordProvider.php

```
... lines 1 - 6
class CustomWordProvider implements WordProviderInterface
{
    public function getWordList(): array
    {
        return ['beach'];
    }
}
```

To tag the service, open config/services.yaml. This class is automatically registered as a service. But to give it a tag, we need to override that: App\Service\CustomWordProvider, and, below, tags: [knpu_ipsum_word_provider].

40 lines | config/services.yaml

```
... lines 1 - 5
services:
... lines 7 - 37
    App\Service\CustomWordProvider:
        tags: ['knpu_ipsum_word_provider']
```

Let's try it! Refresh! Yes! It's alive!

Setting up Autoconfiguration

But... there's something that's bothering me. *Most* of the time in Symfony, you *don't* need to manually configure the tag. For example, earlier, when we created an event subscriber, we did *not* need to give it the kernel.event_subscriber tag. Instead, Symfony was smart enough to see that our class implemented EventSubscriberInterface, and so it added that tag for us *automatically*.

So... what's the difference? Why can't the tag be automatically added in *this* situation? Well... it can! But we need to set this up in our bundle. Open the extension class, go anywhere in the load() method, and add $container->registerForAutoconfiguration(WordProviderInterface::class). The feature that automatically adds tags is called autoconfiguration, and this method returns a "template" Definition object that we can modify. Use ->addTag('knpu_ipsum_word_provider').

35 lines | LoremIpsumBundle/src/DependencyInjection/KnpULoremIpsumExtension.php

```
... lines 1 - 11
class KnpULoremIpsumExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
... lines 16 - 25
        $container->registerForAutoconfiguration(WordProviderInterface::class)
            ->addTag('knpu_ipsum_word_provider');
    }
... lines 29 - 33
}
```

Cool, right? Back in our app code, remove the service entirely. And now, try it! Hmm, no beach the first time but on the second refresh... we got it!

We now have a *true* word provider plugin system. *And* creating a custom word provider is as easy as creating a class that implements WordProviderInterface.

Next, let's finally put our library up on Packagist!

Chapter 19: Publishing to Packagist

Our bundle is ready to be shared with the world! So let's take care of a few last details, and publish our bundle to Packagist!

Choosing a License

But, before we publish this *anywhere*, we need do some boring, but very important legal work. Go to choosealicense.com and find the license that works best for you. Symfony is licensed MIT, and that's *definitely* the best practice. Whatever you choose, copy the license, find your bundle code, and at the root, create the LICENSE file.

21 lines | LoremIpsumBundle/LICENSE

```
MIT License

Copyright (c) [year] [fullname]

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Pushing to GitHub

Legal stuff, done! Next, find your terminal: there are a bunch of uncommitted changes. Oh, before we add them, I made a mistake!

I have an extra tests/Controller/cache directory! Open IpsumApiControllerTest and find the getCacheDir() method. I *meant* to change this to use the same cache directory as FunctionalTest, which is already set to be ignored by git. Add a ../ to the path. Then, delete the extra cache/ dir. There's also an extra logs directory, but it's empty, so just ignore it.

60 lines | LoremIpsumBundle/tests/Controller/IpsumApiControllerTest.php

