Daniel Sipos

Drupal 8 Module Development

Build and customize Drupal 8 modules and extensions efficiently

Packt

Contents

Chapter 1. Developing for Drupal 8

Drupal is a web-based **Content Management System** (**CMS**). While it is useful out of the box, it is designed with developers in mind. The purpose of this book is to explain how Drupal can be extended in many ways and for many purposes. To this end, the version we will use will be the latest one at the time of writing this book--Drupal 8.2. In this book, we will cover a wide range of development topics. We'll discuss how to create a Drupal 8 module, and as we go through the chapters, many concepts and tips that will help you build what you need will be introduced. The goal is not only to explain how things work but also to go through some examples to demonstrate them. Since no book can contain everything, I hope that after reading this book, you'll be able to expand on this knowledge on your own using the resources I reference and by looking into the Drupal code yourself. As helpful as such a book can be for learning any kind of software development, if you really want to progress, you will need to apply the knowledge you learned and explore the source code yourself. Only by doing this you will be able to understand complex systems with many dependencies and layers.

This chapter introduces the terminology, tools, and processes for developing Drupal 8. While subsequent chapters focus on code, this chapter focuses on concepts. We'll talk about the architecture of Drupal and how you can hook into Drupal at strategic places to extend it for accomplishing new tasks.

The following are the major topics we will be covering in this chapter:

- An introduction to Drupal development
- Drupal 8 architecture
- The major subsystems
- Tools for developing in Drupal

By the end of this chapter, you will understand the architectural aspects of Drupal and be ready to start writing code.

Introducing Drupal (for developers)

Out of the box, Drupal performs all of the standard functions of a web-based content management system:

- Visitors can view published information on the site; navigate through menus, view listings, and individual pages; and so on
- Users can create accounts and leave comments
- Administrators can manage the site configuration and control the permissions levels of users
- Editors can create, preview, and then publish content when it is ready
- Content can be syndicated to RSS, where feed readers can pick up new articles as they are published
- With several built-in themes, even the look and feel of the site can be changed easily

With Drupal 8, the scope of what a site builder can do has greatly increased. Core multilingual capabilities make it much easier to configure the site to use multiple languages, creating content listings a few clicks away out of the box, and content management, in general, has greatly improved.

As fantastic as these features are, they will certainly not satisfy the needs of all users. To that end, Drupal's capabilities can be easily extended with modules, themes, and installation profiles. Take a look at Drupal's main website, (http://drupal.org), and you will find thousands of modules that provide new features and thousands of themes that transform the look and feel of the site.

The fact that almost all aspects of Drupal's behavior can be intercepted and transformed through the module and theme mechanisms has led many to claim that Drupal isn't just a CMS, but a **Content Management Framework** (**CMF**) capable of being re-tooled to specific needs and functional requirements. This is particularly the case with Drupal 8--the latest version of Drupal and the focus of this book--as great progress has been made on the extensibility front too.

Establishing whether Drupal is rightly called a CMS or CMF is beyond our purpose here, but it is certain that**Comma-Separated Version** (**CSV**) files? There are several modules for that (depending on what data you want to export). Interested in Facebook support, integration...

Summary

This chapter has been an overview of Drupal 8 for developers. We saw what technologies Drupal uses. We took a look at Drupal's architecture. We took a cursory glance at several prominent subsystems of Drupal. We also got a feel of which developer-oriented tools are to be used while working with Drupal.

Starting with the next chapter, we will be working with code. In fact, each of the subsequent chapters will focus on practical aspects of working with Drupal.

In the next chapter, we will create our first Drupal 8 module with the obligatory Hello World example.

Chapter 2. Creating Your First Module

Now that we have covered some of the introductory aspects of Drupal 8 module development, it’s time to dive right into the meat of what we are doing here--module creation.

Here are some of the important topics that we will cover in this chapter:

- Creating a new Drupal 8 module--the files that are necessary to get started
- Creating a route and controller
- Creating and using a service
- Creating a form
- Creating a custom block
- Working with links
- Using the Event Dispatcher

Concretely, in this chapter, we will create a new custom module called *Hello World*. In this module, we will define a route that maps to a Controller and that outputs the age-old programming message. So, this will be our first win.

Next, we will define a service that our Controller will use to pimp up our message. After all, we don't want the same message presented to the user all day long. This simple example, however, will illustrate what services are and how to interact with the Service Container in order to make use of them.

Then, we will create a form where an administrator will be able to override the message shown on our page. It...

Creating a module

Creating a simple Drupal 8 module is not difficult. You only need one file to get it recognized by the core installation and to be able to enable it. In this state, it won't do much, but it will be installable. Let's first take a look at how to do this, and then we will progressively add meat to it in order to achieve the goals set out at the beginning of the chapter.

Custom Drupal 8 modules typically belong inside the `/custom` directory of the `/modules` folder found inside the root Drupal installation. You would put contributed modules inside a `/contrib` directory instead, in order to have a clear distinction. This is a standard practice, so that is where we will place our custom module, called *Hello World*.

We will start by creating a folder called `hello_world`. This will also be the module's machine name used in many other places. Inside, we will need to create an info file that describes our module. This file is named `hello_world.info.yml`. This naming structure is important--first, the module name, then `info` and followed by the `.yml` extension. You will hear about this file being often referred to as the module's

Using services in Drupal 8

Before we go and use our service in the Controller we created, let's take a breather and run through the ways you can make use of services once they are registered.

There are essentially two ways--statically and injected. The first is done by a static call to the Service Container, whereas the second uses dependency injection to pass the object through the constructor (or in some rare cases, a setter method). However, let's check out how, why, and what is the real difference.

Statically, you would use the global `Drupal` class to instantiate a service:

```
$service = \Drupal::service('hello_world.salutation');
```

This is how we use services in the `.module` files and classes, which are not exposed to the Service Container and into which we cannot inject--although the latter instances are rare. A few popular services also have shorthand methods on the `\Drupal` class, accessing them faster (and easier for IDE autocompletion), for example, `\Drupal::entityTypeManager()`. I recommend that you inspect the `\Drupal` class and take a look at the ones with shorthand methods available.

It is not the best practice, and for me, it is...

Blocks

Custom blocks in Drupal 8 are plugins. Finally, we encounter our first plugin type. For the sake of full disclosure, the *content* blocks that you create through the UI to place in a region and the custom blocks that are placed in a region are content entities. So, the block system is a good example of how entities and plugins work hand in hand in Drupal 8, and not to make matters too complex for you, configuration entities (about which we will learn in a later chapter) also play a big role here.

The block system in Drupal 8 is a great shift from its predecessor. Before, you had to implement two obligatory hooks plus two optional hooks if you wanted the block to have a configuration, and the latter was always saved somewhere that had nothing to do with the block itself. In Drupal 8, we work with a simple plugin class that can be made container-aware (that is, we can inject dependencies into it) and we can store configuration in a logical fashion.

So, how do we create a custom block plugin easily? All we need is one class, placed in the right namespace--`Drupal\module_name\Plugin\Block`. In this case (with plugins), the folder...

Working with links

One of the principle characteristics of a web application is the myriad of links between its resources. They are in fact the glue that brings it together. So, in this section, I want to show you a few common techniques used while working with links programmatically in Drupal 8.

There are two main aspects when talking about link building in Drupal--the URL and the actual link tag itself. So, creating a link involves a two-step process that deals with these two, but can also be shortened into a single call via some helper methods.

The URL

URLs in Drupal 8 are represented with the `Drupal\Core\Url` class, which has a number of static methods that allow you to create an instance. The most important of these is `::fromRoute()`, which takes a route name, route parameters (if any are needed for that route), and an array of options to create a new instance of `Url`. There are other such methods available that turn all sorts of other things into a `Url`, most notably, the `::fromUri()` method that takes in an internal or external URL. Sometimes, these methods are very helpful, especially when dealing with dynamically obtained data...

Event Dispatcher and redirects

A common thing you'll have to do as a module developer is to intercept a given request and redirect it to another page, and more often than not, this will have to be dynamic, depending on the current user or other contextual info. Drupal 7 developers know very well that this has always been an easy task. Simply implement `hook_init()`, which gets called on each request and then use the famous `drupal_goto()` function. This, however, is no longer the case in Drupal 8. What we have to do now is subscribe to the `kernel.request` event (remember this from the preceding chapter?) and then change the response directly. However, before seeing an example of this, let's take a look at how we can perform a simpler redirect from within a Controller.

Redirecting from a Controller

In this chapter, we wrote a Controller that returns a render array. We know from the preceding chapter that this is picked up by the theme system and turned into a response. In Chapter 4, *Theming*, we will go into a bit more detail and see how this process is done. However, this render pipeline can also be bypassed if the Controller returns a...

Summary

In this chapter, we covered a great deal of info about the things you need to know when developing Drupal 8 modules. The first thing we did was create our very own module skeleton that can be installed on a Drupal 8 site. Then, we saw how to create a new page at a specific path (route) and show some basic data on that page. Nothing too complex, but enough to illustrate one of the most common tasks you will do as a module developer. We then took that to new level and abstracted the logic for that data calculation into a service. Not only that, but we also saw how we can use that service, and, more importantly, how we should use it. Next, we saw how we can work with the Form API in Drupal 8 to allow administrators to add some configuration to the site. A very important takeaway here was also that the Form API page in Drupal 8 will prove invaluable going forward because you have many different types of form elements at your disposal. So, keep that close by. Also, since we talked about forms, we also saw how we can alter the existing forms defined by other modules--an invaluable technique for any module developer.

Next, we...

Chapter 3. Logging and Mailing

In the preceding chapter, we learned how to do some of the more common things most Drupal 8 module developers will have to know how to do, starting with the basics, that is, creating a Drupal module.

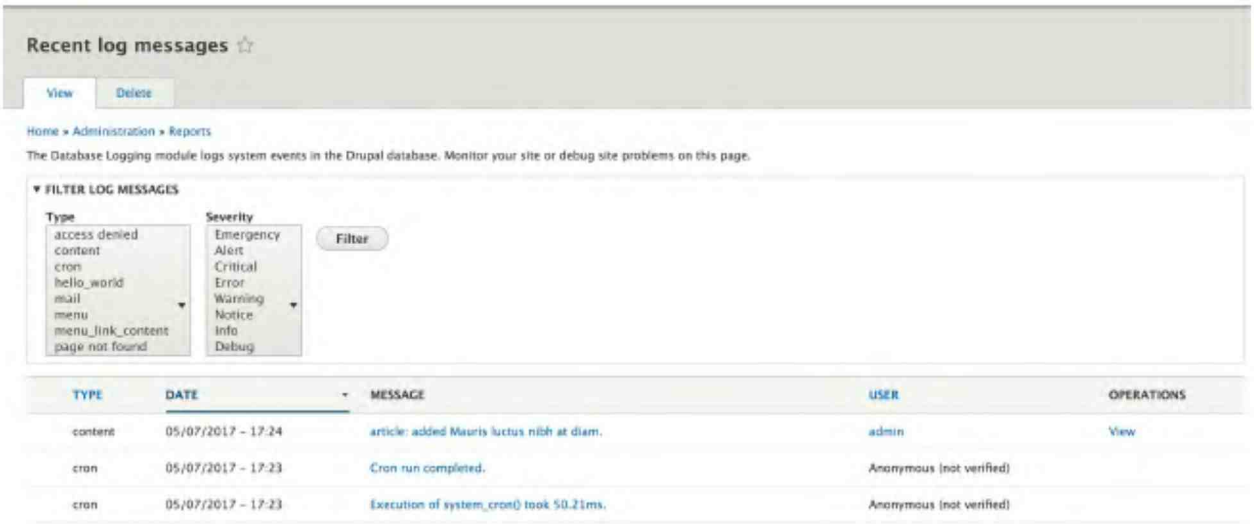
In this chapter, we will take things further and cover some other important tasks that a developer will have to perform.

- We will take a look at how logging works in Drupal 8. In doing so, we will cover some examples by expanding on our *Hello World* module.
- We will look at the Mail API in Drupal 8, namely, how we can send emails with the default setup (PHP mail). However, more than that, I will show you how to create your own email system to integrate with your (perhaps external) mail service; remember plugins? This will be yet another good example of using a plugin to extend existing capabilities.
- At the end of the chapter, we will also look at the Drupal 8 token system. We'll do so in the context of us being able to replace certain *tokens* with contextual data so that the emails we send out are a bit more dynamic.

By the end of this chapter, you should be able to add logging to your Drupal 8 module and feel...

Logging

The main logging mechanism in Drupal is a database log by which client code can use an API to save messages into `watchdog` table. The messages in there are cleared after they reach a certain number, but meanwhile they can be viewed in the browser via a handy interface (at `admin/reports/dblog`):



Alternatively, a core module that is disabled by default, Syslog, can be used to complement/replace this logging mechanism with the Syslog of the server the site is running on. For the purpose of this book, we will focus on how logging works with any mechanism, but we will also take a look at how we can implement our own logging system in Drupal 8.

Drupal 7 developers are very familiar with the `watchdog()` function they use for logging their messages. This is a procedural API for logging that exposes a simple function that takes some parameters, which make the logging flexible--`$type` (the category of the message), `$message`, `$variables` (an array of values to replace placeholders found in the message), `$severity` (a constant), and `$link` (a link to where the message should link to from the UI). It's pretty obvious that this solution is a...

Mail API

Now that we know how to log things in our application, let's turn our attention to the Drupal 8 Mail API. Our goal for this section is to see how we can send emails programmatically in Drupal 8. In achieving this goal, we will explore the default mail system that comes with the core installation (which uses PHP mail), and also create our own system that can theoretically use an external API to send mails. We won't go all the way with the latter because it's beyond the scope of this book. We will stop after covering what needs to be done from a Drupal point of view.

In the next and final section, we will look at tokens so that we can make our mailings a bit more dynamic. However, before we do that, let's get into the Mail API in Drupal 8.

The theory of the Mail API

Like before, let's first cover this API from a theoretical point of view. It's important to understand the architecture before diving into examples.

Sending emails programmatically in Drupal is a two-part job. The first thing we will need to do is define something of a *template* for the email in our module. This is not a template in the traditional sense, but rather...

Tokens

The last thing we will cover in this chapter is the Token API in Drupal 8. We will cover a few theories and, as usual, demonstrate them via examples on our existing "Hello World" module code. We will do this in the context of the mails we are sending out for error logs.

It would be nice if we could include some personalized information in the mail text without having to hardcode it in the module code or configuration. For example, in our case, we might want to include in the email the username of the current user that is triggering the error log that is being emailed.

Let's first understand how the Token API works, before going into our "Hello World" module.

