Acquia

Introduction to Drupal Module Development

Drupal Module Development

Created By: Acquia Learning Services

Maintained By: Rod Martin, Drupal Trainer - Tyto Learning

Last Updated: January 2025

Published by Acquia © 2020
Acquia, Inc. 53 State Street, Boston, MA 02109

Acquia Training Feedback
We're always improving these materials and keeping them up to date. The modules are often updated in between publishing dates. You may find some differences in the user interface, especially discrepancies in screenshots. We appreciate your feedback and will work as quickly as possible to incorporate any corrections. Please complete our training survey here: bit.ly/AcquiaTraining using the authorization code: "Drupal".

About Acquia
Acquia is the open source digital experience company that empowers the world's most ambitious brands to embrace innovation and create customer moments that matter. Acquia helps organizations create great web experiences using Drupal. Co-founded by Drupal's creator in 2007, its customers include Princess Cruises, Wendy's, LuLuLemon, City of Boston, ConAgra, Whole Foods, Johnson & Johnson, Panasonic, Warner Music, Stanford University, New York MTA, and Al Jazeera. As your enterprise guide, Acquia leverages Drupal's power while simplifying its deployment.

Acquia helps you...

- **BUILD** Drupal sites. Acquia Professional Services pairs customer projects with Drupal experts. Try building your own site for free here: https://insight.acquia.com/free/register
- **HOST** Drupal sites. Acquia Cloud provides flexible cloud hosting tuned for Drupal performance. Dev Cloud, specifically for professional developers, features drag and drop deployment. Learn more at https://www.acquia.com/products-services/acquia-cloud
- **MANAGE** Drupal sites. Acquia Remote Administration is a service that manages Drupal sites.
- **LEARN** to improve your Drupal skills. Acquia's training program is always expanding! What courses would you like to see us offer? Review our catalog on Acquia Academy by visiting https://customers.acquiaacademy.com or share your thoughts about our training offerings by emailing training@acquia.com.
- **PARTNER** with us. We have an extensive range of partners with expertise in a variety of sectors and technologies. View them at http://acquia.com/partners
- **FIND** out more at http://acquia.com/

Introductionw

Course description

Module Development introduces the coding concepts behind Drupal and prepares students to tackle feature development beyond what is available through core and contributed modules. This platform is built in PHP and utilizes Symfony components to extend its accessibility to non-Drupal programmers.

If you know how to configure a Drupal site, but you've found limitations with what you can do with core and contributed modules; you are part of a new Drupal implementation team, or you will be taking over on-going maintenance of a site built by a third-party development group - this course will prepare you to manage any customizations required.

This training serves as the final segment of our Drupal Immersion training, and complements our Drupal Site Builder and Layout/Theming training.

Target audience

The target audience for this course are:

- Drupal developers new to module development (Learners).
- PHP or other programmers new to Drupal (Newcomers).

Prerequisites

To get the most out of this course, you should have prior experience in site building with Drupal. A combination of Acquia's Introduction to Drupal (or What's New in Drupal) and Drupal Site Building are ideal preparation for those with no prior Drupal experience. You do have experience with programming. While you don't need to be an expert in PHP, you understand the basic constructs of programming enough to work with a new language.

- Drupal related site configuration
  - Content creation and maintenance
  - Site building.
- Familiarity with writing code in a text editor.

- Programming experience with PHP is preferred, but any programming experience will be helpful.

Course Learning Objectives

By the end of the course, the users should be able to:

- Describe Drupal's hook system and increasingly object-based development architecture
- Articulate the construction of Drupal modules
- Insert and extract data using Drupal's core database abstraction layer.
- Employ and articulate best practices, conventions and coding standards.

Breadcrumbs: This text uses a "breadcrumbs" method when instructing you to navigate to a page. For example, if you are told to go to **Content** > **+ Add Content**, you should search for the link called "Content", click on it, and once that page loads, search for "Add Content" and click on that.

Table of Contents

Introductionw **3**
Course description 3
Target audience 3
Prerequisites 3
Course Learning Objectives 4
Table of Contents **5**
Course Prerequisites **7**
Knowledge Prerequisites 7
1.0 Drupal Module Development Fundamentals **9**
1.1 About the Drupal Framework 9
1.2 The Page Call Process 10
1.3 Object Oriented Programming (OOP) 12
1.4 MVC, Controllers, and Routing 20
1.5 Services and the Service Container 25
1.6 Hooks and Plugins in Drupal 27
1.7 Drupal Formatting & Coding Standards 29
1.8 Summary 34
2.0 Creating Your First Module **36**
2.1 Module Naming, Location, & Basic Requirements 36
2.2 Create mymodule 41
2.3 Implementing a Hook in My Module 42
2.4 Add Pages and Menu Items 46
2.5 Summary 49
3.0 Building a Fully Functional RSVP List Module **50**
3.1 RSVP List: Module Requirements 50
3.2 RSVP List: Create a Form 53
3.3 RSVP List: The Install File 61
3.4 RSVP List: Database Integration 1 67
3.5 RSVP List: Permissions 69
3.6 RSVP List: Create the Block 72

3.7 RSVP List: Build Administrative Settings Page 76
3.8 RSVP List: Reporting Results 82
3.9 RSVP List: Altering the Node Edit Form 86
3.10 RSVP List: Database Integration II 88
3.11 RSVP List: Conditional Display of RSVP Block 94
4.0 Drush and Composer 95
4.1 Different Tools for Differing Needs 95
4.2 Downloading and Installing Drush 96
4.3 Using Drush 96
4.3 Using Composer 97
4.4 Summary 97
Appendix: How Would You Do That In Drupal? 98
Real-Life Design Considerations 98
Summary 100
Acknowledgements 101
Writing and Editing Credits 101

Course Prerequisites

Knowledge Prerequisites

This course is intended for users familiar with Drupal's essential features. You've already been through a site building course. You have some experience with programming languages, and you want to know how to create a custom module.