```
... lines 1 - 26
class KnpULoremIpsumControllerKernel extends Kernel
{
... lines 29 - 55
    public function getCacheDir()
    {
        return __DIR__.'/../cache/'.spl_object_hash($this);
    }
}
```

Now move back to your terminal, add everything to git, give it an inspiring message, and commit!

With everything committed, let's push this to GitHub! Well, you can host it *anywhere*, but GitHub is the most common place. I'll click "New Repository", choose the KnpUniversity organization, and name it lorem-ipsum-bundle.

It's not *required*, but it's usually nice to name the repository the same as the package name in composer.json. Give it a clever description, make sure it's public, and create repository!

Copy the code to push to an existing repository, go find your terminal, quick! Paste, hit enter, wait impatiently... then... say hello to our new repository!

Registering on Packagist

With that done, we can *now* put our bundle up on Packagist! Go to packagist.org and make sure you're logged in. Then, it's *super* easy: click "Submit", copy the GitHub URL, paste, and click "Check".

This does some sanity checks in the background, like parsing your composer.json file and waiting for Jordi to search for any similar packages on Packagist, to help avoid duplication.

Looks ok! Moment of truth: Submit!

Boom! We are a package!

Auto-updating with the GitHub Service Hook

Oh, but notice this message:

> The package is not auto-updated. Please setup the Github Service Hook

This is actually important. When we create a new tag in GitHub, we want Packagist to automatically see it.

Go back to GitHub, click Settings, Integration & services, "Add service" and find Packagist. You'll need to enter your username and a token you can find on your Packagist profile page. Then, add service!

Requiring the new Package

And, for now, we're done! We have a *real* package! Next, open our application's composer.json file. We're still using this path repository option. Let's *finally* install our package *properly*. Remove the repositories section.

Then, go to the terminal for your app, and, first, *remove* the current package:

```
$ composer remove knpuniversity/lorem-ipsum-bundle
```

Gone! And thanks to the Flex recipe, it also removed the bundle from bundles.php. Cool!

Now, lets re-install it:

```
$ composer require knpuniversity/lorem-ipsum-bundle
```

This downloads dev-master, so the master branch, because there's no tag yet. *And*! Flex re-added the bundle to bundles.php.

Writing a Decent README

Cool! But, go back to the GitHub page for our bundle. See anything missing? Yea, no README! That's not ok! If you go back to the "Symfony bundle best practices" page, this has an example README you can use to get started.

Head back to our code, I'll close a few files, then create a new README.md file. And, bam! I just wrote us a README file!

94 lines | LoremIpsumBundle/README.md

~~~
# Hello LoremIpsumBundle!

LoremIpsumBundle is a way for you to generate "fake text" into
your Symfony application, but with *just* a little bit more joy
than your normal lorem ipsum.

Install the package with:

```console
composer require knpuniversity/lorem-ipsum-bundle --dev
```

And... that's it! If you're *not* using Symfony Flex, you'll also
need to enable the `KnpU\LoremIpsumBundle\KnpULoremIpsumBundle`
in your `AppKernel.php` file.

## Usage

This bundle provides a single service for generating fake text, which
you can autowire by using the `KnpUIpsum` type-hint:
... lines 21 - 94
~~~

Don't worry, I'm not going to lecture you on how to write README files. Well, actually, can I take just *one* minute to point out the *most* important parts that I think people sometimes forget?

To start, make sure your bundle has these four parts. One, at the top, say what the bundle does in plain language! Two, show the composer require installation command. Three, give a simple usage example, before talking about any other technical jargon. And four, show the configuration.

After that, you can talk about whatever complex or theoretical stuff you want, like how to create a word provider.

Also, when you create code examples, there are *two* common mistakes. First, make sure you include the file path as a comment: people don't always know where a file should live. Second, *don't* create the code blocks here. Believe me, you'll make a mistake. Code them in a *real* app, paste them here, then tweak.

Oh, and for the configuration section, remember, you can run:

```
$ php bin/console config:dump knpu_lorem_ipsum
```

to get a *full* config tree to paste here. Oh, and, if the user needs to *create* a file - like knpu_lorem_ipsum.yaml, say that explicitly: sometimes people think they're doing something wrong if a file doesn't already exist.

A Recipe?

The *last* thing I would recommend is, if it makes sense, create a recipe for your bundle. Do this at github.com/symfony/recipes-contrib. We're not going to do this, but if your bundle needs a config file or *any* other setup, this is a *huge* way to make it easier to use.

If you *don't* create a recipe, Flex will at least enable the bundle automatically. And in a lot of cases - like for this bundle - that's enough.

Ok, just *one* topic left, and it's fun! Let's setup continuous integration on Travis CI so that we can be sure our tests are always passing.

Chapter 20: CI with Travis CI

Our bundle is missing only two things: it needs a stable release and it needs continuous integration.

Before we automate our tests, we should probably make sure they still pass:

```
$ ./vendor/bin/simple-phpunit
```

Bah! Boo Ryan: I let our tests get a bit out-of-date. The first failure is in FunctionalTest.php in testServiceWiringWithConfiguration().

Of course: we're testing the word_provider option, but that doesn't even exist anymore! We *could* update this test for the tag system, but it's a little tricky due to the randomness of the classes. To keep us moving, just delete the test. Also delete the configuration we added in the kernel, and the loadFromExtension() call. But, just for the heck of it, I'll keep the custom word provider and tag it to integrate our stub word list.

62 lines | LoremIpsumBundle/tests/FunctionalTest.php

```
... lines 1 - 26
class KnpULoremIpsumTestingKernel extends Kernel
{
... lines 29 - 40
    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(function(ContainerBuilder $container) {
            $container->register('stub_word_list', StubWordList::class)
                ->addTag('knpu_ipsum_word_provider');
        });
    }
... lines 48 - 52
}
... lines 54 - 62
```

The *second* failure is in KnpUIpsumTest. Ah yea, the first argument to KnpUIpsum is now an *array*. Wrap the argument in square brackets, then fix it in all three places.

67 lines | LoremIpsumBundle/tests/KnpUIpsumTest.php

```
... lines 1 - 8
class KnpUIpsumTest extends TestCase
{
    public function testGetWords()
    {
        $ipsum = new KnpUIpsum([new KnpUWordProvider()]);
... lines 14 - 23
    }
... line 25
    public function testGetSentences()
    {
        $ipsum = new KnpUIpsum([new KnpUWordProvider()]);
... lines 29 - 37
    }
... line 39
    public function testGetParagraphs()
    {
... lines 42 - 43
        for ($i = 0; $i < 100; $i++) {
            $ipsum = new KnpUIpsum([new KnpUWordProvider()]);
... lines 46 - 64
        }
    }
}
```

Ok, try the tests again!

```
$ ./vendor/bin/simple-phpunit
```

Yes! They pass.

Adding the .travis.yml File

The *standard* for continuous integration of open source libraries is definitely Travis CI. And if you go back to the "Best Practices" docs for bundles, near the top, Symfony has an *example* of a robust Travis configuration file! Awesome!

Copy this *entire* thing, go back to the bundle, and, at the root, create a new file - .travis.yml. Paste!

59 lines | LoremIpsumBundle/.travis.yml

```
language: php
sudo: false
cache:
  directories:
    - $HOME/.composer/cache/files
    - $HOME/symfony-bridge/.phpunit

env:
  global:
    - PHPUNIT_FLAGS="-v"
    - SYMFONY_PHPUNIT_DIR="$HOME/symfony-bridge/.phpunit"

matrix:
  fast_finish: true
  include:
... lines 16 - 59
```

We'll talk about some of the specifics of this file in a minute. But first, in your terminal, add everything we've been working on, commit, and push.

Activating Travis CI

With the Travis config file in place, the next step is to activate CI for the repo. Go to travis-ci.org and make sure you're signed in with GitHub. Click the "+" to add a new repository, I'll select the "KnpUniversity" organization and search for lorem.

Huh. Not found. Because it's a new repository, it probably doesn't see it yet. Click the "Sync Account" button to fix that. And... search again. There it is! If it's *still* not there for you, keep trying "Sync Account": sometimes, it takes several tries.

Activate the repo, then click to view it. To trigger the first build, under "More options", click, ah, "Trigger build"! You don't need to fill in any info on the modal.

Oh, and from now on, a new build will happen automatically whenever you push. We only need to trigger the *first* build manually. And... go go go!

Adjusting PHP & Symfony Version Support

While this is working, let's go look at the .travis.yml file. It's... well... *super* robust: it tests the library on multiple PHP version, uses special flags to test with the *lowest* version of your library's dependencies and even tests against multiple versions of Symfony. Honestly, it's a bit ugly, but the result is impressive.

Back on Travis CI, uh oh, we're starting to see failures! No! Let's click on one of them. Interesting... it's some PHP version issue! Remember, we decided to support only PHP 7.1.3 or higher. But... we're testing the bundle against PHP 7.0! We *could* allow PHP 7.0... but let's stay with 7.1.3. In the Travis matrix, delete the 7.0 test, and change the --prefer-lowest to use 7.1.

58 lines | LoremIpsumBundle/.travis.yml