The Token API

Tokens in Drupal are a standard formatted placeholder, which can be found inside a string and replaced by a real value extracted from a related object. The format tokens use is `type:token`, where `type` is the machine-readable name of a token type (a group of related tokens), and `token` is the machine-readable name of a token within this group.

The power of the Token API in Drupal is not only given by its flexibility but also by the fact that it is already a...

Summary

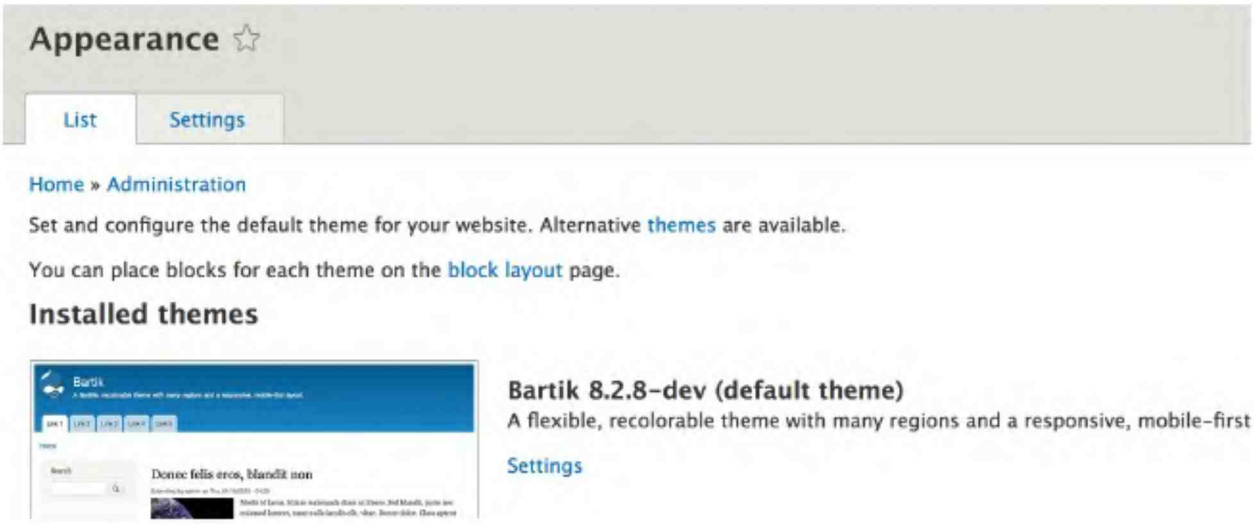
In this chapter, we discussed many things. We saw how logging works in Drupal 8, how the mail API can be used programmatically (and extended), and how the token system can be employed to make our text more dynamic.

While going through this chapter, we also enriched our *Hello World* module. So, apart from understanding the theory about logging, we created our own logging channel service and logger plugin. For the latter, we decided to send out emails when log messages were of the type *error*. In doing this, we took a look at the mail API and how we can use it programmatically. We saw that, by default, PHP's native `mail()` function is used to send out emails, but we can create our own plugin very easily to use whatever external service we want--yet another great example of extensibility via plugins.

Lastly, we looked at tokens in Drupal 8. We saw what components make up the API, how we can programmatically use existing tokens (replace them with the help of contextual data), and how we can define our own tokens for others to use. These are the main tenets of extensibility (and sharing)--using something someone else has exposed to...

Chapter 4. Theming

The most obvious part of Drupal's theming system is the **Appearance** admin page found at `admin/appearance`, which lists all the themes installed on your website. The page is shown in the following screenshot:



When you choose a theme from the **Appearance** page, you are applying a specific graphic design to your website's data and functionality. However, the applied theme is in reality only a small part of the entire theming layer.

This book mostly focuses on building modules that encapsulate chunks of a functionality. However, since we're ultimately building a web application, everything outputted by our functionality will need to be marked up with HTML. In Drupal, this process of wrapping data in HTML and CSS is called theming.

In this chapter, we will discuss how our module should integrate with the theme layer. We will talk about the architecture of the system, theme templates, hooks, render arrays, and others. Then, we will provide some practical examples.

Business logic versus presentation logic

We will start this chapter by discussing an important architectural choice modern applications make--how to turn data into a presentation.

So, what would be the best way to get our data and functionality marked up? Do we simply wrap each piece of data in HTML and return the whole as a giant string, as shown in the following example?

```
return '<div class="wrapper">' . $data . '</div>';
```

No, we don't. Like all other well-designed applications, Drupal separates its business logic from its presentation logic. It's true, previous versions of Drupal did have this kind of approach, especially when it came to theme functions, but even so, they were easily overridable. So, constructs like these were not found smack in the middle of business logic but were encapsulated in a special theming function that was called by the client code. So, the separation of business logic from presentation logic was clearly there, if at times, not so much one between PHP and HTML code.

Traditionally, the primary motivations for this separation of concerns were as follows:

- To make the code easier to maintain
- To make it possible...

Twig

Theme engines are responsible for doing the actual output via template files. Although previous versions of Drupal were capable of using different theme engines, one stood out and was used 99.9 percent of the time (statistic made up by me on the spot)--`PHPTemplate`. This theme engine used PHP files with `.tpl.php` extension and contained both markup and PHP. Seasoned Drupal developers grew accustomed to this practice, but it was always more difficult for frontend developers to use and theme against.

In Drupal 8, it was abandoned in favor of the Twig templating engine created by SensioLabs (the people responsible for the Symfony project). As mentioned, theme functions were also deprecated in favor of outputting everything through a `Twig` file. This brought about many improvements to the theme system and quite some joy to the frontend community. For example, it improved security and readability and made it much less important to be actually versed in PHP to be able to take part in the theming of a Drupal site.

All Twig template files in Drupal 8 have `.html.twig` extension.

Theme hooks

Since we have covered some of the principles behind the Drupal theme system--most notably, the separation of concerns--let's go a bit deeper and take a look at how they are actually put into practice. This all starts with the theme hooks. Yes, Drupal always loves to call things *hooks*.

Theme hooks define how a specific piece of data should be rendered. They are registered with the theme system by modules (and themes) using `hook_theme()`. In doing so, they get a name, a list of variables they output (the data that needs to be wrapped with markup), and other options.

The modules and themes that register theme hooks also need to provide an implementation (one that will be used by default). In Drupal 7, this was done in the following two ways--either via a PHP function that returned a string (markup) or a `PHPTemplate` template file. Both were equally important, but the latter was always more "correct" in my (and many people's) opinion. This is also supported by the fact that the functions in Drupal 8 have been completely ditched in favor of `Twig` templates. Also, together with a complete overhaul of the theme system, almost...

Theme hook suggestions

A great thing about theme hooks is their reusability. However, one problem you'll encounter is that theme hook templates lose context when a theme hook is reused. For example, the `item_list` theme hook whose definition we saw in the previous section has no idea what list it is theming. And this makes it difficult to style differently depending on what that content is. Fortunately, we can provide context to the theme system using a theme hook pattern instead of the original theme hook name, and this pattern looks something like this:

```
base_theme_hook__some_context
```

The parts of the pattern are separated with a double underscore, since some theme hooks could be confusing if we were to use a single underscore to delineate the parts of the pattern, and together they are called a *theme hook suggestion*. However, how does this work?

Client code (the render arrays as we will see soon), when using a theme hook to render a piece of data, can append the context to the theme hook. The theme system will then check for the following:

- If there is a template file that matches that suggestion (inside a theme), it uses it instead...

Render arrays

Render arrays also existed in the previous versions of Drupal and they were important to the theme system. In Drupal 8, however, they have become the thing--a core part of the Render API that is responsible for transforming markup *representations* into actual markup.

Acknowledging my limits as a writer, I will defer to the definition found in the Drupal.org documentation that best describes what render arrays are:

> *… a hierarchical associative array containing data to be rendered and properties describing how the data should be rendered.*

Simple, but powerful.

One of the principle reasons behind having render arrays is that they allow Drupal to delay the actual rendering of something into markup to the very last moment. In Drupal 7, often-times as module developers, we would call the actual rendering service on an array in order to turn it into markup,for example, in preprocessor functions so that the resulting strings can be printed in the template. However, this made it impossible to change that data later in the pipeline,for example, in another preprocessor that comes after the one which did the rendering.

For this...

Assets and libraries

Now that we know more about the render arrays, how they are structured and the pipeline they go through, we can talk a bit about asset management from a module development perspective, as although it is usually a theme responsibility, module developers also often have to add and use CSS and JS files to their modules, and it all happens in the render arrays.

Working with CSS and JS files has become standardized in Drupal 8 compared to its preceding version, where you had more than one way to do things. It is doing so via the concept of Libraries, which are now in Drupal 8 core and also work differently than their D7 contrib module counterpart (*Libraries API*). So, let's see what we have by going through some examples of making use of some CSS of JS files.

There are three steps to this process:

1. Creating your CSS/JS file.
2. Creating a library that includes them.
3. Attaching that library to a render array.

Libraries

Assuming that you already have the CSS/JS files, libraries are defined inside a `module_name.libraries.yml` file in the module root folder. A simple example of a library definition inside this file would look like...

Common theme hooks

In this section, we will look at three common theme hooks that come with Drupal core that you are likely to use quite often. The best way to understand them is, of course, referring to an example of how to use them. So, let's get to it.

Lists

One of the most common HTML constructs are lists (ordered or unordered), and any web application ends up having many of them, either for listing items or for components that do not even look like lists but for the purposes of marking up; an `ul` or `ol` fits the bill best. Luckily, Drupal has always had the `item_list` theme hook, which is flexible enough to allow us to use it in almost all cases.

The `item_list` theme hook is defined inside `drupal_common_theme()`, is preprocessed (by default) in `template_preprocess_item_list()`, uses the `item-list.html.twig` template by default, and has no default theme hook suggestions (because it's so generic and registered outside the context of any business logic). If we inspect its definition, we'll note that it takes a number of variables that build up its flexibility. Let's take a look at an example of how to use it.

Imagine that we have the...

Attributes

In the previous three examples of theme hooks, we encountered the concept of `attributes` in the context of using them to render HTML elements. Attributes here are understood in the same way as with HTML. For example, `class`, `id`, `style`, and `href` are all HTML element attributes. Why is this important?

The reusability of theme hooks makes it so that we cannot hardcode all our HTML attributes in the Twig template files. We can have some, including classes, but there will always be the case when the business logic will need to inform the theme hook of certain classes or other attribute values to print on the HTML element, for example, an `active` class on a link. This is why, we have this concept of attributes.

Most theme hooks you'll see will have attributes in some form or another, usually the variable being called `$attributes`, `$wrapper_attributes`, or something of this nature. Also, this variable always needs to be a multidimensional array with the attribute data you want to be passed. The keys in this array are the name of the attribute, whereas the value is the attribute value. If the value can have multiple items, such as...

Theming our Hello World module

The `HelloWorldController` we built in Chapter 2, *Creating Your First Module*, currently uses a service to retrieve the string to be used as the salutation and then returns a simple markup render array with it. Let's imagine now that we want to output this message, but wrap it in our own specific markup. To make an easy thing complicated, we want to break up the salutation string into parts so that they can be styled slightly differently. Additionally, we want to allow others to override our theme using suggestions that can depend on whether or not the salutation has been overridden via the configuration form. So, let's see how we can do these things.

To get things started, this is the markup we are after:

```
<div class="salutation">
    Good morning <span class="salutation--target">world</span>
</div>
```

The first thing we need to do is to define our own theme hook capable of outputting this. To this end, we implement `hook_theme()`:

```
/**
 * Implements hook_theme().
 */
function hello_world_theme($existing, $type, $theme, $path) {
  return [
    'hello_world_salutation' => [
      'variables' => ['salutation' =>...
```

Summary

The Drupal 8 theming system is complex and flexible, thus, it is impossible to cover it fully in one chapter of a module development book. However, we did go through the basics necessary to get you started--understanding the core tenets of the theme system, some of its most important Drupal specificities and practical use cases.

We started this chapter by discussing the abstract principle of separating business from presentation logic--a principle that is used by all modern web applications. We saw why it is critical for flexible and dynamic theming. Next, we discussed a great deal about how Drupal does this separation--the mighty theme hooks that act as a bridge between the two layers. Here, we also covered some of the highly used practices surrounding them--preprocessor functions and theme hook suggestions for added flexibility. Then, we covered how the business logic can actually use theme hooks--the render arrays (perhaps one of the most important Drupal constructs). Also, since we were on the subject, we outlined the Drupal and Symfony render pipeline to get a better understanding of the process building up an entire...

Chapter 5. Menus and Menu Links

Navigation is an important part of any web application. The ability to create menus and links easily in order to connect pages together is a core aspect of any content management system. Drupal 8 is fully equipped with both the site-building capabilities and developer API to easily build and manipulate menus and links.

In this chapter, we will discuss menus and menu links from a Drupal 8 module developer perspective. In doing so, we will touch upon a few key aspects:

- The general architecture of the menu system in Drupal 8
- Manipulating and rendering menus
- Defining various types of menu links

By the end of this chapter, you should be able to understand what menus and menu links are, how to use them in your code, and how to define menu links in your module. So, let's get started.

The menu system

Before we get our hands dirty with menus and menu links, let's talk a bit about the general architecture behind the menu system. To this end, I want to describe a bit about its main components, what some of its key players are, and what classes you should be looking at. As always, no great developer has ever relied solely on a book or documentation to figure out complex systems.

Menus

Menus are configuration entities represented by the following class: `Drupal\system\Entity\Menu`. I mentioned in Chapter 1 , *Developing for Drupal 8*, that we have something called configuration entities in Drupal 8, which we will explore in detail later in this book. However, for now, it's enough to understand that menus can be created through the UI and become an exportable configuration. Additionally, this exported configuration can also be included inside a module so that it gets imported when the module is first installed. This way, a module can ship with its own menus that are already created. We will see how this latter aspect works when we talk about the different kinds of storage in Drupal 8. For now, we will work with either...

Rendering menus

Now that we have covered some theory about the menu system, it's time to get our hands dirty with some code. The first thing we will look at is how to work with menus programmatically in view of rendering them in our module. For this, we will work with the default **`Administration`** menu that comes with Drupal core and has many links in it, at various levels. Note that the code we write in this section will not be included in the code repository.

Drupal core provides a block called `SystemMenuBlock`, which can be used to render any menu inside a block. However, let's take a look at how we can do this ourselves instead.