This course assumes you can:

- Navigate the administration area to manage essential tasks.
- Set up and configure a basic Drupal site.
- Install and configure modules.
- Work in code to develop web functionality in PHP or another language.
- Understand Procedural and Object-Oriented Programming.

Setting Up a Drupal Learning Environment

Before you begin the following exercises, you will need to have a working Drupal website. You should have received a set of instructions when you enrolled in this course.

Composer

Drupal uses Composer as a dependency management tool. It is recommended to install and use Composer for additional modules and themes for Drupal (we will cover these in different sessions of the course.)

If you are using the method provided by your trainer, composer is already available. If you are using your own environment, you can go to https://getcomposer.org and follow the documentation specific to your Operating System and environment to install composer on your machine.

Lorem Ipsum Generation

Since this course is fast paced, you may not have time to create new content on the fly during exercises.

Website-Based Lorem Ipsum Generators

You can visit http://www.lipsum.com/ to generate lorem ipsum if needed. Copy and paste from the website to wherever you need it in Drupal.

Review Other Websites

It's a good idea to review other websites to get an idea of what you'd like to learn to do, prior to class. Businesses of all kinds use Drupal! You can review completed sites by industry vertical, region, and category by visiting the Drupal Case Study website: https://www.drupal.org/case-studies.

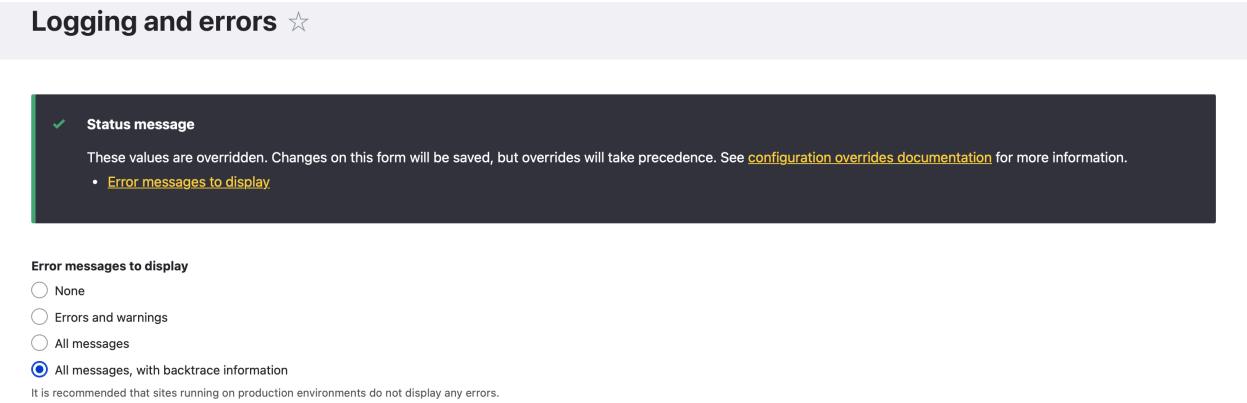
Exercise: Turn on Verbose Error Messaging

When your site encounters an error and you get the "white screen of death", the typical approach is to click the back button in your browser and click on Reports -> Recent Log Messages.

However, this doesn't always work! During development it's a great idea to turn on verbose error messaging. You'll see the errors printed on the screen right away!

Steps:

1. Go to: Configuration -> Development -> Logging and Errors.
2. Put a checkmark in **All messages, with backtrace information**.
3. Click Save



1.0 Drupal Module Development Fundamentals

Drupal is a modular, open-source web content management framework that ships with basic functionality in the form of core modules.

For the most part, additional functionality is added by enabling third-party modules (known as contributed modules) that can be downloaded from the contributed modules section of Drupal.org: http://drupal.org/project/modules.

1.1 About the Drupal Framework

Although Drupal does quite a bit out of the box, the richness of Drupal applications comes from the modules you can add to the core platform to extend its capabilities. In this way, Drupal is as much a "framework" as it is a CMS platform.

What Are Modules?

A module is a set of folders and files that contain PHP code, YAML configuration files to add new or alter existing functionality, and, optionally, Twig templates so your module can be themed with HTML and CSS. When a site builder enables the module via the modules page, the Drupal system executes required code from the module as per the given scenario. For example, if you have added hook_form_alter, then Drupal will read your hook code on every form render.

Modules That You Already Use

If you're already experienced with Drupal site building, you might already have a few modules you like to use.

For example, the Devel module is popular with Site Builders and Themers, as it allows you to generate a lot of placeholder content quickly.

As Drupal matures, the number of popular modules will expand to fill new niches. Perhaps yours will be one of them!

What Do You Want Drupal to Do?

If no modules exist that can extend Drupal in the way you need to extend it--for example, perhaps you need to import data from an external data source, or you need to

integrate Drupal with functionality from a third-party website--you can write a new module to handle this.

It's best to have a very good idea of exactly what you want a module to do before you start creating one yourself.

1.2 The Page Call Process

Drupal utilizes an architecture where the index.php file acts as a *Front Controller*. The front controller's ultimate duty is to take a request object--such as a path to a particular page in the site, or another HTTP request and produce a response object.

Of course, there are a few more steps to it than that!



Looking in index.php

If you open up index.php to look at the code, you'll see there are two main classes being utilized. Symfony's HttpFoundation\Request and DrupalKernel. DrupalKernel was originally based on Symfony's HttpKernel, but was modified by the Drupal team to add some Drupal-specific functionality.

DrupalKernel is basically what boots Drupal into a usable state. So when you land on the index.php page of a Drupal site, Drupal does everything it needs to make sure it's good to go. Among those checks, it makes sure it's properly installed, has any PHP configuration it needs completed, registers the site path, the namespaces of any enabled modules, makes sure any services (including things like the database, or services contributed by modules) are ready, and once it's content with the environment it has, it processes your request.