```
... lines 1 - 12
matrix:
... line 14
  include:
... lines 16 - 18
    - php: 7.1
... lines 20 - 21
      # Test the latest stable release
    - php: 7.1
    - php: 7.2
... lines 25 - 58
```

Go back to the main Travis page again. Hmm: two failures at the bottom deal with something called symfony/lts. These make sure that Symfony works with the LTS - long-term support version - of Symfony 2 - so Symfony 2.8 - as well as the LTS of version 3 - so Symfony 3.4. Click into the LTS version 3 build. Ah, it can't install the packages: symfony/lts v3 conflicts with symfony/http-kernel version 4.

The test is trying to install version *3* of our Symfony dependencies, but that doesn't work, because *our* bundle requries everything at version 4!

And... that's *maybe* ok! If we *only* want to support Symfony 4, we can just delete that test. But I think we should *at least* support Symfony 3.4 - the latest LTS.

To do that, in composer.json, change the version to ^3.4 || ^4.0. Use this for *all* of our Symfony libraries.

```
34 lines | LoremIpsumBundle/composer.json
    ... lines 1 - 11
    "require": {
    ... line 13
        "symfony/config": "^3.4 || ^4.0",
        "symfony/dependency-injection": "^3.4 || ^4.0",
        "symfony/http-kernel": "^3.4 || ^4.0"
    },
    "require-dev": {
        "symfony/framework-bundle": "^3.4 || ^4.0",
        "symfony/phpunit-bridge": "^3.4 || ^4.0",
        "symfony/browser-kit": "^3.4 || ^4.0"
    },
    ... lines 23 - 34
```

The cool thing is, we don't *actually* know whether or not our bundle *works* with Symfony 3.4. But... we don't care! The tests will tell us if there are any problems.

Also, in .travis.yml, remove the lts v2 test.

Ok, find your terminal, add, commit with a message, and... push!

This should immediately trigger a build. Click "Current"... there it is!

Let's fast-forward... they're starting to pass... and... cool! The first 5 pass! The last one is still running and, actually, that's going to fail! But don't worry about it: this is testing our bundle agains the latest, unreleased version of Symfony, so we don't care too much if it fails. But, I'll show you why it's failing in a minute.

Tagging Version 1.0

Now that our tests are passing - woo! - it's time to tag our first, official release. You can do this from the command line, but I kinda like the GitHub interface. Set the version to v1.0.0, give it a title, and describe the release. This is where I'd normally include more details about new features or bugs we fixed. Then, publish!

You can also do pre-releases, which is a good idea if you don't want to create a stable version 1.0.0 immediately. On Packagist, the release *should* show up here automatically. But, I'm impatient, so click "Update" and... yes! There's our version 1.0.0!

Oh, before I forget, back on Travis, go to "Build History", click master and, as promised, the last one failed. I just want to show you *why*: it failed because of a deprecation warning:

> Referencing controllers with a single colon is deprecated in Symfony 4.1.

Starting in Symfony 4.1, you should refer to your controllers with *two* colons in your route. To stay compatible with 4.0, we'll leave it.

Installing the Stable Release

Now that we *finally* have a stable release, let's install it in our app. At your terminal, first remove the bundle:

```
$ composer remove knpuniversity/lorem-ipsum-bundle
```

Wait.... then re-add it:

```
$ composer require knpuniversity/lorem-ipsum-bundle
```

Yes! It got v1.0.0.

We have an awesome bundle! It's tested, it's extendable, it's on GitHub, it has continuous integration, it can bake you a cake and it has a stable release.

I hope you learned a ton about creating re-usable bundles... and even more about how Symfony works in general. As always, if you have any questions or comments, talk to us down in the comments section.

All right guys, seeya next time.