The first thing we will need to do is get the `MenuLinkTree` service. We can inject it or,if that's not possible, get it statically via the helper `\Drupal` class:

```
$menu_link_tree = \Drupal::menuTree();
```

Next, we will need to create a `MenuTreeParameters` object so that we can use it to load our menu tree. There are two ways we can do this. We can either create it ourselves and set our own options on it or we can get a default one based on the current route:

```
$parameters =...
```

Working with menu links

Now that we know how to load and manipulate trees of menu links, let's talk a bit more about the regular menu links. In this section, we will look at how our module can define menu links and how we can work with them programmatically once we get our hands on them from a tree or somewhere else.

Defining menu links

In our "Hello World" module, we defined a couple of routes, one of which is the `/hello` path, which shows our themed salutation component. Let's create a link to that path that goes in the main menu that comes with Drupal core.

As I mentioned, menu links are defined inside a `*.links.menu.yml` file. So, let's create that file for our module and add our menu link definition in it:

```
hello_world.hello:
  title: 'Hello'
  description: 'Get your dynamic salutation.'
  route_name: hello_world.hello
  menu_name: main
  weight: 0
```

In a typical YAML notation, we have the machine name (in this case, also the plugin ID) `hello_world.hello` followed by the relevant information below it. These are the most common things you will define for a menu link:

- The `title` is the menu link title, whereas the `description` is, by...

Defining local tasks

Let's now take a look at an example of how we can define local task links by heading back to our Hello World module. On the `/hello` page, let's add two local tasks--one for the regular `/hello` page, and the other for the configuration form where the salutation can be changed. This is a good example of using local tasks (tabs), as the configuration form is strictly related to what is on the page and is used to make changes to it.

As I mentioned, local tasks go inside a `*.links.task.yml` file. So, let's create one for our module with two links in it:

```
hello_world.page:
  route_name: hello_world.hello
  title: 'Hello World'
  base_route: hello_world.hello
hello_world.config:
  route_name: hello_world.greeting_form
  title: 'Configuration'
  base_route: hello_world.hello
  weight: 100
```

As usual, the top-most lines are the machine name (plugin IDs) of the links and we have the definitions under them. We have a `route_name` property again to specify what route these links should go to, a `title` for the link title, and a `base_route`. The latter is the route the local task should show up on. As you can see, both our links will...

Defining local actions

Nothing about our Hello World module calls for defining a local action link. So instead of doing that, let's check out one that actually makes sense. If you navigate to the `admin/content` screen, you'll see the `+ Add content` button. It looks exactly the same as the example we saw earlier on the user management page. That is a local action link for this route. The `+` styling indicates that these links are primarily used to add or create new items relevant to the current route.

This particular local action link is defined in the `node` module inside the `node.links.action.yml` file, and it looks like this:

```
node.add_page:
  route_name: node.add_page
  title: 'Add content'
  appears_on:
    - system.admin_content
```

Again, we have the machine name (plugin ID) and the definition. I hope that `route_name` and `title` are, by now, clear to you. A new thing here, though, is the `appears_on` key that is used to indicate the routes (plural) on which this action link should show up. So, a key feature is that one action link can exists on multiple pages.

Defining contextual links

Contextual links are a bit more complicated than the other types of links we've seen before, but nothing is too challenging for us. So let's take a look at how we can add contextual links to our salutation component so that users can override the message via a contextual link.

First, we will need to create the `.links.contextual.yml` file and define the link:

```
hello_world.override:
  title: 'Override'
  route_name: hello_world.greeting_form
  group: hello_world
```

Nothing too complicated here. Again, we have a `title` link and a `route_name`. Additionally, we have a `group` key, which indicates the group name that this link will be a part of. We will reference this later.

Next, we will need to alter our theme hook template file because the contextual links are printed in a `title_suffix` variable that is available in all theme hooks and is used by various modules to add miscellaneous data to templates. The Contextual module is one such example. So, we will need to get that printed. This is what it will look like now:

```
<div {{ attributes }}>
  {{ title_prefix }}
  {{ salutation }}
  {% if target %}
    <span...
```

Summary

In this chapter, we covered a lot of ground for working with menus and menu links. We started by getting an overview of the architecture of the menu system in Drupal 8. I threw many classes and hooks at you because I am a firm believer that the best way to learn is to dig into the code.

We also saw what types of menu links there are in Drupal 8. We not only have regular links that belong to actual menus but also all sorts of other utility link systems, such as local tasks, local actions, and contextual links.

Then, we got our hands dirty and started with a practical example of how to load menu links in a tree, manipulate them, and finally turn them into a render array. Right after that, we looked at how we can define all these types of menu links Drupal 8 comes with and also how to understand the individual menu links if you need to deal with them programmatically.

In the next chapter, we will look at one of the most important aspects of any kind of content management framework--the different types of data storage we can have in Drupal 8 and how we can work with them.

Chapter 6. Data Modeling and Storage

We have already gone through five chapters in this book, but we have yet to cover a topic that has to do with one of the main purposes of a CMS--data storage. Okay, we hinted at it in the preceding chapter and also saw an example of a configuration object in the second one. However, we merely scratched the surface of what is possible. It's now time to go ahead and dive into everything related to how you can store data in Drupal 8.

In this and the next chapter, we will talk about a lot of things related to storage and data manipulation and take a look at a lot of examples in the process. The focus of this chapter will, however, be more theoretical. There is a lot of ground to cover, as there are many APIs and concepts that you will need to understand. However, we will still see plenty of code examples to demonstrate in practice what we are talking about. In the next chapter, though, to make up for it, we will almost entirely work with code and build a few functionalities.

More concretely, however, this chapter will be divided into three main logical parts (not necessarily represented by...

Different types of data storage

Storing and using data are a critical part of any (web) application. Without somehow persisting data, we wouldn't be able to build much of anything. However, different uses of data warrant different systems for storing and manipulating it. For the purposes of this chapter, I will use the word *data* to mean almost anything that has to be persisted somewhere, for any given period of time.

If you've done development in Drupal 7, you already know a few ways of storing data--we had entities (primarily, the Node entity type, but others could be defined as well); the `variables` table, which was a relatively simple Key/Value store; and an API to interact with the database and do whatever we fancied. This caused many problems, such as a lack of consistency between APIs and much too heavy reliance on the database for configuration storage.

In Drupal 8, various layered APIs have been introduced to tackle common use cases for data storage. The strength of these new systems is mirrored in the fact that we rarely, if ever, need to even use the mother of all storage APIs, the database API. This is because everything...

State API

The State API is a Key/Value database storage and the simplest way you can store some data in Drupal 8. One of its main purposes is to allow developers to store information that relates to the *state* of the system (hence the name). Also, because the *state* of the system can be interpreted in various ways, think of this as simple information related to the current environment (Drupal installation) that is not editorial (content), for example, a timestamp of the last time they ran or any flags or markers the system sets to keep track of its tasks. It is different from caching in that it is not meant to be cleared as often and only the code that set it is responsible for updating it.

One of the main characteristics of this system is the fact that it is not for human interaction. I mean this in the sense that it is the application itself that needs to make use of it. The option for humans is the configuration system that we will talk about in detail in a later section. However, in quite a few cases, the latter falls short, and we need to (and it's perfectly acceptable) use the State system in order to store certain values...

Tempstore

The next system we will look at is the tempstore (temporary store).

The tempstore is a Key/Value, session-like storage system for keeping temporary data across multiple requests. Think of a multistep form or a wizard with multiple pages which are great examples of tempstore use cases. You can even consider "work in progress", that is, not yet permanently saved somewhere but kept in the tempstore so that a certain user can keep working on it until it's finished. Another key feature of the tempstore is that entries can have an expiration date at which point they get automatically cleared so that the user rushes the work.

There are two kinds of tempstore APIs--a private and shared one. The difference between the two is that with the first one, entries strictly belong to a single user, whereas with the second one, they can be shared between users. For example, the process of filling in a multistep form is the domain of a single user, so the data related to that must be private to them. However, that form can also be open to multiple users, in which case the data can either be shared between the users (quite uncommon) or used...

UserData

Now, I want to briefly talk about another user-specific storage option, also provided by the User module, called *UserData*.

The purpose of the UserData API is to allow the storage of certain pieces of information related to a particular user. Its concept is similar to the State API in that the type of information stored is not configuration that should be exported. In other words, it is specific to the current environment (but belonging to a given user rather than a system or subsystem).

Users are content entities, who can have fields of various data types. These fields are typically used for structured information pertaining to the user, for example, a first and a last name. However, if you need to store something more irregular, such as user preferences or flag that a given user has done something, the UserData is a good place to do that. This is because the information is either not something structured or is not meant for the users themselves to manage. So, let's see how this works.

The UserData API is made up of two things--the `UserDataInterface`, which contains the methods we can use to interact with it (plus developer...

Configuration

The configuration API is one of the most important topics a Drupal 8 developer needs to understand. There are many aspects to it that tie it into other subsystems, so it is critical to be able to both use and understand it properly.

In this subchapter, we will cover a lot about the configuration system. We start by understanding what configuration is and what it is typically used for. Then, we will go through the different options we have for managing configuration in Drupal 8, both as a site builder and a developer using the Drush commands. Next, we will talk about how configuration is stored, where it belongs, and how it is defined in the system. We will also cover a few ways that configuration can be overridden at different levels. Finally, we look at how we can interact with a simple configuration programmatically. So, let's begin with an introduction.

Introduction

Configuration is the data that the proper functioning of an application relies upon. It is those bits of information that describe how things need to behave and helps control what code does. In other words, it configures the system to behave in a certain...

Entities

We have finally reached the point where we talk about the most complex, robust, and powerful system for modeling data and content in Drupal 8--the Entity API.

Entities have been around since Drupal 7, which shipped with a few types such as node, taxonomy terms, users, comments, files, and so on. However, Drupal core only provided a basic API for defining entities and loading them consistently. The *Entity API* contributed module bridged a large gap and provided a lot of functionality to make entities much more powerful. In Drupal 8, however, these principles (and more) are found in core as part of a robust data modeling system.

The Entity API integrates seamlessly with the multilingual system to bring fully translatable content and configuration entities. This means that most data you store can be translated easily into multiple languages. In Drupal 7, this was always a herculean task that involved over 10 contributed modules to achieve something not nearly as powerful as we have now.

Because there is so much to cover about entities, in this section we will start with just a general overview of the entity system. But not to...

TypedData

In order to really understand how entity data is modeled, we need to understand the TypedData API. Unfortunately, this API, at the time of writing, still remains quite a mystery for many. But you're in luck because in this section we're gonna get to the bottom of it.

Why?

It helps to understand things better if we first talk about why there was the need for this API. It all has to do with the way PHP as a language *is*, compared to others, and that is loosely typed. This means that in PHP it is very difficult to use native language constructs to rely on the type of certain data or understand more about that data, as opposed to languages like Java or Python.

The difference between the string `"1"` and integer `1` is a very common example. We are often afraid of using the `===` sign to compare them because we never know what they actually come back as from the database or wherever. So, we either use `==` (which is not really good) or forcefully cast them to the same type and hope PHP will be able to get it right.

In PHP 7, we have type hinting for scalar values in function parameters which is good, but still not enough. Scalar values...

Interacting with the Entity API

In this final section of the chapter, we're going to cover the most common things you will be doing with content and configuration entities. These are the main topics we discuss going forward:

- Querying and loading entities
- Reading entities
- Manipulating entities (update/save)
- Creating entities
- Rendering entities
- Validating entity data

So, let's hit it.

Querying and loading entities

One of the most common things you will do as a programmer is query for stuff, such as data in the database. This is what we were doing a lot in Drupal 7 to get our data. A lot. We'd either use the database API or simple query strings and load our data. However, in Drupal 8 the entity API has become much more robust and offers a layer that reduces the need to query the database directly. In a later chapter, we will see how to do that still for when things become more complex. For now, since most of our structured data belongs in entities, we will use the entity query system for retrieving entities.

The service we use for querying entities is the `entity.query` service (`QueryFactory`). So, we can inject this into our class or use it...

Summary

You didn't think you we are ever going to see this heading did you? This chapter has been very long and highly theoretical. We haven't built anything fun and the only code we saw was to exemplify most of the things we talked about. It was a difficult chapter as it covered many complex aspects of data storage and handling. But trust me, these things are important to know and this chapter can serve both as a starting point to dig deeper into the code and a reference to get back to when unsure of certain aspects.

We saw what the main options for storing data in Drupal 8 are. Ranging from the State API all the way to entities, you have a host of alternatives. After covering the more simple ways, such as the State API, the private and shared tempstores and the UserData API, we dove a bit more into the configuration system which is a very important one to understand. There, we saw what kinds of configuration types we have, how to work with simple configuration, how it's managed and stored, and so on. Finally, in what is arguably the most complex part of the chapter, we looked at entities, both content and configuration. Just as...

Chapter 7. Your Own Custom Entity and Plugin Types

I am sure that you are looking forward to applying some of the knowledge gained from the previous chapters and do something practical and fun. As promised, in this chapter, we will do just that. Also, apart from implementing our own entity types, we will cover some new things as well. So, here's the game plan.

The premise is that we want to have products on our site that hold some basic product information, such as an ID, a name, and a product number. However, these products need to somehow get into our site. One way will be manual entry. Another, more important way will be through an import from multiple external sources (such as a JSON endpoint). Now, things will be kept simple. For all intents and purposes, these products aren't going to do much, so don't expect an e-commerce solution being laid out for you. Instead, we will practice modeling data and functionality in Drupal 8.

First, we will create a simple content entity type to represent our products. In doing so, we will make sure that we can use the UI to create, edit, and delete these products with ease by taking...

Custom content entity type

As we saw in the preceding chapter, when looking at the Node and NodeType entity types, entity type definitions belong inside the `Entity` folder of our module's namespace. In there, we will create a class called `Product`, which will have an annotation at the top to tell Drupal this is a content entity type. This is the most important part in defining a new entity type:

```
namespace Drupal\products\Entity;

use Drupal\Core\Entity\ContentEntityBase;
use Drupal\Core\Entity\EntityChangedTrait;
use Drupal\Core\Entity\EntityTypeInterface;
use Drupal\Core\Field\BaseFieldDefinition;

/**
 * Defines the Product entity.
 *
 * @ContentEntityType(
 *   id = "product",
 *   label = @Translation("Product"),
 *   handlers = {
 *     "view_builder" = "Drupal\Core\Entity\EntityViewBuilder",
 *     "list_builder" = "Drupal\products\ProductListBuilder",
 *
 *     "form" = {
 *       "default" = "Drupal\products\Form\ProductForm",
 *       "add" = "Drupal\products\Form\ProductForm",
 *       "edit" = "Drupal\products\Form\ProductForm",
 *       "delete" = "Drupal\Core\Entity\ContentEntityDeleteForm",
 *     },
 *    "route_provider"...
```

Drush command

So our logic is in place, but we will need to create a handy way we can trigger the imports. One option is to create an administration form where we go and press a button. However, a more typical example is a command that can be added to the crontab and that can be run at specific intervals automatically. So that's what we are going to do now, and we will do so using Drush.