Router

Routing is an integral part of the page request process. When a site visitor goes to www.example.com/about on a Drupal site, Drupal will reference all the routes registered and then send the request over to the controller or form associated with the route that fits the pattern.

Controller

The controller or form will execute and do any database lookups it needs, and it will return with a render array or response object. (A lot of the time this is HTML, but it can also be JSON or another format if necessary.)

View

Drupal will render the render array or response object in the theming layer, where Drupal pulls in Twig templates, CSS, etc. to produce the HTML for a complete page.

Response

Once Drupal has created the page, Symfony's HttpFoundation\Request component sends the result back to the site visitor, and assuming they were looking for a page exists, the /about page they were looking for shows up in their browser. (Otherwise they might see something like 404 Not Found, or 403 Forbidden).

The Symfony Framework

Drupal incorporates portions of the Symfony 6 framework. This allows PHP developers already familiar with Symfony to pick up Drupal programming skills more quickly. So, if you're a Symfony developer, you already know a lot about how Drupal's event system works, what a service is, how it relates to a dependency injection container and the very root of how Drupal's request-response workflow works.

Other Libraries

Drupal also utilizes a number of other libraries. Composer is used to help autoload classes, files, namespaces, and such. Twig is used for templating and theming, PHPUnit for unit testing. Visit the /vendor/ folder in your Drupal installation to see what is being used on the framework side of things.

For other libraries available for you to use in themes and modules, also see /core/assets/vendor. This is where assets like jQuery and Normalize.css appear.

1.3 Object Oriented Programming (OOP)

OOP, PHP, and Drupal

Drupal was developed to adopt more Object-Oriented Programming (OOP). While earlier versions of PHP supported procedural programming only, more recent versions of PHP allow OOP styles to be adopted. Drupal takes advantage of that flexibility so that you'll see both procedural (functions) and OOP (classes/methods) used in module development.

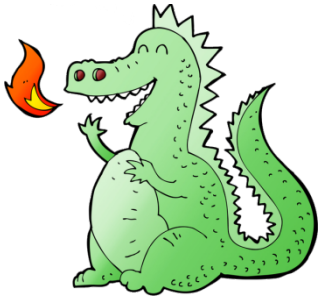
What is Object-Oriented Programming?

Where procedural coding starts with actions or functions before data, OOP starts with data before functions. It is based on the idea of having your code organized into *objects*. Objects have *properties* or fields which encode data related to the object, and they have *methods* which describe what actions are possible to take with the object.

This differs from procedural programming, because it forces you to think about the problem you're trying to solve and create abstractions of the data you're interested in manipulating. Since you're forced to examine your data so that you can create classes from it, you'll naturally gather more requirements at an earlier stage, theoretically reducing backtracking and reworking later on. OOP also helps you encapsulate different parts of your program, which reduces the chance that a method is used on an object or variable that it shouldn't be able to affect.

An Example

Say you have a *dragon* class that is used to generate dragon objects. What properties do you think a Dragon has? And what types of actions (methods) might a Dragon take?



You might decide that Dragons need to have a size, a color, a number of limbs (two, four, or six), and an age. These are all *properties* of dragons. Additionally, you want to specify the actions that any dragon can take: all dragons should be able to burnThings(), eat(), hoardGold(), and fly(). These actions are called *methods* in OOP.

But what happens if you create another class for *pigs*? And generate a pig object from it?

Well, because a pig is not a dragon...pigs can't fly. But you might have a use-case scenario where the dragon takes an input of $pig, and makesFly($pig). Or maybe it just eats($pig). It's up to you to define how objects interact with one another.



Puns aside, let's take a closer look at some of the aspects of Object Oriented Programming that Drupal utilizes.

Classes

A *class* is essentially a blueprint for an object, sometimes called a data type definition. You can assign its *properties* and *methods*. Once you *instantiate* (create an instance of) a class, you get an *object*.

For example, if you have a zoo program, you might make an *Animal class* to populate the zoo with Animal objects. Example syntax for an *Animal* class and some methods might look like this:

```
class Animal {
  public $name;
  public $age;
  public function eat(){
    //code to eat
  }
  public function sleep(){
    //code to sleep
  }
  public function move(){
    //code to move
  }
}
```

You may hear *class* and *object* being used interchangeably when casually talking or reading about OOP. The difference is that the *class* is the blueprint, and the *object* is instantiated (built, or created) based on the instructions contained in the *class*.

Class Properties

In OOP, the variables a class uses are known as the *properties* or fields. They work similarly to other variables; except they are bound to the object that defines them.

In Drupal, you should always indicate whether a property is *public*, *private*, or *protected*. This enforces appropriate encapsulation. Public properties are visible anywhere while private are only meant to be visible to methods of the same class.

If you use var_dump() on an object, it will recursively explore the structure and return the properties of the object in a structured way, including public, private, and protected properties.

Class Methods

In OOP, the *functions* that a class has are known as *methods*. Any actions you need your object to perform will be defined in a *method*.

As with properties, you should always indicate whether a method is *public, private*, or *protected*.

When working with a *method* inside a *class*, you can use $this to reference the object that the method is currently working with. In this example, which is a part of a book module controller, you can see $this was used in the ***return*** statement below:

```
 public function bookRender() {
    $book_list = [];
    foreach ($this->bookManager->getAllBooks() as $book) {
      $book_list[] = Link::fromTextAndUrl($book['title'],
$book['url']);
    }
    return [
      '#theme' => 'item_list',
      '#items' => $book_list,
      '#cache' => [
        'tags' => $this->entityTypeManager()->getDefinition('node')-
>getListCacheTags(),
      ],
    ];
  }
```

Magic Methods

PHP has a number of *magic methods*, indicated by two proceeding underscores and a specific keyword, that allow you to tap into some useful pre-existing functionality for your classes.

A few will be mentioned below, but for a full list visit this link:

http://php.net/manual/en/language.oop5.magic.php

Constructor Methods

Constructor methods are a special type of *magic method* that initializes a class and sets its properties. In PHP, you would name it __construct() . (Note that it leads with two underscores.)

Read more about them here:

http://php.net/manual/en/language.oop5.decon.php

Destructor Methods

Destructor methods destroy an object when called. In PHP, you would name this method __destruct(). Destructor methods are automatically called when there are no longer any other references to an object.

However, you can accidentally create a few scenarios where your destructors don't fire even though you think they should. Read comments at the bottom of the page here for details:

http://php.net/manual/en/language.oop5.decon.php

Setter Methods

A setter method will set (write) some property of an object. The syntax to define this is __set().

Getter Methods

A getter method will get (read) some property of an object. The syntax to define this is __get().

Instantiation

As noted earlier, a class is only a blueprint of an object. A blueprint of a house can't actually be lived in, but it can be used to create a house where possible. Likewise, a class is used to create an object, and your object is what you manipulate to get the results you desire.

The creation of an object is called *instantiating* it. Syntax in PHP uses the keyword new.

```
$toothless = new Dragon();
```

Now you'll have a new object called $toothless, *instantiated from the Dragon class*.

See more on PHP.net:

http://php.net/manual/en/language.types.object.php

Extending

You can extend an existing class for your own purposes. This allows you to inherit properties and methods that already exist for the class, then put your own spin on everything else. The class you are extending from is the *parent class*, and the class you create is the *subclass*.

Say that dragons and pigs are both *animals*. (Well, one is, and the other would be if it existed!) Instead of repeating properties and methods that ALL Animals might have in common, you could set up an Animal parent class instead. This parent class can do things like eat(), sleep(), and move().

From there, you would extend it to the Dragon and Pig classes, and specify dragon-specific and pig-specific aspects from there.

The syntax for extending a class is straight-forward:

```
class Dragon extends Animal {
  public function burnThings(){
    //code to burninate
  }
  public function hoardGold(){
    //code to hoard gold
  }
}
```

As you can see, the dragon can now do dragony things. This is in addition to the methods any Animal might do, such as eat(), sleep(), etc. If you were to extend the Animal class for a Pig class, you might add wallow(), etc. and then a Pig object could do Pig things in addition to Animal things.

Also, if you need to change how sleep() works for all Animals, you now only change it once, and the other classes, such as Dragon and Pig will inherit the changes.

Namespaces

In Drupal, you'll always see a namespace declared near the top of your .php files.

```
<?php
namespace Drupal\book\Controller;
```

The purpose of a namespace is to allow a programmer to name their class in a simple, pragmatic way without having to worry that another module or plugin uses the same class name which could cause a conflict with the code.

For Further Study: Namespaces in PHP

Visit the php.net site for more information on namespaces:

http://php.net/manual/en/language.namespaces.basics.php

https://api.drupal.org/api/drupal/core%21core.api.php/group/oo_conventions/8

Interfaces

An object interface acts like an outline; it declares what methods must be implemented but leaves the details of the implementation to the class *implementing* the interface.

All methods in an interface must be public, and you cannot create an object directly from an interface; you must always instantiate an object from a class.

At first glance interfaces look like an unnecessary layer of complexity, but from an organizational standpoint on a large project they allow a project manager to essentially outline what methods MUST be used in a class implementing a specific interface, without micromanaging how the development team actually accomplishes that task.

When creating Drupal modules, you may decide to implement or extend an existing interface for your own purposes.

Implementing Interfaces

You implement an interface with the following syntax:

```
class YourClass implements AnInterface {
  //stuff goes here
}
```

Remember, when you *implement* an interface, you will have to define its required methods. Not doing so will result in an error. See PHP.net for more information:

http://php.net/manual/en/language.oop5.interfaces.php

Factories

In OOP, *factory objects* provide a degree of separation between an object that might be created, and all the different places in your software that it might be used. It allows for polymorphism of objects themselves, depending on what is requested from the factory.

For example, say your Dragons are hired to do jobs. Some might guard castles, others might guard toll bridges. If new opportunities for Dragons pop up, it's much easier for someone to call up the Dragon Factory and ask for a dragon that complies with their specifications than it is for each place to individually figure out how to go about hiring a dragon with the correct traits.

In a coding sense, using factories means if you refactor a class, and the refactoring changes some of the details of implementation, all you have to do is update the class and the factory...you don't have to visit the fifty other places in your code that need to

implement dragons, because they're all calling up the factory, which presumably is rolling out the new and improved Dragons already.

In the context of creating modules for Drupal, best practices will have you interacting with existing factories if you need a new object created, instead of going directly to the class.

Object Oriented Programming Conclusion

This is just a brief overview of some basic concepts of OOP. If you're new to OOP, you will probably need to study OOP more in depth to become fully fluent in it. Luckily, you can leverage Drupal's codebase as well as resources online to do this.

If you need an example to see how OOP concepts are used, you can always look in /core/modules/(some module)/src to see how Drupal itself has handled it. Drupal naming conventions are very transparent, and files are broken up into bite-sized chunks. For modules, you'll typically see files devoted to defining each custom class, for example, and you can open that up to see how the creator accomplished that task.

You should NOT alter any core files directly, but you can always copy the code to your module and make changes to it there.

For Further Study: Object Oriented Programming

Here is a useful resource for learning OOP for PHP:

http://php.net/manual/en/language.oop5.php

http://code.tutsplus.com

1.4 MVC, Controllers, and Routing

While every module created for Drupal will be different, there are certain parts of Drupal that most modules will need to interact with.

In this section, you'll learn a little about the MVC philosophy, as well as how Drupal utilizes controllers and routing to ensure output from your module is served up to a site visitor when needed.