The Drush command we are going to write will take an optional parameter for the ID of the importer configuration entity we want to process. This will allow the use of the command for more than one just importer. Alternatively, passing no options will process each importer (in case this is something we want to do later on). One thing to note is that we won't focus on performance in this example. This means the command will work just fine for smaller sets of data (as big as one request can process) but will be probably better to use a queue and/or batch processing for larger sets. Also, we will have a chapter dedicated for these subsystems later on, but, for now, let's get on with our example.

Before we actually write our new Drush command, let's...

Summary

In this chapter, we got to implement some fun stuff. We created our very own content and configuration entity types as well as a custom plugin type to handle our logic.

What we built was a Product entity type that holds some product-like data in various types of fields. We even created a bundle configuration entity type so that we can have multiple types of products with the possibility of different fields per bundle, a great data model.

We wanted to be able to import products from all sorts of external resources. For this reason, we created the Importer plugin type that is responsible for doing the actual imports--a great functional model. However, these only work based on a set of configurations, which we represented via a configuration entity type. These can then be created in the UI and exported into YAML files like any other configuration.

Finally, to use the importers, we created a Drush command, that can process either a single Importer or all the existing ones. This can be used inside a crontab for automatic imports.

There are still some shortcomings to the way we constructed the importing functionality, though. For...

Chapter 8. The Database API

In the previous two chapters, we discussed and our options as Drupal 8 module developers for modeling and storing data in Drupal 8. We also saw some examples of how to use things such as the State, Configuration, and Entity APIs, going to greater lengths with the latter by using it to build something useful. One of the key takeaways from these chapters is that--compared to Drupal 7, at least--the need for custom database tables and/or direct queries against these and the database has become minimal.

The Entity system is much more flexible and robust, the combination of configuration and content entities providing much of the needs for storing data. Moreover, the Entity query and loading mechanisms have also made finding them easy.Odds are, this is enough for most of your cases.

Furthermore, storage subsystems such as the State API (key value) and UserData have also removed much of the need for creating custom tables to store that kind of "one-off" data. Also, the configuration API provides a unified way to model exportable data, leaving no need for something else.

However, apart from these features,...

The Schema API

The purpose of the Schema API is to allow the definition of database table structures in PHP and to have Drupal interact with the database engine and turn these definitions into a reality. Apart from the fact that we don't ever have to see things such as `CREATE TABLE`, we ensure that our table structures can be applied to multiple types of databases. If you remember, in Chapter 1, *Developing for Drupal 8*, it was mentioned that Drupal can work with MySQL, PostgreSQL, SQLite, and others, if they support PDO, so the Schema API ensures this cross-compatibility.

The central component of the Schema API is `hook_schema()`. This is used to provide the initial table definitions of a given module. Implementations of this hook belong in the `*.install` file of the module and are fired when the module is first installed. If alterations need to be made to the existing database tables, there are a number of methods that can be used inside update hooks to make these changes.

In this section, we will create a new module called `sports` in which we want to define two tables--`players` and `teams`. The records in the former can reference records...

Running queries

Now that we have some tables to work with, let's take a look at how we can run queries against them. If you are following along, for testing purposes, feel free to add some dummy data into the tables via the database management tool of your choice. We will look at `INSERT` statements soon, but before that, we will need to talk about the more common types of queries you'll run--`SELECT`.

Queries using the Drupal 8 database abstraction layer are run using a central database connection service--`database`. Statically, this can be accessed via a shortcut:

```
$database = \Drupal::database();
```

This service is a special one compared to the ones we saw before, because it is actually created using a factory:

```
database:
   class: Drupal\Core\Database\Connection
   factory: Drupal\Core\Database\Database::getConnection
   arguments: [default]
```

This is a definition by which the responsibility for the instantiation is delegated to the factory mentioned, instead of the container as we've seen before. So, the resulting class does not necessarily need to match the one specified for the `class` key. However, in this case, the

Summary

In this chapter, we looked at the basics of interacting with the database API. Although it's something that has taken a significant step back in importance in day-to-day Drupal module development, it's important to understand and be able work with.

We started the chapter by creating our very own database tables to hold player and team information in a relational way. We've done so using an API that transforms definitions into actual tables without us having to even understand much of MySQL. The SQL terminology and basic operations are, however, something that every developer should be familiar with, notwithstanding their actual day to day application in Drupal.

Then, we looked at some examples of how we can run `SELECT`, `INSERT`, `UPDATE`, and `DELETE` queries using both the more SQL-oriented way of writing statements and the query builder approach, which uses an OO representation of the queries. We've also seen how these queries can be wrapped into transactions (where supported) so that we can commit data changes while minimizing the potential for incomplete or corrupt data. Finally, we've seen how these queries can be altered...

Chapter 9. Custom Fields

In Chapter 6, *Data Modeling and Storage*, and Chapter 7, *Your Own Custom Entities and Plugin Types*, we talked quite extensively about content entities and how they use fields to store the actual data that they are supposed to represent. Then, we saw how these fields, apart from interacting with the storage layer for persisting it, extend TypedData API classes in order to organize this data better at the code level. For example, we saw that the `BaseFieldDefinition` instances used on entities are actually data definitions (and so are the `FieldConfig` ones). Moreover, we also saw the Datatype plugins at play there, namely the `FieldItemList` with their individual items, which down the line extend a basic DataType plugin (`Map` in most cases). Also, if you remember, when we were talking about these items, I mentioned how they are actually instances of yet another plugin--`FieldType`. So, essentially, they are a plugin type whose plugins extend plugins of another type. I recommend that you revisit that section if you are fuzzy on the matter.

Most of these concepts are buried inside the Entity API and are only seen and...

Field type

The primary plugin type for creating a field is, as we discussed, the `FieldType`. It is responsible for defining the field structure, how it is stored in the database, and various other settings. Moreover, it also defines a default widget and formatter plugin that will be autoselected when we create the field in the UI. You see, a single field type can work with more than one widget and formatter. If more exist, the site builder can choose one when creating the field and adding it to an entity type bundle.

Otherwise, it will be the default; each field needs one because without a widget, users can't add data, and without a formatter, they can't see it. Also, as you'd expect, widgets and formatters can also work with more than one field type.

The field we will create in this section is for the license plate data, which, as we saw, needs two individual pieces of information--a code (such as the state code) and the number. License plates around the world are more complex than this, but I chose this example to keep things simple.

Our new `FieldType` plugin needs to go inside the `Plugin/Field/FieldType` namespace of a new module we...

Field widget

Our new license plate field type could be added to an entity type, but there would be no way users can use it. For this, we will need at least a widget. A given field type can work, however, with multiple widgets. So, let's create that default license plate widget plugin we referenced in the annotation of the field type, which belongs in the `Plugin/Field/FieldWidget` namespace of our module:

```
namespace Drupal\license_plate\Plugin\Field\FieldWidget;

use Drupal\Core\Field\FieldItemListInterface;
use Drupal\Core\Field\WidgetBase;
use Drupal\Core\Form\FormStateInterface;

/**
 * Plugin implementation of the 'default_license_plate_widget' widget.
 *
 * @FieldWidget(
 *   id = "default_license_plate_widget",
 *   label = @Translation("Default license plate widget"),
 *   field_types = {
 *     "license_plate"
 *   }
 * )
 */
class DefaultLicensePlateWidget extends WidgetBase {}
```

Again, we started by examining the annotation and class parents for just a bit. We will notice nothing particularly complicated, except maybe the `field_types` key, which specifies the `FieldType` plugin IDs this widget can work with. Just as a field type can have...

Field formatter

Alright, so our field now also has a widget that users can input data with. We can already use this field if we want, but when viewing the nodes, we have no way of displaying the field data, unless we do some custom preprocessing and retrieve it manually as we've seen earlier in the book. So, let's instead create the default field formatter because even if we don't need one, it's still a good practice to have one in place to make the field whole.

Before actually coding it, let's establish what we want our formatter to look and behave like. By default, we want the license plate data to be rendered like this:

```
 <span class="license-plate--code">{{ code }}</span> <span class="license-plate--number">{{
number }}</span>
```

So, each component is wrapped inside its own span tag, and some handy classes are applied to them. Alternatively, we may want to concatenate the two values together into one single span tag:

```
<span class="license-plate">{{ code }} {{ number }}</span>
```

This could be a setting on the formatter, allowing the user to choose the preferred output. So, let's do it then.

Field formatters go inside the

Field settings

When we created our field type, we specified some storage settings, and we saw that these are typically linked to underlying storage and cannot be changed once the field has data in it. This is because databases have a hard time making table column changes when there is data present in them. However, apart from storage settings, we also have something called field settings, which are specific to the field instance on a certain entity bundle. Even more, they can (or should) be changeable even after the field has been created and has data in it. An example of such a field setting, which is available from Drupal core on all field types, is the "required" option that marks a field as required or not. So, let's see how we can add our own field settings to configure what we said we want to do.

Back in our `LicensePlateItem` plugin class, we start by adding the default field settings:

```
/**
 * {@inheritdoc}
 */
public static function defaultFieldSettings() {
  return [
      'codes' => '',
    ] + parent::defaultFieldSettings();
}
```

This is the same pattern we've been seeing by which we specify what are the settings and what are...

Using as a base field

In the beginning of this chapter, I stressed the importance of understanding the makeup of a field (type, widget, and formatter) for being able to easily define base fields on custom entity types. This understanding allows you to navigate through Drupal core code and discover their settings and use them on base fields. So, let's cement this understanding by seeing how our new field could be defined as a base field on a custom entity type.

Here is an example where we actually use all the available settings we defined for each of the three plugins. Note that any settings that are left out default to the values we specified in the relevant defaults method, as follows:

```
$fields['plate'] = BaseFieldDefinition::create('license_plate')
  ->setLabel(t('License plate'))
  ->setDescription(t('Please provide your license plate number.'))
  ->setSettings([
    'number_max_length' => 255,
    'code_max_length' => 5,
    'codes' => implode("\r\n", ['NY', 'FL', 'IL']),
  ])
  ->setDisplayOptions('view', [
    'label' => 'above',
    'type' => 'default_license_plate_formatter',
    'weight' => 5,
    'settings' => [
     ...
```

Summary

In this chapter, we covered how to create custom fields that site builders (and developers) can add to entity types. This has implied defining three plugin types--`FieldType`, `FieldWidget`, and `FieldFormatter`, each with its own responsibility. The first defined the actual field and its storage and individual data properties, using the TypedData API. The second defined the form through which users can input field data when creating or editing entities that use the field. The third defined how the values inside this field can be displayed when viewing the entity.

We also saw that each of these plugins can have arbitrary sets of configurable settings that can be used to make the field dynamic--both in how the widget works and in how the values are displayed. Moreover, these settings are parts of exported field configuration, so we saw how we can define their respective configuration schemas.

Lastly, we also saw how--aside from creating our new field through the UI--developers can add it to an entity type as a base field, making it available on all bundles of that entity type.

In the next chapter, we will talk about access control,...

Chapter 10. Access Control

We've already talked about quite a few topics in the previous chapters, but we have been purposefully omitting an important aspect in many of them--access control. Much of what we covered deals in some way or another with access, but we have kept it out of our discussions to keep things more to the point. However, access control is an immensely important topic for Drupal development because it has implications in almost everything we do. So, for this purpose, we have a chapter dedicated to it in which we will cover the most important things you need to know in order to keep your application secure.

When I say secure I don't mean writing code in a secure way to avoid your site getting hacked. For that, we have an appendix at the end of the book to give you some pointers. Instead, I mean handling access control programmatically to ensure that your pages and any other resources are only accessible to the right users.

In this chapter, aside from introducing new concepts that stand on their own, we'll be revisiting some of the previous topics and seeing how we can apply access control in that context. We...

Introduction to the Drupal access system

If you've been doing some site building in Drupal 8 or have experience with previous versions of Drupal, you may already know a thing or two about roles and permissions. If not, no need to worry, as we will talk a bit about how these work.

Essentially, one of the things that makes Drupal special is the flexible access system it has out of the box, based on user roles and permissions. Roles are attributes that can be given to a user. The latter can have multiple roles assigned, but always has at least the default "Authenticated User" role. Permissions are the individual access indicators that can be assigned to roles. By the transitive property, users have all the permissions assigned to the roles they have been assigned. So, the end result is a matrix of permissions by role, and that's actually how it is visualized in the UI at `admin/people/permissions`:

PERMISSION	ANONYMOUS USER	AUTHENTICATED USER	ADMINISTRATOR
Block			
Administer blocks	☐	☐	☑
Comment			
Administer comment types and settings *Warning: Give to trusted roles only; this permission has security implications.*	☐	☐	☑
Administer comments and comment settings	☐	☐	☑
Edit own comments	☐	☐	☑
Post comments	☐	☑	☑
Skip comment approval	☐	☑	☑
View comments	☑	☑	☑

Drupal core, by default, comes with three roles--**`ANONYMOUS USER`**, **`AUTHENTICATED USER`**, and **`ADMINISTRATOR`**. Also, by default, there are a large number of permissions already defined by Drupal core (and contributed) modules, ready to be assigned...

Defining permissions

The way to create permissions in a custom module is by creating a `*.permissions.yml` file and adding the definitions in there. Consider the following example:

```
administer my feature:
  title: 'Administer my feature'
  restrict access: true
```

In this example, `administer my feature` is the machine name of the permission and actually the most important part. This is what you will use in your code to reference it. Then, we have a title that shows up on the permissions management page we saw earlier. Finally, we have a `restrict access` key by which we can specify whether we need a warning to be output on the permissions management page regarding the security implications, as follows--*Warning: Give to trusted roles only; this permission has security implications.* This is to indicate that our permission is more sensitive, and administrators should pay attention to who they assign it to. This option, can however, be left out (as you will in most cases actually).

You may have noted the static nature of this way of defining permissions. In other words, we hardcoded the permission name and only have one permission. In most...

Checking the user credentials

You can easily check whether a given user should access a certain resource as long as you have that user account at hand. Here, you can encounter two scenarios:

- You want to interrogate the current user
- You want to interrogate a given user, not necessarily the current one

As we saw in Chapter 2, *Creating Your First Module*, the current user is represented by a service, which implements the `AccountProxyInterface` interface. This service can be accessed by the `current_user` key or statically with this shorthand:

```
/** @var AccountProxyInterface $accountProxy */
$accountProxy = \Drupal::currentUser();
```

From this account proxy, we can request the `AccountInterface`, which represents the actual logged-in user account (the `UserSession` object). It holds a reference to the User entity, with a few of its "account" related data, but that is pretty much it. If we need to access its entity fields, we need to load the entity as we normally do:

```
/** @var UserInterface $user */
$user = \Drupal::entityTypeManager()->getStorage('user')->load($accountProxy->id());
```

The resulting `UserInterface`, by the way, also implements the same

Route access

Now that we saw how the access system works in Drupal 8 at a basic level, and how we can define permissions and check user credentials, it's time to talk about routes.