MVC - Model, View, and Controller

Drupal uses an MVC design pattern similar but not identical to Symfony's MVC architecture.

What is "MVC"? MVC stands for Model, View, and Controller, and is a pattern of programming that separates business logic from display logic. **Model** handles the business logic, **View** handles the display logic, and the **Controller** coordinates communication between the two.

The MVC pattern (like OOP) makes it easier to maintain or change the code later on because you won't have to worry about accidentally altering, say, how a result is displayed if all you need to do is tweak the business logic code.

Drupal Routing, Controllers, and the Front Controller

You're probably already aware of how a simple, static HTML page treats a request for a URL. If you go to www.example.com/blog/blog1.html, on a **static** website you'll have a root folder representing www.example.com which is called something like public_html, www, or wwwroot. Within that folder, you'll have another folder called /blog/ and within that a file called blog1.html. Basically, for each directory in the URL there's a corresponding folder in the file system of your webhost, and for every page a corresponding HTML file.

With a CMS like Drupal, you abandon that 1:1 relationship. Instead, as presented in lesson 1.2, the path Drupal is given by the http request is **routed** by Drupal's **front controller** within index.php to the **php controller file** (or a form) specified for the path a site visitor is requesting.

Once a request is routed by the index.php front controller to your module's controller file, your controller returns a response...typically some sort of render array that'll be combined by Drupal with the other information that makes up a page so that the visitor

visiting the path you specified will be shown the information or form that you need them to see for your module.

Controller vs. Front Controller

There's a bit of jargon here that might be confusing to newcomers to Drupal or the MVC. Drupal uses a **front controller** to bootstrap Drupal into a working state and handle incoming routing requests, etc. This front controller is index.php.

The types of **controllers** *you'll* be creating specifically for your module aren't the same as the front controller. *Your* controllers will be much smaller in scope, and can be considered **thin controllers**.

What's in a Controller File?

A controller file contains the code needed to create a render array or Symfony-style request object to send to Drupal so it can combine your module's results with all the other aspects of a Drupal page, including the view and theming layers, so it can be served to a site visitor.

Here's a part of BookController.php, used by Drupal core's Book module:

```
<?php

namespace Drupal\book\Controller;
use Drupal\book\BookExport;
use Drupal\book\BookManagerInterface;
use Drupal\Core\Controller\ControllerBase;
use Drupal\Core\Link;
use Drupal\Core\Render\RendererInterface;
use Drupal\Core\Url;
use Drupal\node\NodeInterface;
use Symfony\Component\DependencyInjection\Container;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
/**
 * Controller routines for book routes.
 */
class BookController extends ControllerBase {
```

You can view the full code for BookController.php by navigating to /core/modules/book/src/Controller/ in your Drupal files.

> *For Further Study: Adding a Basic Controller*
>
> *You can read more about adding a basic controller here:*
>
> *https://www.drupal.org/node/2464199*

Naming Controllers

Controllers in Drupal should be named in the format of ModulenameController.php and should live in the /src/Controller/ folder of its project, as per the PSR-4 class autoloader standard. You will need to create these folders in your module's directory.

Drupal's PSR-4 class autoloader (Composer) won't be able to find your module's classes upon Drupal's bootstrap if you are not following the expected folder hierarchy, so it's important to name these folders correctly, and make sure /Controller/ is inside of /src/.

Drupal was several years in development and some of the core files make use of the PSR-0 standard and the /lib/ folder instead of /src/. Don't use the PSR-0 standard for your custom module; use PSR-4.

Routing Requests to Your Module

One you have created your controller, Drupal needs to know when to use it.

Controller files work hand-in-hand with routing files. Basically, Drupal needs to know when to direct someone to a page for your module, which is what you specify in your module's routing file.

Since your routing file is a YML file, it uses Symfony's YML syntax, where proper indentation and whitespace is meaningful and important. This is different from PHP which doesn't care what your white space looks like - PHP will run regardless if everything else is correct. But YML files missing appropriate indentation or whitespace will result in an error if you're missing even a single space.

Each module that defines routes will have a file named *modulename.routing.yml*. Here is an example of some of the contents of the routing.yml file for the Drupal core Book module:

```
book.render:
  path: '/book'
  defaults:
    _controller: '\Drupal\book\Controller\BookController::bookRender'
    _title: 'Books'
  requirements:
    _permission: 'access content'
```

BookController.php, the controller file for the Book module contains methods that correspond to the route shown here.

Breaking a Routing File Down

For each route you create, the **path:**, **defaults:**, and **requirements:** sections are required. You *always* need to specify them. You can also specify **options:** which is (you guessed it) optional.

Commonly Used Route Keys

modulename.routename	The name or ID of a route. Settings that are a part of this route need to be indented beneath it, or they will not be properly recognized. Required.
path:	The URL to the route, with a leading forward slash.Needs to be indented by two spaces. Required.
defaults:	Defines the default properties of a route. Should include one of the following to specify how the output is generated. This line has the same indentation as path. Required.
_controller	Is an option under defaults. Must be indented by two spaces under defaults. When using _controller, the value is callable (classname::method) returning a renderable array or a response object.
_form	Is an option under defaults. You'll format this similarly to _controller (using namespaces) but instead of your namespace referring to a controller, you'll refer to a form in /src/Form/ instead.
_title	Is an option under defaults. This sets the title of the page your site visitor sees when they go to this route.

requirements:	Requirements basically are used to tell Drupal if the visitor trying to access the route should be allowed to the route at all or only if the request is for a specific format like JSON. You can set your requirements to whatever they need to be for your module. This line has the same indentation as path. Required.
_permission	Is an option under requirements. Sets the name of the permission if you need Drupal to restrict access.
_scheme	Is an option under requirements. Set to https or http. If set, urls will have this scheme set fixed.
_role	Is an option under requirements. Sets the role required for access.

The above is not a comprehensive listing of all that can be done or specified in a module's .routing.yml file. For more info, visit these links:

- https://www.drupal.org/developing/api/8/routing
- https://www.drupal.org/node/2092643

Namespaces & Routing YML Files

Note how the value specified in the .routing.yml file for a Controller or a Form isn't the actual file location. You might also notice these values use a backslash instead of a forward slash.

```
book.settings:
  path: '/admin/structure/book/settings'
  defaults:
    _form: '\Drupal\book\Form\BookSettingsForm'
    _title: 'Books'
  requirements:
    _permission: 'administer site configuration'
```

This is because Drupal uses Namespaces and the PSR-4 class autoloading standard to point to files. Learn more about PSR-4 here: https://www.drupal.org/docs/develop/standards/psr-4-namespaces-and-autoloading-in-drupal-8

For Further Study: Scope Resolution Operator (::)

You might have noticed some of the namespaces contain a double colon. You can read more about how the double-colon works in PHP here:

http://php.net/manual/en/language.oop5.paamayim-nekudotayim.php

1.5 Services and the Service Container

Drupal inherits some of Symfony's methods for handling globally needed objects and calls them services. They are loaded into the service container (aka Dependency Injection Container).

A **service** is simply a class that Drupal will use to do something. For example, there's a service to send email, a service for processing HTTP, a service for the CKEditor plugin, and more. Not all classes or objects are considered services, but those that might be needed across a wide array of contexts and modules are probably best created as services.

What if your module needs to add a service to Drupal? How does Drupal know which services to make available for use? You basically *teach* Drupal by configuring a *.services.yml* file.

If you visit /core/modules/ and look in any of the folders, most of them will have a .services.yml file. Drupal looks for these files when it starts and uses the definitions within them to load the services needed into its Service Container.

A list of all core services can be found at /core/core.services.yml. You can also find a list in the Drupal API documentation here: https://api.drupal.org/api/drupal/services

Service Tags

Some services use tags, which define relationships between a related set of services, or explain how an aspect of the service behaves.

You can learn more about them here: https://api.drupal.org/api/drupal/core%21core.api.php/group/service_tag/9.3.x.

Using an Existing Service in Your Module

There are two ways to implement an existing service in your module. One way is more preferred than the other, as it results in code that's easier to unit-test and maintain.

Preferred Method

The preferred method of accessing services in Drupal is to use *dependency injection*.

There are three types of dependency injection: constructor (using the magic method _construct()), setter (defining a setter method in your class), and property injection which can be done on the properties of a class that are public.

An example of syntax injecting a service into a form can be found here:

https://www.drupal.org/node/2203931

Controllers that are extended from ControllerBase have a create() function that allow you to get any needed services when the controller object is created from your class.

Alternate Method

There is a global static Drupal class you can use if the preferred method of dependency injection is not feasible. For example, .install or .module files are not class files. So we cannot use dependency injection. In such cases, we can use this method.

However, although using the \Drupal class is possible and many tutorials use it, it is not recommended if this way of using services can be avoided.

For more information on Services, visit these pages:

https://api.drupal.org/api/drupal/core!core.api.php/group/container/11.x
https://api.drupal.org/api/drupal/services/11.x

For Further Study: Symfony and Services

Drupal's usage of services is based on Symfony. You can view the Symfony documentation here:

http://symfony.com/doc/current/book/service_container.html

1.6 Hooks and Plugins in Drupal

About Hooks

With the incorporation of many Symfony components into Drupal, and the adoption of OOP, there was a major shift away from hooks in cases where more modern PHP architectural decisions could be implemented. Plugins, annotations and the Symfony Event Dispatcher have replaced many of the hooks you would have commonly invoked in previous versions of Drupal, and you use other ways to configure these things, such as creating various .yml configuration files. However, Drupal still depends on hooks for many key functions, and many phases of Drupal's page building process can still be intercepted, and the data modified by implementing hooks.

When hooks are in play, at strategic times during page processing Drupal looks for any functions that follow a specific naming pattern and invokes all matching functions.

For example, when rendering a form Drupal looks for anything that implements hook_form_alter(). It then executes any matching functions, gathers the returned data, and displays the requested changes to the final form. To implement hook_form_alter(), all you need to do is declare a function where the hook part of the name is replaced with the module's machine name. So, if your custom module was called *purple*, your form_alter hook function would be named purple_form_alter().

When you define a function in your module according to the predefined hook naming pattern, Drupal will find it and use it at the appropriate time in the page build, allowing your change to go into effect.

Here is a list of the hooks still used in Drupal:

https://api.drupal.org/api/drupal/core!core.api.php/group/hooks/11.x

About Plugins

Plugins are small pieces of functionality that are swappable and can be grouped by type when they perform similar functionality.

The D8 plugin system provides a set of guidelines and reusable code components to allow developers to expose pluggable components within their code and (as needed) support managing these components through the user interface.

Plugins are a little like Services in that they are specific, limited scope and php classes that are swappable, but they have different purposes.

Plugins implement different behaviors via a common interface, while Services provide the same functionality and are interchangeable, differing only in their internal implementation.

The plugin system has three base elements:

- **Plugin Types.** The plugin type is the central controlling class that defines how the plugins of this type will be discovered and instantiated. The type will describe the central purpose of all plugins of that type
- **Plugin Discovery.** Plugin Discovery is the process of finding plugins within the available code base that qualify for use within this particular plugin type's use case.
- **Plugin Factory.** The Factory is responsible for instantiating the specific plugin(s) chosen for a given use case.

There are different types of plugins, most commonly you will use the Block and Field plugins.