As we've seen from the very first time we wrote code in this book, routes are the entry points into your application. Also, as a developer, it is one of the main things you'll be dealing with, and controlling who exactly can access these routes falls under the purview of the access system.

There are a number of ways we can ensure that routes are only accessible to the right users, so let's see what these are.

The simplest way is by checking for a permission. We've actually done that in Chapter 2, *Creating Your First module*, when we defined our `hello_world.hello` route:

```
hello_world.hello:
  path: '/hello'
  defaults:
    _controller: '\Drupal\hello_world\Controller\HelloWorldController::helloWorld'
    _title: 'Our first route'
  requirements:
    _permission: 'access content'
```

The `requirements` key in a route definition contains all the data that the request trying to reach this route must have. This can contain mostly access-like information, and also things...

Entity access

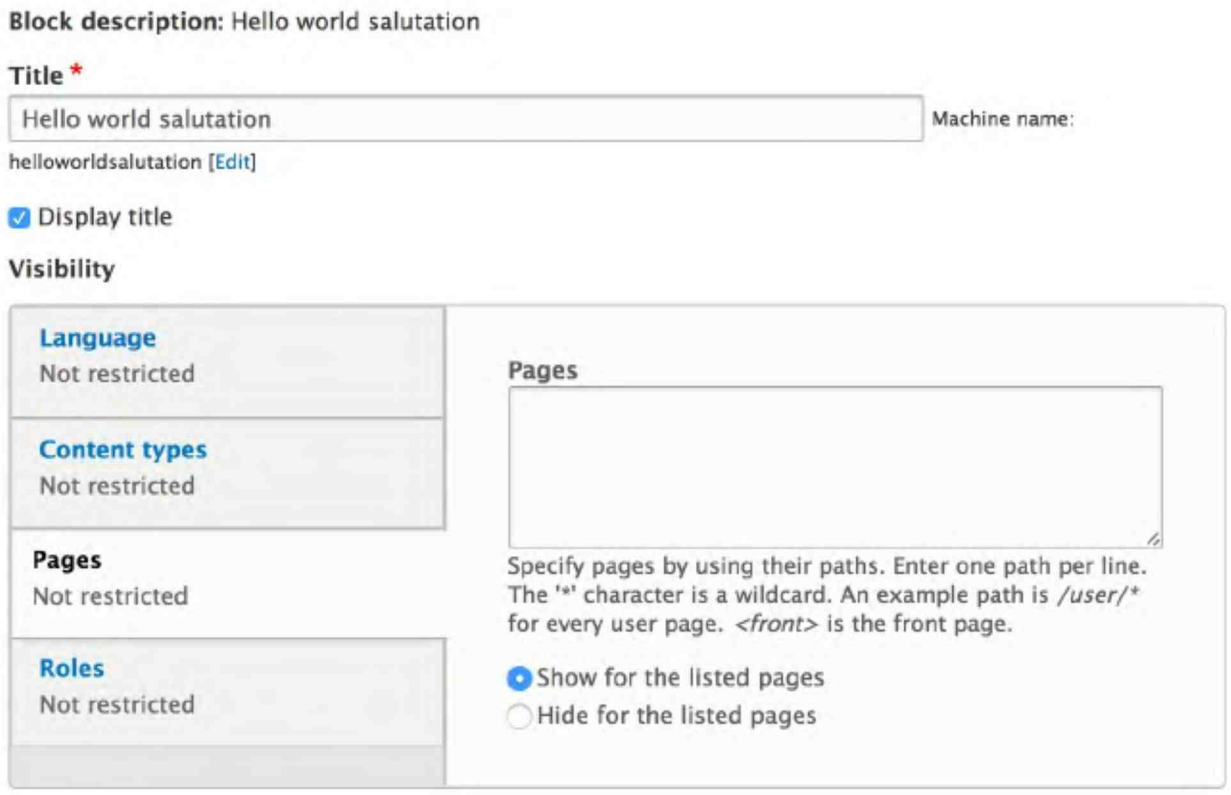
Now that we've covered how access control works on routes, let's dive into the entity access system and see how we can ensure that only the right users interact with our entities. To demonstrate these, we will work with the Product entity type we created in Chapter 7, *Your Own Custom Entity and Plugin Types*.

When we created the Product entity type, the annotation we wrote had an `admin_permission` property, where we referenced the general permission to be used for any interaction with the entities of this type. Since we didn't reference and implement an access control handler, this is the only access checking done on products. In many cases, this is enough. After all, entity types can be created for the sole purpose of structuring some data that nobody even needs to interact with in the UI. However, many other cases require more granular access control on operating with the entities, especially the content-oriented ones, such as Node.

There are four operations for which we can control access when it comes to entities--`view`, `create`, `update`, and `delete`. The first one is clearly the most common one, but we always need to...

Block access

Another major area where you will deal with access is when trying to control access to a custom block. If you remember in Chapter 2, *Creating Your First Module*, we created the `HelloWorldSalutationBlock` plugin so that our salutation can also be rendered using a block. Now, that block can be placed in a region and even configured to show up only on certain pages, for certain user roles, or even on node pages restricted by bundle. This is all done in the UI:



However, this is oftentimes not enough, and you will want to have a block placed in a region and control yourself under what circumstances it should show up--enter block access.

Inside the `BlockBase` plugin base class, there is the `blockAccess()` method, which always returns positively. This is because, by default, all blocks will be rendered--once they are placed in a region. Unless, of course, they are configured to only show in certain cases, in which case a system of visibility based on the available contexts kicks in to control that. However, if we override this method in our block plugin class, we can control whether or not the block is shown. So, we can leave the...

Summary

In this chapter, we talked about many access-related topics and techniques. In doing so, we covered what you need to know when starting Drupal 8 module development. Of course, as you progress, you'll dive deeper into the code and learn more subtle aspects and advanced concepts that you can employ in your modules. However, what we covered should set you well on your way. So, what exactly did we talk about?

We started by introducing the high-level Drupal 8 access system that is made up of the matrix between roles and permissions. In doing so, we've seen how we can define permissions in code and also how we can check whether a user has those permissions. Of course, we looked at other ways we can check a user's credentials and saw how we can use the `AccountInterface` for this.

Then, we moved on to routes and saw all the various ways we can ensure access control there. In doing so, we covered simple checks such as permissions and roles, but also went into more advanced examples of using custom access checkers. We saw that these can be both static and service based to make access checking fully dynamic. To demonstrate these...

Chapter 11. Caching

Application performance has always been one of the pain points when performing development with Drupal, and there are many reasons for this. For example, PHP is not the fastest language out there. Many beginner Drupal developers fell pray to the multitude of modules available and go a bit overboard with enabling more than needed. Indeed, the Drupal architecture is simply not the most performant. In its defense though,it is a very complex architecture that does a lot out of the box will have some speed trade-offs.

One critical component in this game, however, is caching. For those of you not familiar with this term, caching is the application strategy of storing copies of processed code (or anything that results from it) in view of delivering it faster to the user when requested subsequent times. For example, when you go to a website, you browser is most likely going to *cache* (store) certain assets locally on your computer so that when you visit the site the next time, it can show them to you faster.

Although caching has been steadily improving with recent versions of Drupal, it was still lacking significantly....

Introduction

The first thing I would like to mention before getting into the meat of the Cache API is that this subsystem is one of the best documented ones (at the time of writing). You can check out the main entry page (https://www.drupal.org/docs/8/api/cache-api/cache-api), and I recommend keeping it close by when carrying out Drupal 8 development.

The Cache system in Drupal 8 provides the API needed to handle the creation, storage, and invalidation of cached data. From a storage perspective, it is extensible, allowing us to write our own custom cache *backends* (`CacheBackendInterface`). By default, however, cache data gets stored in the database and hence the default backend is `DatabaseBackend`. Going forward, we will focus only on this implementation, since it is the most commonly used one, especially when starting a new project. Quite often though, once the site becomes more complex, alternative caching backends can be employed for better performance--such as Memecache or Redis.

The simplest type of cache in Drupal 8 is the so called *Internal Page Cache*, whose functionality resides inside the *Page Cache* core module. The goal of...

Cacheability metadata

Cacheability metadata is used to describe the *thing* which is rendered with respect to its *dynamism*. Most of the time, as Drupal 8 module developers, we will be using this metadata when working with render arrays. We will see a bit later where else these come into play, but for now, let's see what the actual properties are and what they are used for in the context of render arrays.

When creating render arrays, there are a few things we need to think about when it comes to caching. We always need to think about these things.

Cache tags

The first thing we need to think about is what our render array depends on. Are we rendering some entity data? Are we using some configuration values? Or anything that might be changed elsewhere impacting what we have to render? If the answer is yes, we need to use *cache tags*. If we don't use them, our render array gets cached as it is, and if the underlying data changes, we end up showing our users stale content or data.

To look at this an other way, imagine a simple Article node. This content can be shown on its main detail page, in a listing of article teasers or even a listing...

Placeholders and lazy building

Now that we've seen a bit how the cacheability metadata can be used in the more common scenarios, let's shift gears and talk about those page components, which have highly dynamic data.

When we set the maximum age of our Hello World salutation to zero seconds (don’t cache), I mentioned that there are ways this can be improved in order to help performance. This takes the form of postponing the rendering of the respective bit to the very last moment with the help of placeholders. But first, a bit of background.

Each of the cache properties we talked about can have values that make caching the render array pointless. We've already talked about the maximum age being set to zero, but you can also argue very low expiration times to have the same effect. Additionally, certain cache tags can be invalidated too frequently, again making the render arrays that depend on what they represent pointless to cache. Finally, certain cache contexts can provide many variations that significantly limit the effectiveness of the cache to the point it may even be counterproductive (high storage cost).

Cache tags are...

Using the Cache API

So far in this chapter, we mostly preoccupied ourselves with render arrays and how we can expose them to the Cache API for better performance. It's now time to talk a bit about how cache entries are stored by default in Drupal and how we can interact with them ourselves in our code.

As mentioned earlier, a central interface for the cache system is the `CacheBackendInterface`, which is the interface any caching system needs to implement. It basically provides the methods for creating, reading, and invalidating cache entries.

As we might expect, when we want to interact with the Cache API, we use a service to retrieve an instance of the `CacheBackendInterface`. However, the service name we use depends on the cache *bin* we want to work with. Cache bins are repositories that group together cache entries based on their type. So, the aforementioned implementation wraps a single cache bin, and each bin has a machine name, so the service name will be in the following format--`cache.[bin]`. This means that for each cache bin, we have a separate service.

The static shorthand for getting this service looks like this:

```
$cache =...
```

Summary

In this chapter, we covered the main aspects related to caching in Drupal 8, any module developer needs to be familiar with. We introduced some key concepts and talked about the two main types of caching--Internal Page Cache (used for anonymous users) and Dynamic Page Cache (used for any kind of user).

We dug deeper into cacheability metadata, which is probably the most important and common thing we need to understand. It's imperative to use this properly so that all the render arrays we build are cached and invalidated correctly. We've also seen how block plugins have specific methods we can use to define their cacheability metadata and how access results should also receive cacheability dependencies as needed. Stemming from this, we also explored lazy builders and the *auto-placeholdering* strategies that allow us to handle highly dynamic components while maintaining good cacheability overall.

Lastly, we looked into using the Cache API ourselves in order to store, read, and invalidate our own cache entries. We even saw how to create our own custom cache bin.

Caching is a very important aspect of Drupal 8 module development....

Chapter 12. JavaScript and the Ajax API

So far in this book, we've only talked about topics that can be considered to relate to *backend development*. This means heavy PHP, working with the APIs and the database, and so on. This is because this book is oriented toward module developers rather than themers. Also, the author of this book is admittedly not a JavaScript, or any kind of frontend, developer.

Nevertheless, in this chapter we'll switch gears and talk a bit about *frontend development*, namely, how to work with JavaScript in a Drupal 8 application. This is because there are many things developers can and should be doing in their modules that require frontend technologies. There are a few approaches and techniques specific to Drupal when it comes to adding and using JavaScript files and we will talk about those here. Moreover, we will also prove how powerful Drupal 8 is in allowing us to do quite a bit of "JavaScript" work without actually writing a single line of JavaScript code.

So, there are a few things we will cover in this chapter.

First, we will talk about the approach of writing JavaScript in Drupal. You already learned...

JavaScript in Drupal

Drupal 8 relies on a number of JavaScript libraries and plugins to perform some of its frontend tasks. For example, the use of *Backbone.js* is another example of advancement from previous versions of Drupal when it comes to adopting established libraries rather than reinventing new ones. Of course, as we already saw, the ubiquitous *jQuery* library continues to be used in Drupal 8 as well. But of course, there's others.

Another thing I have already mentioned but which is helpful to bring up again is the fact that Drupal no longer loads things such as jQuery or its Ajax framework on all pages needlessly. For example, many pages serving anonymous users which do not require jQuery won't even load it. This can greatly improve performance. But it also means that when we define our libraries to include our own JavaScript files, we must always declare these as dependencies (if we need them). For example, jQuery is something you'll often depend on.

Drupal behaviors

One of the most important things you need to know when writing JavaScript files in Drupal is the concept of behaviors. But in order to understand that, let's...

Ajax API

Now that you are on your way and ready to write whatever JavaScript you need for your application, and you are able to integrate this with the Drupal backend APIs, let's take a look at the Ajax framework. There's a lot we can do on the client side without having to write a single line of JavaScript code.

The Drupal Ajax API is a robust system that allows us to define client-side interactions via PHP. The most common time we use Ajax is when we interact with forms--triggering certain actions that change the DOM without having to reload the page. We will demonstrate how all this works by expanding a bit more on the importer functionality we built in Chapter 7, *Your Own Custom Entity and Plugin Types*. Before, though, let's take a quick look at the simpler use case of Ajax in Drupal 8.

Ajax links

The simplest way to interact with Drupal's Ajax API is to add the class `use-ajax` to any link. This will cause the link to make an Ajax request to the path of the link rather than moving the browser to it. A similar thing can be done with the submit button of a form using the class `use-ajax-submit`. This makes the form submit via Ajax to...