Plugins are added to a module project by adding a folder called /Plugin/ to the /src/ folder. All your plugins for a module will be stored in its /*src/Plugin* folder. Subfolders are made in the Plugin folder for each type of Plugin. So, if you were creating a block plugin called FirstBlock.php to add a block to your First Module project, the file structure would look like this:

```
modules/custom/first/src/Plugin/Block/FirstBlock.php
```

Plugin Discovery

There are four different plugin discovery types:

- *StaticDiscovery* allows for direct registration within the class,
- *HookDiscovery* users info hooks to retrieve available plugins,
- *YMLDiscovery* is used for local tasks and local actions and allows plugins to be defined in YML.
- *AnnotatedClassDiscovery* uses special annotations in comments. The annotation lives in the same file as the class that implements the plugin. Annotations allow for complex structured data, and are available to the translation system in Drupal.

In short, plugins are an Object Oriented replacement for info hooks and any hook associated with an info hook, though they provide a more robust mechanism for replacement of logic. With plugins you can usually swap a class for a particular plugin and run completely different code than what core or a contrib module provided.

> *For Further Study: Plugins*
>
> *Plugins API: https://api.drupal.org/api/drupal/core!core.api.php/group/plugin_api/11.x*
>
> *More on why Plugins are used: https://www.drupal.org/node/1637614*
>
> *Explaining plugin discovery: https://www.drupal.org/node/1638040*
>
> *Plugin types (aka managers): https://www.drupal.org/node/1637730*
>
> *More about annotations: https://www.drupal.org/node/1882526*

Annotations syntax comes from the Doctrine project. More about Doctrine:

https://www.doctrine-project.org

1.7 Drupal Formatting & Coding Standards

Certain parts of Drupal are structured to require certain naming standards, and certain syntaxes, to work as expected. Configuration files in YML format also have strict interpretation of whitespace, which you may be unfamiliar with if you typically work with a language, like PHP, where whitespace is irrelevant.

Additionally, as Drupal is a very community-oriented platform, some decisions regarding syntax or style have been adopted to ensure code readability among a wide and diverse group of Drupal developers.

Below are highlights of expected formatting and coding standards you should follow when coding for the Drupal community.

File Structure and Naming

- Each class must be in its own file within the /src/ folder.
- The file should be named after its class. Example: FooInterface -> FooInterface.php
- One class, interface, or trait per file.
- Don't use "Drupal" or "Class" in the name.
- Interfaces should end in Interface.php.
- Test class names should end in Test.php..

Classes and Namespaces

- Classes must use a namespace starting with \Drupal\module_name.
- Classes autoload based on PSR-4 namespacing convention
- The PSR-4 'tree' for modules starts under modulename/src.
- When using a class that's in a different namespace, declare it in the top of your file with use.
- Methods and properties of classes must specify visibility (public, protected, private)
- The PHP 4 style "var" declaration must not be used
- The use of public properties is discouraged, as it can defeat the purpose of OOP.
- Protected or private properties and methods should not use an underscore prefix
- Make sure to use type hinting when a function or method requires conformity to a specific interface
- Do not use a class as the type for type hinting. Specify an interface instead.

See the Drupal API documentation standards for classes and namespaces page below for more detail: https://www.drupal.org/coding-standards/docs#classes

PHP Indenting and Whitespace

- Indent with 2 spaces. Do not use tabs.
- Lines should have no trailing whitespace.
- Lines should not be longer than 80 characters (in general). Use reasonable judgement; sometimes long function names, or various definitions or declarations will need more space.
- Files should use Unix line endings (\n) not Windows line endings (\r\n)
- Files should end in a single newline (\n)
- Binary operators should have a space on either side. Examples:
  - $foo + $bar
  - $foo != $bar

- Unary operators should not have a space between the operator and the number or variable they affect. Example:
  - $this->temporaryNameIndex++;
  - ++$i
- Put a space between the (type) and the $variable when casting a type. Example:
  - (int) $somevariable
- Use "elseif" not "else if" when this type of control structure is needed
- When using a control structure (if, for, while, elseif, switch) have a space between the control keyword and the opening parenthesis. This distinguishes the control structure from a function call.
- Always use curly braces for control structures, even in situations where they are optional.
- Opening curly brace should be on same line as opening statement
- Closing curly brace should be by itself, and indented to the same level as the opener.

```
public function submitForm(array &$form, FormStateInterface
$form_state) {
    // code here
}
```

.YML File Syntax and Standards

Whitespace is very important within .yml files. Spacing and indentation is used to indicate how certain elements are related to one another.

Because .yml configuration files are ubiquitous in Drupal, and are used to help define everything from modules and themes to routes, libraries, and services, it's very important to make sure you have the syntax correct. The first bugs you run into when first creating a new Drupal module will likely be due to spacing errors in your .yml files.

- When indenting, use two spaces for each indentation level
- Do NOT use tabs.
- Some text editors, such as Notepad++, will use tabs by default for indenting. You'll need to change this default so your spaced indents are not "autocorrected" to tabs.
- Refer to the Symfony documentation for more detail: http://symfony.com/doc/current/components/YML/YML_format.html

Twig Syntax

While you likely won't be working as much with Twig templates as you might when creating a new theme, it's possible you will need to create some Twig templates to handle the HTML markup of your custom module.

The alternate control statement syntax using a colon (:) instead of curly braces ({ }) is allowed.

https://www.drupal.org/node/1823416

Documentation Within Code

Drupal uses a module that will detect and parse documentation text added in comment blocks that follow a specific syntax. In-file documentation should be organized into *docblocks* that preface your classes, functions, etc.

Example:

```
/**
 * The summary comes first, on this line.
 *
 * Next paragraph goes here.
 *
 */
```

Code you are documenting goes here. Notice there's no space between this "code" and the docblock above.

Other things to note:

- Comments starting with **//** or with **/*** (with only 1 asterisk) are **not** recognized as docblocks
- Each line starts with an * (aside from the opening line)
- A paragraph break is detonated by a line with only the beginning asterisk, and nothing else.
- The first line is considered the "summary".
- Use standard English grammar when creating comments.