Summary

In this chapter, we took on the client-side and talked about JavaScript and client-side capabilities in Drupal 8. We started with the approach we need to take when writing JavaScript in a Drupal context. You learned about behaviors, why they are important, and how to use them. We also saw how we can pass around data from the server (Drupal) to the client-side and make use of it in JavaScript.

Funnily enough, we then we switched to a no-JavaScript allowed policy for the rest of the chapter. We did this to prove how powerful the Drupal Ajax API is, in who we can perform complex server-to-client interactions even if we are not frontend developers that can write JavaScript code. And to demonstrate the API, we first looked at how simple links can be turned into Ajax requests. We followed that up with an important refactor to our earlier products importer functionality that relied on Ajax to make the Importer configuration entity form dynamic (dependent on the selected plugin). Let's not forget another nugget of information--dynamic configuration schema--which allows us to decouple the configuration entity data definitions from...

Chapter 13. Internationalization and Languages

Even though there have been great advancements across the board, Drupal 8 has a couple of almost revolutionary developments compared to its predecessor. Notable among these are the configuration API and the caching system, which are both light-years ahead of what was capable in Drupal 7. Another one is the multilingual initiative that sought to make Drupal fully multilingual out of the box--rather than having to use 20 contributed modules to achieve results that don't even come close. This also includes the internationalization (i18n: https://www.w3.org/standards/webdesign/i18n) aspect that allows sites to be translated in any of the installed languages.

In this chapter, we are going to talk about internationalization and multilingual features in Drupal 8 from the point of view of a module developer. Many of the built-in capabilities of this system are oriented toward site builders--enabling languages, translating content and configuration entities as well as the Drupal interface (for administrators and visitors alike). Our focus will be what we as module developers need to do...

Introduction

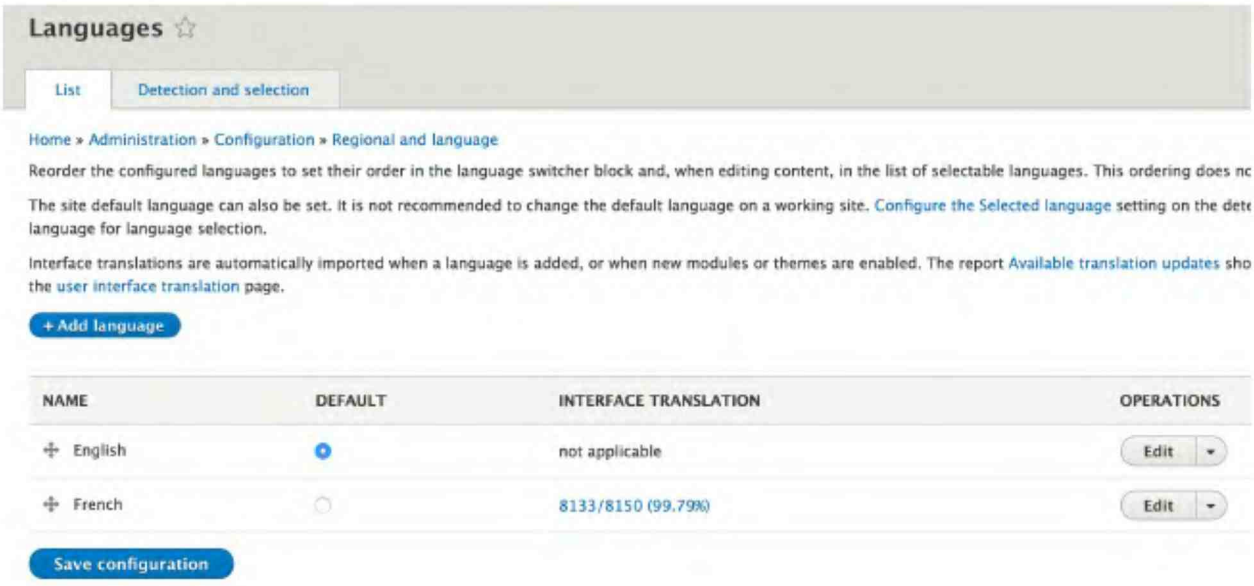
The multilingual and internationalization system is based on four Drupal core modules. Let's quickly go through them and see what they do:

- Language
- Content Translation
- Configuration Translation
- Interface Translation

Language

The *Language* module is responsible for dealing with the available languages on the site. Site builders can choose to install one or more languages from a wide selection. They can even create their own custom language if necessary. The installed languages can then be added to things such as entities and menu links in order to control their visibility, depending on the current language. Apart from the installed ones, Drupal 8 comes with two extra special languages as well--*Not Specified* and *Not Applicable*.

The modules also handles the contextual language selection based on various criteria, as well as provides a language switcher to change the current language of the site.



Content Translation

The *Content Translation* module is responsible for the functionality that allows users to translate content. Content entities are the principle vehicle for content, and with this module, the data inside can be...

Internationalization

The idea behind internationalization is to ensure that everything that gets output on the site can be translated into the enabled languages through a common mechanism--in this case, using the Interface Translation module. This refers to content, visible configuration values, and the strings and texts that come out of modules and themes. But there are many different ways this can happen, so let's see how in each of these cases we would ensure that our information can be translated.

A principle *rule* when writing Drupal modules or themes is to always use English as the code language. This is to ensure consistency and keep open the possibility that other developers will work on the same codebase, which may not speak a particular language. This is also the case for text used to be displayed in the UI. It should not be the responsibility of the code to output the translated text, but rather to always keep it consistent, that is, in English.

Of course, this is dependent on it being done right, in order to allow it to be translated via interface translation. There are multiple ways this can be ensured, depending on the...

Content entities and the Translation API

So far in this chapter, we've mostly talked about how to ensure that our modules output only text that can also be translated. The Drupal best practice is to always use these techniques regardless of whether the site is multilingual or not. You never know if you'll ever need to add a new language.

In this section, we are going to talk a bit about how we can interact with the language system programmatically and work with entity translations.

A potentially important thing you'll often want to do is check the current language of the site. Depending on the language negotiation in place, this can either be determined by the browser language, a domain, a URL prefix, or others. The `LanguageManager` is the service we use to figure this out. We can inject it using the `language_manager` key or use it via the static shorthand:

```
$manager = \Drupal::languageManager();
```

To get the current language, we do this:

```
$language = $manager->getCurrentLanguage();
```

Where `$language` is an instance of the `Language` class which holds some information about the given language (such as the language code and name). The language...

Summary

In this short chapter, we talked about the Drupal 8 multilingual and internationalization system from a module developer perspective. We started with an introduction to the four main modules responsible for languages and translating content, and configuration entities as well as interface text.

Then, we focused on the rules and techniques we need to respect in order to ensure that our output text can be translated. We saw how we can do this in PHP code, Twig, and YAML files, and even in JavaScript. Finally, we looked a bit at the language manager and Translation API to see how we can work with content entities that have been translated.

The main takeaway from this chapter should be that languages are important in Drupal 8 even if our site is only in one language. So, in developing modules, especially if we want to contribute them back to the community, we need to ensure that our functionality can be translated as needed.

In the next chapter, we are going to talk about data processing using batches and queues, as well as the cron system that comes with Drupal.

Chapter 14. Batches, Queues, and Cron

If in the previous chapter we kept things a bit more theoretical with me throwing "rules" at you, but in this chapter I am going to make up for it, and we are going to have some fun. This means we are going to write some code that demonstrates concepts related to data processing, especially larger amounts of it. And in doing so, we are going to cover a few topics.

First, we are going to look back at the `hook_update_N()` hook we saw in Chapter 8, *The Database API*, when we talked about the Database API. More specifically, we are going to see how the `&$sandbox` parameter can be used in order to handle updates that need to process some data, which may take a bit longer and should be split across multiple requests. Next up, we are going to look at standalone *batches* (which basically use the same system) to process data in batches across multiple requests. And what better example to illustrate this technique than with our Chapter 7, *Your Own Custom Entity and Plugin Types* Importer that needs to process an undefined number of products?

things for later processing (either in batches, during cron, or in...

Batch powered update hooks

The first thing we are going to look at is update hooks, revisiting of our previous Sports module created in Chapter 8 , *The Database API*. We will focus on the `&$sandbox` parameter we didn't use then. The goal is to run an update on each of our records in the `players` table and mark them as *retired*. The point is to illustrate how we can process each of these records one at the time in individual requests to prevent a PHP timeout, in case we have many records.

So to get us going, here is all the code, and we'll see right after what everything means:

```
/**
 * Update all the players to mark them as retired.
 */
function sports_update_8002(&$sandbox) {
  $database = \Drupal::database();

  if (empty($sandbox)) {
    $results = $database->query("SELECT id FROM {players}")->fetchAllAssoc('id');
    $sandbox['progress'] = 0;
    $sandbox['ids'] = array_keys($results);
    $sandbox['max'] = count($results);
  }

  $id = $sandbox['ids'] ? array_shift($sandbox['ids']) : NULL;

  $player = $database->query("SELECT * FROM {players} WHERE id = :id", [':id' => $id])-
>fetch();
  $data = $player->data ?...
```

Batch operations

Now that we have a basic understanding of Drupal's capabilities to do multi-request processing, let's switch gears and look at the Batch API.

In order to demonstrate how this works, we are going to rebuild the way our product `JsonImporter` plugin processes the product data it retrieves. Currently, we simply load all the products into an array of objects and loop through each, saving them to the database. So if there are 100,000 products in the JSON response, we might get into trouble with this approach. To be fair, if the remote provider has so many products, it usually provides a paginated way of requesting them by passing an offset and a limit. This keeps the payloads smaller (which is good for both communicating servers) and makes it easier on the processing. On our side, we can treat it as we would treat a database. But for now, we'll go with the assumption that the number of returned products is large, but not too large as to pose problems with the communication or with the ability of PHP to store them in memory.

Moreover, while illustrating the Batch API, we will also perform an operation we "forgot" in

Cron

In the previous section, we created an awesome multirequest batch processing of our JSON product import. In the next section, we'll jump into the Queue API and see how we can plan the processing of multiple items at a later stage. However, before we dive into that, let's talk a bit about how the Drupal 8 cron works and what we can do with it. This is because our discussion about the Queue API is closely related to it.

First of all, Drupal doesn't actually have a fully fledged cron system. That is because it's an application and not a server capable of scheduling tasks that run at specified times of the day at intervals. However, what it has is a cron-like system, which can come very close, especially on busy websites. Often, it is affectionately referred to as the poor man's cron. Why? Since Drupal cannot by itself do anything without any sort of impetus, it relies on visitors coming to the website to trigger the cron tasks. So, even if we can configure the frequency of Drupal's cron, we are relying on visitors coming to the website and triggering it inadvertently. Drupal then keeps track of when the cron ran and ensures that...

Queues

It's finally time to talk a bit about the Queue API, how it works, and what it's main components are; the theory, basically. We will do this before diving into code examples that we all thoroughly enjoy.

Introduction to the Queue API

The main purpose of the Queue API is to provide a way for us to add items to a *queue* in order to have them processed at a later time. In charge of processing these items are the *queue worker* plugins, which can be enlisted either automatically by the Drupal cron, manually (programmatically) by us, or by Drush. We will look at an example of all three.

The central player in this API is the implementation of the `QueueInterface`, which is the actual queue into which we put items. There are two types of queues Drupal can handle-- reliable and unreliable. The first preserves the order in which the items are processed (first in, first out) and guarantees that each item gets processed at least once. In this chapter, we will focus only on this type of queue. But there is also the possibility of working with unreliable queues which give their best effort when maintaining the item order and do not guarantee...

Summary

In this chapter, we looked at some of the ways we as module developers can set up simple and complex data processing tasks that can run at any time we want.

We started by looking into using the multirequest capabilities of the update hooks. This was a continuation from Chapter 8, *The Database API*, where we introduced them for the first time, and we have now seen how we can expand on their capabilities. Then, we turned to the more complex Batch API that uses similar, albeit more complex, techniques. This system allowed us to construct a series of operations that leveraged Drupal's multirequest capabilities. Our playground was the JSON products Importer, which now can handle large amounts of data without the concern of PHP memory timeouts. Next, we looked at how Drupal's cron system works and why it is there, and even saw an example of how as module developers we can hook into it and process our own tasks whenever it runs. But then, we took things to the next level with the introduction of the Queue API, which allowed us to add items to a queue so that they can get processed at a later stage. This processing, as we saw, can...

Chapter 15. Views

Views has always been a staple module for any Drupal site. It was so popular and needed that it ended up being incorporated into Drupal 8 core. So now, each new Drupal site ships with Views out of the box, fully integrated with the rest of the system and powering a great number of core features.

Essentially, Views is a tool for creating and displaying lists of data. This data can be almost anything, but we mostly use Drupal entities as they are now so robust. It provides the architecture to build and manipulate complex queries through the UI as well as many different ways of outputting the resulting data. From a module developer's point of View (yes, pun intended), much of this power has been broken down into multiple layers of building blocks, abstracted as plugins. Moreover, in keeping with tradition, there are also a multitude of hooks that are fired at different stages, with which we can programmatically contribute to or influence Views.

In this chapter, we will look at the Views ecosystem from a module developer's perspective. As such, we won't be spending that much time with its site-building capabilities...

Entities in Views

Even in Drupal 7, Views had a pretty good integration with the entity system. But seeing as there was no robust entity API to speak of, this integration was not so organic. It required more contributed modules and some custom code to make an entity type work with Views.

In Drupal 8, however, the two are very closely linked and it's a breeze exposing new content entities to Views. If you've followed along with Chapter 7 ,*Your Own Custom Entity and Plugin Types*, and have the Product entity type set up, you'll notice that if you try to create a View, you will have no option to make it based on products. That is because in the entity type definition, we did not specify it should be exposed to Views. That's all there is to it, actually. We just have to reference a new handler:

```
"views_data" = "Drupal\views\EntityViewsData"
```

That is it. Clearing the cache, we are now able to create Views with products that can show any of the fields, can filter and sort by them, and can even render them using View modes. All of these work consistently with the other entity types (at least fundamentally, as we will see in a moment).

You'll...

Exposing custom data to Views

To get a better understanding of how Views works, we are going to look at an example of totally custom data and how we can expose it to Views. Based on that, we will begin understanding the role of various plugins and can begin to create our own. Additionally, we'll be able to expand on our Product entity type data to enrich its Views interaction.

To exemplify all of this, we are going to revisit our sports module in which we declared the `players` and `teams` tables of data and which we will now be exposing to Views. The goal is to allow site builders to create dynamic listings of this data as they see fit. The lessons learned from this example can be applied to other data sources as well, even things such as remote APIs (with some extra work).

Views data

Whenever we want to expose data to Views, we need to define this data in a way Views can understand it. That is actually what `EntityViewsData::getViewsData()` does for content entities. However, since we are dealing with something custom, we can do so by implementing `hook_views_data()`. A lot can go into it, but we'll start things simple.

Let's implement...

Custom Views field

Now that we have seen how data is exposed to Views, we can start understanding the `NodeViewsData` handler I mentioned earlier (even if not quite everything) a bit better. But this also provides a good segue back to our `Product` entity type's `views_data` handler, where we can now see what the responsibility of `getViewsData()` is. It needs to return the definition for all the tables and fields, as well as what they can do. Luckily for us, the base class already provides everything we need to turn our product data into Views fields, filters, sorts, arguments, and potentially relationships, all out of the box.

But let's say we want to add some more Views fields that make sense to us in the context of our product-related functionality. For example, each product has a `source` field that is populated by the Importer entity from it's own `source` field. This is just to keep track of where they come from. So we may want to create a Views field that simply renders the name of the Importer that has imported the product.

You'll be quick to ask,"But hey, that is not a column on the products table! What gives?" As we will see, we can...

Custom Views filter

In a previous section we exposed our `players` and `teams` tables to Views, as well as made the team name a possible string filter to limit the resulting players by team. But this was not the best way we could have accomplished this because site builders may not necessarily know all the teams that are in the database nor their exact names. So we can create our own `ViewsFilter` to turn it into a selection of teams the user can choose from. Kind of like a taxonomy term filter. So let's see how it's done.

First, we need to alter our data definition for the team name field to change the plugin ID that will be used for the filtering (inside `hook_views_data()`):

```
'filter' => array(
  'id' => 'team_filter',
),
```

Now we just have to create that plugin. And naturally, it goes in the `Plugin/views/filter` namespace of our module:

```
namespace Drupal\sports\Plugin\views\filter;

use Drupal\Core\Database\Connection;
use Drupal\views\Plugin\views\filter\InOperator;
use Drupal\views\ViewExecutable;
use Drupal\views\Plugin\views\display\DisplayPluginBase;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Filter class which...
```

Custom Views argument

When we first exposed the player and team data to Views, we used an argument plugin so that we could have a contextual filter on the team ID a player belongs to. To do this, we used the existing `numeric` plugin on the actual `team_id` field of the `players` table. But what if we wanted an argument that works on more levels? For example, we don't exactly know what kind of data we'll receive, but we want to be able to handle nicely both a numeric one (team ID) and a textual one (team name). All in one argument. To achieve this, we can create a simple `ViewsArgument` plugin to handle this for us.

First thing, like always, is to define this field. We don't want to mess with the `team_id` field onto which we added the earlier argument as that can still be used. Instead, we'll create a new field, this time on the `teams` table, which we will simple call `team`:

```
$data['teams']['team'] = array(
  'title' => t('Team'),
  'help' => t('The team (either an ID or a team name).'),
  'argument' => array(
    'id' => 'team',
  ),
);
```

This time, though, we don't create a *field* for it as we don't need this displaying anything. Rather, we...

Views theming

Frontend developers had many pain points in Drupal 7 and many of them were also related to theming Views output. Luckily, Drupal 8 has made things much easier to handle. We will look at a bit of that here in order to nudge you in the right direction when applying what you learned in Chapter 4, *Theming*.

Views is very complex and is made up of many pluggable layers. A View has a *display* (such as a Page or Block), which can render its content using a given *style* (such as an Unformatted list or Table). Styles can decide whether to control the rendering of a given result item (row) themselves or delegate this to a *row* plugin (such as Fields or Entity). Most, in fact, do the latter. The two most common scenarios for using *row* plugins is either using the `EntityRow` one which renders the resulting entities using a specified view mode or the `Fields` plugin, which uses individual `ViewField` plugins to render each field that is added to the View.

If we wanted to theme a View, there are all these points we can look at. Want the View to output a slideshow? Maybe create a new *style* plugin. Want to do something crazy with each entity...

Views hooks

Views also comes with a lot of hooks. We've already see an important one that allowed us to expose our own data to Views. But there are many more, and you should check out the `views.api.php` file for more information.

Quite a few exist for altering plugin information for all sorts of plugin types. But there are also some important ones that deal with Views execution at runtime. The most notable of these is `hook_views_query_alter()`, which allows us to make alterations to the final query that is going to be run. There is also `hook_views_post_render()` and `hook_views_pre_render()`, which allow us to make alterations to the View results. For example, to change the order of the items or something like that.

I recommend you check out their respective documents and make yourself aware what you can do with these hooks. At times they can be helpful, even if with Drupal 8 most of the action happens in plugins and you can easily now write your own to handle your specific requirements. This is why we won't be going into great detail about these.

Summary

In this chapter, we looked at Views from all sorts of module developer-oriented angles. We saw how we can expose our product entity type to Views. That was a breeze. But then, we also saw how our custom player and team data from Chapter 8, *The Database API*, can also be exposed to Views. Even if we did have to write some code for that, much of it was quite boilerplate as we were able to leverage the existing Views plugin ecosystem for almost everything we wanted. However, since these are all plugins, we also saw how we can create our own field, filter, and argument plugins to handle those exceptional cases in which what exists may not be enough.

Closely tied to this, we also talked a bit about altering the way other modules expose their data to Views. The most notable example here was the ability to easily add more fields (and plugins) to entity based Views in order to enrich them with custom functionalities.

Finally, we talked a bit about how we can approach the theming aspect of Views. We saw the different layers that make one up, starting from the display all the way down to the *field*. We closed the chapter with a...

Chapter 16. Working with Files and Images

Drupal comes with many capabilities for handling and manipulating files and images and has been adding to its toolset more and more with recent versions. Of course, this is not to say that media management has not been always a pain point for Drupal developers. In Drupal 7, a complicated suite of contributed modules was needed to achieve a basic level of functionality, something that users of "competitors" like WordPress enjoy out of the box. In Drupal 8, there is more emphasis placed on media management, however much of the work is still in the contributed sphere. But due to the new way Drupal development will work, it's expected that much of this effort will find its way into the core system with future versions.

In this chapter, we will look at how we can work with files and images in Drupal, supported by the core features. We are not going to go into topics such as media management but rather focus on the module developer tools that can be used for working with files. We will see some examples along the way. So, what are we going to discuss?

First, we are going to get an understanding...

The filesystem

Drupal defines four main types of file storage for any given site--the *public*, the *private*, the *temporary* and the *translation* filesystems. When installing Drupal, the folders that map to these filesystems are created automatically. In case that fails--most likely due to permission issues--we have to create them ourselves and give them the right permissions. Drupal takes care of the rest (for example, adds relevant `.htaccess` files for security reasons). Make sure you check out the documentation on Drupal.org for how to successfully install Drupal 8 if you are unsure how this works.

Public files are available to the world at large for viewing or downloading. This is where things such as image content, logos, and anything that can be downloaded are stored. Your public file directory must exist somewhere under Drupal's root, and it must be readable and writeable by whatever *user* your web server is running under. Public files have no access restrictions. Anyone, at anytime, can navigate directly to a public file and view or download it. This also means that accessing these files does not require Drupal to bootstrap.

We...

Stream wrappers

If you've been writing PHP for a long time, you may have needed to work with local or remote files at some point. The following PHP code is a common way to read a file into a variable that you can do something with:

```
$contents = '';
$handle = fopen("/local/path/to/file/image.jpg", "rb");
while (!feof($handle)) {
 $contents .= fread($handle, 8192);
}
fclose($handle);
```

This is pretty straightforward. We get a handle to a local file using `fopen()` and read 8KB chunks of the file using `fread()` until `feof()` indicates that we've reached the end of the file. At that point, we use `fclose()` to close the handle. The contents of the file are now in the variable `$contents`.

In addition to local files, we can also access remote ones through `fopen()` in the same exact way but by specifying the actual remote path instead of the local one we saw before (starting with `http(s)://`).

Data that we can access this way is streamable, meaning we can open it, close it, or seek to an arbitrary place in it.

Stream wrappers are an abstraction layer on top of these streams that tell PHP how to handle specific types of data. When using a stream wrapper,...

Managed versus unmanaged files

The Drupal File API allows us to handle files in two different ways. Files essentially boil down to two categories: they are either *managed* or *unmanaged*. The difference between the two lies in the way the files are used.

Managed files work hand in hand with the Entity system and are in fact tied to File entities. So whenever we create a *managed* file, an entity gets created for it as well, which we can use in all sorts of ways. And the table where these records are stored is called `file_managed`. Moreover, a key aspect of *managed* files is the fact that their usage is tracked. This means that if we attach them to another entity, reference them, or even manually indicate that we use them, this usage is tracked in a secondary table called `file_usage`. This way, we can see where each file is used and how many times, and Drupal even provides a way to delete "orphaned" files after a specific time in case they are no longer needed.

A notable example of using *managed* files is the simple `Image` field type that we can add to an entity type. Using these fields, we can upload a file and *attach* it to the respective...

Using the File and Image fields

In order to demonstrate how to work with *managed* files, we will go back to our Product entity importer and bring in some images for each product. However, in order to store them, we need to create a field on the Product entity. This will be an *image* field.

Instead of creating this field through the UI and attaching it to a bundle, let's do it the programmatic way and make it a base field (available on all bundles). We won't need to do anything complex, for now we are only interested in a basic field that we can use to store the images we bring in from the remote API. It can look something like this:

```
$fields['image'] = BaseFieldDefinition::create('image')
  ->setLabel(t('Image'))
  ->setDescription(t('The product image.'))
  ->setDisplayOptions('form', array(
    'type' => 'image_image',
    'weight' => 5,
  ));
```

If you remember from Chapter 6, *Data Modeling and Storage* and Chapter 7, *Your Own Custom Entity and Plugin Types*, how this works is we are creating a base field definition which, in this case, is of the type `image`. This is the `FieldType` plugin ID of the `ImageItem` field. So that is where we need...

Working with managed files

In this section, we will look at two examples of working with managed files. First, we will see how we can import product images from our fictional remote JSON-based API. Second, we will see how to create a custom form element that allows us to upload a file, and use it in a brand new CSV based importer.

Attaching managed files to entities

Now that we have our product image field in place and we can store images, let's revisit our JSON response that contains the product data and assume it looks something like this now:

```
{
  "products" : [
    {
      "id" : 1,
      "name": "TV",
      "number": 341,
      "image": "tv.jpg"
    },
    {
      "id" : 2,
      "name": "VCR",
      "number": 123,
      "image": "vcr.jpg"
    }
  ]
}
```

What's new is the addition of the `image` key for each product, which simply references a filename for the image that goes with the respective product. The actual location of the images is at some other path we need to include in the code.

Going back to our `JsonImporter::persistProduct()` method, let's delegate the handling of the image import to a helper method called

Our own stream wrapper

At the beginning of this chapter, we briefly talked about stream wrappers and what they are used for. We saw that Drupal comes with four mainstream wrappers that map to the various types of file storage it needs. Now its time to see how we can create our own. And the main reason why we would want to implement one is to expose resources at a specific location to PHPs native file system functions.

In this example, we will create a very simple stream wrapper that can basically only read the data from the resource. Just to keep things simple. And the data resource will be the product images hosted remotely (the ones we are importing via the JSON Importer). So there will be some rework there, as well, to use the new stream wrapper instead of the absolute URLs. Moreover, we will also learn how to use the site-wide settings service by which we can have environment specific configurations set in the `settings.php` file and then read by our code.

The native way of registering a stream wrapper in PHP is by using the `stream_wrapper_register()` function. However, in Drupal 8 we have an abstraction layer on top of that in...

Working with unmanaged files

Working with *unmanaged* files is actually pretty similar to doing so with *managed* files, except that they are not tracked in the database using the File entity type. There is a set of helper functions similar to what we've seen for *managed* files that do the same things--but they have the word *unmanaged* in them. Let's see some examples.

To save a new file, we do almost like we did before:

```
$image = file_get_contents('products://tv.jpg');
$path = file_unmanaged_save_data($image, 'public://tv.jpg', FILE_EXISTS_REPLACE);
```

We load the file data from wherever and use the `file_unmanaged_save_data()` the same way as we did `file_save_data()`. The difference is that the file is going to be saved but no database record is created. So the only way to use it is to rely on the path it is saved at and either try to access it from the browser or use it for whatever purpose we need. The `file_unmanaged_save_data()` returns the URI of where the file is now saved or FALSE if there was a problem with the operation. So if all went well with the preceding example, `$path` would now be--`public://tv.jpg`.

And just like with the *managed*...

Private file system

The private file system is used whenever we want to control access to the files being downloaded. Using the default public storage, users can get to the files simply by pointing to them in the browser, thereby bypassing Drupal completely. However, `.htaccess` rules prevent users from directly accessing any files in the private storage making it necessary to create a route that delivers the requested file. It goes without saying that the latter is a hell of a lot less performant as Drupal needs to be loaded for each file. So it's important to only use it really when files should be restricted based on certain criteria.

Drupal already comes with a route and Controller ready to download private files but we can create one as well if we really need to. For example, the image module does so in order to control the creation and download of image styles--`ImageStyleDownloadController`.

The route definition for the default Drupal path looks like this:

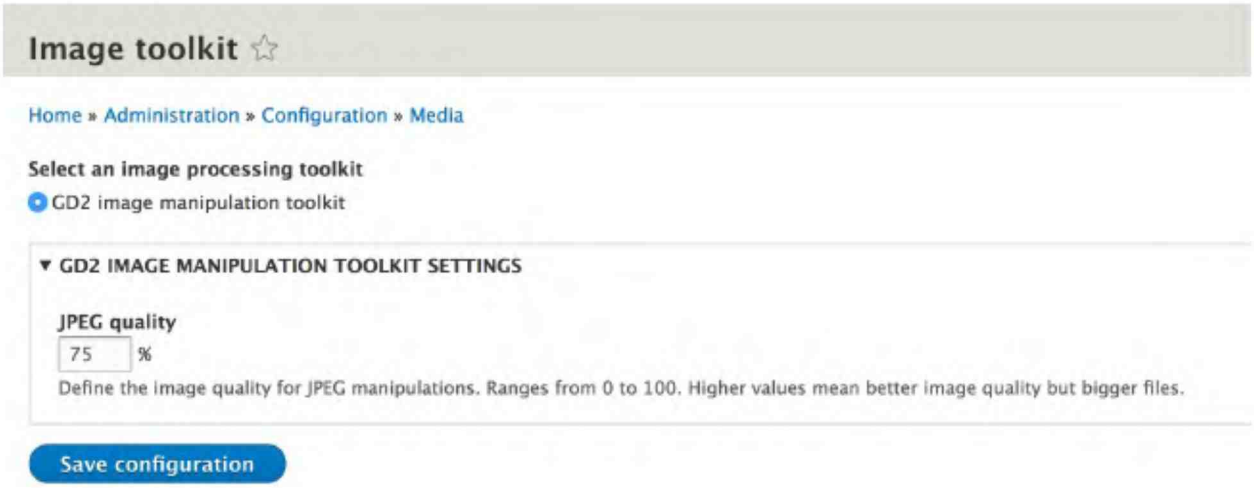
```
system.files:
  path: '/system/files/{scheme}'
  defaults:
    _controller: 'Drupal\system\FileDownloadController::download'
    scheme: private
  requirements:
    _access:...
```

Images

In this section, we are going a bit deeper into the world of images in Drupal 8 while keeping the focus on module developers.

Image toolkits

The Drupal 8 Image toolkits provide an abstraction layer over the most common operations used for manipulating images. By default, Drupal uses the GD image management library that is included with PHP. However, it also offers the ability to switch to a different library if needed by using the `ImageToolkit` plugins.



For instance, a contributed module could implement the `ImageMagick` library for developers who needed support for additional image types such as TIFF, which GD does not support. However, only one library can be used at a time as it needs to be configured site-wide.

Programmatically manipulating images using a toolkit involves instantiating an `ImageInterface` object that wraps an image file. This interface (implemented by the `Image` class) contains all the needed methods for applying the common manipulations to images, as well as saving the resulting image to the file system. And to get our hands on such an object, we use the `ImageFactory` service:

```
$factory =...
```

Summary

We are closing this chapter after covering a lot of different topics that have to do with working with files in Drupal 8.

We started with a couple of introductory sections in which we introduced some general concepts such as the various file systems (storages) that Drupal 8 uses, as well the how stream wrappers come into play for working with them. We also introduced the different ways to work with files--*managed* vs *unmanaged*.

Next, we dove into working with *managed files* and created an image field on our Product entity type so that we could import images into it. The other example of working with *managed* files had us create a new Product importer based on a CSV file of data and we also saw how to upload, read and process such a file, as well as manually track its usage. As a parenthesis, we introduced a very powerful feature of Drupal 8 that allows us to hook into the entity CRUD operations and perform actions whenever these are fired. This is a majorly important technique module developers typically use in Drupal.

We then switched gears and implemented our own stream wrapper to serve our imaginary remote API that stored...

Chapter 17. Automated Testing

Automated testing is a process by which we rely on special software to continuously run pre-defined tests that verify the integrity of our application. To this end, automated tests are a collection of steps that cover the functionality of an application and compare triggered outcomes to the expected ones.

Manual testing and review is a great way to ensure that a piece of written functionality works as expected. The main problem encountered by most adopters of this strategy and those who use it exclusively is a regression. Once a piece of functionality is tested, the only way they can guarantee regressions (or bugs) were not introduced by another piece of functionality is by retesting it. Also, as the application grows, it becomes impossible to handle. This is where automated tests come in.

Automated testing uses special software that has an API which allows us to automate the steps involved with testing the functionality. This means that we can rely on machines to run these tests as many times as we want, and the only thing stopping us from having a fully-working application is the lack of proper...

Testing methodologies in Drupal 8

Like many other development aspects, automated testing has been greatly improved in Drupal 8. In the previous version, the testing framework was a custom one built specifically for testing Drupal applications--*Simpletest*. Its main testing capability focused on functional testing with a strong emphasis on user interaction with a pseudo-browser. However, it was quite strong and allowed a wide-range of functionality to be tested.

Drupal 8 development started with *Simpletest* as well, lots of older tests still using this framework. However, with the adoption of PHPUnit, Drupal is moving away from it and is in the process of deprecating it. To replace it, there is a host of different types of tests--all run by PHPUnit--that can cover more testing methodologies. So let's see what these are.

Drupal 8 comes with the following types of testing:

- Simpletest: Existing for legacy reasons but no longer used to create new tests. This will most likely be removed in Drupal 9.
- Unit: Low-level class testing with minimal dependencies (usually mocked).
- Kernel: Functional testing with the kernel bootstrapped, access to the...

PHPUnit

Apart from the old and more or less deprecated *Simpletest*, Drupal 8 uses PHPUnit as the testing framework for all types of tests. In this section, we will see how we can work with it to run tests.

Note

On your development environment (or wherever you want to run the tests), make sure you have the composer dependencies installed with the `--dev` flag. This will include PHPUnit. Keep in mind not to ever do this on your production environment as you can compromise the security of your application.

Although Drupal has a UI interface for running tests, PHPUnit is not well integrated with this. So, it's recommended that we should run them using the command line instead. However, it's actually very easy to do so. To run the entire test suite (of a certain type), we have to navigate to the Drupal core folder:

```
cd core
```

Run the following command:

```
../vendor/bin/phpunit --testsuite=unit
```

This command goes back a folder through the vendor directory and uses the installed `phpunit` executable. As an option, in the preceding example, we have specified that we only want to run unit tests. Omitting that we would run all types of tests, however, for...

Registering tests

There are certain commonalities between the various test suite types regarding what we need to do in order for Drupal (and PHPUnit) to be able to discover and run them.

First, we have the directory placement where the test classes should go in. The pattern is this--`tests/src/[suite_type]` where `[suite_type]` is a name of the test suite type this test should be. Also, it can be one of the following:

```
Unit
Kernel
Functional
FunctionalJavascript
```

So, for example, unit tests would go inside the `tests/src/Unit` folder of our module.

Second, the test classes need to respect a namespace structure as well:

```
namespace Drupal\Tests\[module_name]\[suite_type]
```

This is also pretty straightforward to understand.

Third, there is a certain metadata that we need to have in the test class PHPDoc. Every class must have a summary line describing what the test class is for. Only classes that use the `@coversDefaultClass` attribute can omit the summary line. Moreover, all test classes must have the `@group` PHPDoc annotation indicating the group they are part of. This is how PHPUnit can run tests that belong to certain groups only.

So now that we know...

Unit tests

As briefly mentioned in the beginning, unit tests are used for testing single *units* that make up the code architecture. In practice, this means testing individual classes, especially the methods they contain and what they should be doing. Since the testing happens at such low level, they are by far the fastest tests that can be run.

The logic behind unit tests is quite simple--after providing input, the test asserts that the method output is correct. Typically, the more *input -> output* scenarios it covers, the more stable the tested code is. For example, tests should also cover unexpected scenarios as well as exercise all the code contained in the tested methods (such as forks created by *if/else* statements).

The programming pattern of dependency injection--objects should receive as dependency other objects they might need--becomes critical when it comes to unit testing. The reason is that if class methods work with the global scope or instantiate other objects, we can no longer test them cleanly. Instead, if they require dependencies, we can *mock* them and pass these within the context of the executed tests. We will see...

Kernel tests

Kernel tests are the immediate higher level of testing methodology we can have in Drupal 8 and are actually integration tests which focus on testing various components. They are faster than regular Functional tests as they don't do a full Drupal install, but use an in-memory pseudo installation that is much faster to bootstrap. For this reason they also don't handle any browser interactions and don't install any modules automatically.

Apart from the code itself, Kernel tests also work with the database and allow us to load the modules that we need for running the test. However, unlike the Functional tests we will see next, Kernel tests also require us to manually trigger the installation of any database schemas we need. But we will see how we can do this in the two examples we cover in this section.

Before we can work with Kernel tests though, we need to make sure we have a connection to the database and PHPUnit is aware of this. Inside the `core` folder of our Drupal installation we find a `phpunit.xml.dist` file which we need to duplicate and rename to `phpunit.xml`. This is the PHPUnit configuration file. Normally this...

Functional tests

In the previous section, we looked at Kernel tests and said that they are basically integration tests which focus on components rather than interactions with the browser. In this section, we'll go one level up and talk about the fully-fledged Functional tests, otherwise called browser tests (from the name of the base class we need to extend).

Functional tests in Drupal 8 use a simulated browser (using the popular Mink emulator) that allows users to click links, navigate to pages, work with forms and make assertions regarding HTML elements on the page. What they don't allow us is to test JavaScript based interactions (see next section for those).

In Drupal 7, Functional tests were the most common type of tests used, most classes extending from Simpletest's `WebTestBase` class. But in Drupal 8 we have the `BrowserTestBase` class which is integrated with PHPUnit like the ones we've seen before. And the base class contains loads of methods both for asserting things and shortcuts to performing Drupal (and web) related tasks--creating users, entities, navigating to pages, filling in and submitting forms, logging in, and so...

Functional JavaScript tests

The last type of tests we can write in Drupal 8 is the JavaScript-powered Functional tests. FunctionalJavascript tests are useful when we want to test more dynamic client-side functionality such as JavaScript behaviors or Ajax interactions.

They are an extension of the regular Functional tests but which use the Phantom.js driver instead of Goutte for Mink in emulating the browser. This also means that in order to run these tests we need to install Phantom.js.

Installing Phantom.js is very simple. We have to go to the website (http://phantomjs.org/download.html) and download the package somewhere onto our system. Since we are using a Linux system so we can pull the archive into our home folder (the following link may change for you depending on the version):

```
wget https://bitbucket.org/ariya/phantomjs/downloads/phantomjs-2.1.1-linux-x86_64.tar.bz2
```

Then we can *untar* the archive:

```
tar xjf phantomjs-2.1.1-linux-x86_64.tar.bz2
```

This will unpack the archive into the current folder. And that is it.

To run Phantom.js, we need to run the executable with the following command (from within the Drupal `core` folder):

Summary

In this chapter, we talked a bit about automated testing in Drupal 8. We started with an introduction about why it's useful and actually important to write automated tests, and then briefly covered also a few of the more popular types of software development testing methodologies.

Drupal 8 comes with advantages in this field over its predecessor by integrating with the PHPUnit framework for all the different types of testing it does. And there is a capability for quite a lot of methodologies as we've seen exemplified. We have unit tests--the lowest level form of testing that focuses on single architectural units and which are by far the fastest running tests of them all. Then we have Kernel tests which are integration tests focusing on lower level components and their interactions. Next, we have Functional tests, which are higher level tests that focus on interactions with the browser. And finally, we have the FunctionalJavascript tests which extend on the latter and bring in `Phantom.js` to allow for the testing of functionalities that depend on JavaScript.

We've also seen that all these different types of tests are...

Chapter 18. Drupal 8 Security

Writing secure code is an important aspect of any web application. Preventing ever-so-creative hacking techniques can be really daunting, and this is partly the reason why we as developers sometimes choose a well-established framework with solid and up-to-date security measures baked right in.

Drupal is a CMS that takes security very seriously. The community has a dedicated security team that is always on the lookout for vulnerabilities and advises core contributors and module developers on ways to fix potential vectors of attack. It is also responsible for the fast mitigation of any such issue and disseminating the correct information to the affected parties.

When it comes to out-of-the-box installation, Drupal 8 has come a long way in addressing many security concerns present in previous versions, to the point where much of what Drupal 7 developers had to worry about can now be taken for granted. For this reason, in this annex, we will talk about some of the most prominent security features that Drupal 8 comes with out-of-the-box and that are directly related to our work as module developers....

Cross-Site Scripting (XSS)

Drupal 7 was not inherently vulnerable to XSS attacks, but made it easy for novice developers to open such vulnerabilities. The PHP-based templating system, in particular, made it easy for developers to forget to properly sanitize user input and any other kind of data before outputting it. Moreover, it allowed novice developers to perform all kinds of business logic directly in the template. Apart from not keeping a separation of concerns (business logic vs presentation), this also meant that third-party themes were much more difficult to validate and could easily include security holes.

Most of these concerns have been addressed in Drupal 8, in principle with the adoption of Twig as the templating system. There are two main consequences of this adoption. The first one addresses the need for separating presentation from business logic. In other words, themers and developers can no longer directly access Drupal's APIs nor can they run SQL queries from templates. To expose any such functionality, Twig extensions and filters can be used, but they require the logic to be encapsulated inside a module.

The...

SQL Injection

SQL Injection still remains a very popular vector attack on unsuspecting vulnerable applications that incorrectly make use of database drivers. Luckily, using the Drupal 8 database abstraction layer, we go a long way toward ensuring protection against such vulnerabilities. All we have to do is use it correctly.

When it comes to Entity queries, there isn't much we can do wrong. However, when using the Database API directly as we did in Chapter 8, *The Database API*, we have to pay attention.

Most of the time, vulnerabilities have to do with improper placeholder management. For example, we should never do things like this:

```
$database->query('SELECT column FROM {table} t WHERE t.name = ' . $variable);
```

This is regardless of what `$variable` is--direct user input or otherwise. Because by using that direct concatenation, malicious users may inject their own instructions and complete the statement in a different way than intended. Instead, we should use code like we did in Chapter 8, *The Database API*:

```
$database->query("SELECT column FROM {table} t WHERE t.name = :name", [':name' => $variable]);
```

In other words, use placeholders which...

Cross-Site Request Forgery (CSRF)

CSRF attacks are another popular way applications can be overtaken, by forcing a user with elevated privileges to execute unwanted actions on their own site. Usually, this happens when certain URLs on the application trigger a process simply by being accessed through the browser (and being authenticated)--for example, deleting a resource.

The most important thing to consider in this respect is to never have such actions happening simply by accessing a URL. To help with this, we have the powerful Form API, which already had token-based CSRF protection embedded from previous versions of Drupal. So basically you can create forms whose submit handlers perform the potentially damaging actions (as we learned in Chapter 2, *Creating Your First Module*) or even add a second layer using a confirmation form (as we saw in Chapter 6, *Data Modeling and Storage*, and Chapter 7, *Your Own Custom Entity and Plugin Types*, when talking about entities). The latter is actually recommended for when the action is irreversible or has greater implications.

Although the Form API should account for most use cases, we may also...

Summary

Drupal 8 has come a long way with locking down its APIs to attack vulnerabilities. Of course, this does not mean it's perfect nor that a bad developer cannot create security holes. For this reason, it's extremely important to pay attention to the security implications of all the code you write, follow the standards (including the OWASP checklist), and be aware of what contributed modules you use (to at least be covered by the Drupal security team). Moreover, it's also very important to keep up to date with security announcements from the Drupal security team, as new vulnerabilities may be discovered and updates required to remedy them. These are more time-sensitive in some cases than others, but it's always good to stay up to date as quickly as possible (by following the communication from the Drupal security team). Luckily, though, historically speaking, Drupal has not had many security crises--at least not compared to other open source frameworks out there. So, from a security standpoint, it has a good reputation. However, do not take this to mean that you as a module developer are unburdened by the heavy responsibility...