Tags for your Docblock

- For functions, you should document parameters using @param in the docblock.
- If there's a return value, also document that with the @return tag in the docblock.
- If you want to indicate to the API Module that your comment applies to the entire file, put @file before your summary line.
- If you want to inherit documentation from a base class or interface, use {@inheritdoc}
- @var is used to document the type of a class property.
- Visit the API documentation and comment standards page for many, many more tags along with examples of usage. https://www.drupal.org/node/1354
- Note that some examples might be for versions of Drupal other than Drupal. Use your best judgement if you can't find a Drupal example.

Contributed Module Documentation Standards

If you plan to release your module to the Drupal community it's a good idea to make sure your module is well-documented.

- Provide a helpful project page
- Provide a useful README file
- Document functions using Doxygen style comments.
- See API documentation and comment standards: https://www.drupal.org/node/1354
- Also see Module Documentation Guidelines: https://www.drupal.org/node/161085
- API Documentation Samples (including for modules): https://www.drupal.org/node/1918356

> *For Further Study: Drupal Standards*
>
> *You've only seen selected important excerpts above. To keep on top of the standards as they are refined and updated with examples for Drupal, visit these pages:*
>
> *https://www.drupal.org/node/608152*
>
> *https://api.drupal.org/api/drupal/core%21core.api.php/group/oo_conventions/8*
>
> *https://www.drupal.org/coding-standards/docs*

1.8 Summary

- The Drupal Framework allows you to extend its capabilities
- Some Drupal components are based on the Symfony 2 framework, which makes it easier for developers familiar with Symfony to develop for Drupal.
- The class autoloader Composer is used to autoload classes.
- Twig is used for templating.
- The index.php file acts as a Front Controller for Drupal.
- The Front Controller accepts pages requests from site visitors and returns a response object.
- The page call process goes in this order: Request > Router > Controller > View > Response.
- Drupal is written using Object Oriented Programming concepts.

- Drupal architecture uses a Model, View, Controller MVC pattern, similar to but not the same as Symfony 2's architecture.
- Since the business object in the model and the display logic in the view are separate, you will use controllers and a .routing.yml file to pass data generated from your module to Drupal to theme and display.
- Namespaces are used extensively; Drupal uses the PSR-4 standard, and you should store controllers and forms in a /src/ folder.
- Drupal uses a services container and dependency injection to make certain classes available everywhere in Drupal—for example, there is a mail service you can have your module call if you need to send an email.
- You can create a .services.yml file to add a part of your module to Drupal as a service.
- Hooks in Drupal are specialized functions in Drupal that can be called to alter the behavior of a page.
- Plugins allow certain pieces of code to be reused elsewhere.
- Drupal has several coding standards. Some are for readability, others are necessary, so your module is accurately discovered and used by Drupal.
- Errors in the syntax of your .yml files will cause your module to break.
- Errors in the syntax of your file and folder naming will cause Composer, the PSR-4 autoloader, to not correctly discover and load your module and its classes.
- PHP and Twig coding standards help other Drupalists read your code more easily if they have to update or maintain it.

2.0 Creating Your First Module

There's no better way to understand the process of module development than to jump right in and give it a try.

Let's start by taking a look at the requirements to get your project recognized by the Drupal framework, and after that you'll build your first module!

2.1 Module Naming, Location, & Basic Requirements

Here are a few of the initial things you'll need to know when creating modules. You may already know some of these things due to earlier courses or exercises, but it can be helpful to have a reminder.

Naming Conventions

The first step in creating a new module is choosing a descriptive name. Modules have both machine names and human-readable names. For instance, in one of the exercises you'll be doing later on, the **human-readable** name will be *RSVP List* and the **machine** name will be rsvplist.

It's good to go for a short but memorable or descriptive name. This way people can know at a glance what your module does.

There are also a few other naming conventions you should follow. Not following these conventions can mean the difference between Drupal recognizing your project, or not.

Naming Convention	Bad Example	Good Example
Multiple words in module names should be separated with underscores.	great-module	great_module
Module names cannot begin with a number.	2nd_module	module2
All Drupal filenames (as well as function definitions and variable names) must be in lowercase. All non-class based files like .module, .install file names should be in lower case.	Awesome	awesome
Class-based file names should be in UpperCamelCase.	myclass.php	MyClass.php
Variables declared in non-class based files should be in lower case and underscore(_) can be used.	Info.YAML	info.yaml
Variables declared in class-based files, should be in camelCase. e.g.	$Routematch	$routeMatch
Custom modules should not conflict with core modules	aggregator	custom_aggregator

You should check if the name you've chosen is already in use by another module creator--particularly if you intend to contribute your module back to the Drupal community. This is so your module doesn't conflict with other modules.

If you go to www.drupal.org/project/YOURMODULE (replacing YOURMODULE with your chosen machine name in all lowercase), you can see if a page already exists, or if you get a 404 Page Not Found error. If you get a 404, you know your project name is unique to the community.

Folder Hierarchy

For Drupal to find your project, module project folders need to be stored within the /module/ folder. File organization within the module directory is up to you as long as you adhere to the PSR-4 standard when necessary. (For example, controllers will go in /modules/yourmodule/src/Controller.)

Adding modules and modifications outside of the /modules/ directory is considered hacking core. One of the cardinal rules of Drupal is "Never hack core." This is so you have a clear upgrade path if Drupal releases a minor revision--you don't want to lose your modifications in the event of a Drupal core upgrade.

For more information, see:

http://drupal.org/best-practices/do-not-hack-core

During this class you will set up a /custom folder under /modules to store your new custom modules. In later exercises, you will add some files to /src/ and folders within it.

Minimum Required File to Create a Module

Drupal requires one file to be present before it will "recognize" your project as a module.

Having this file present doesn't necessarily mean your custom module will *work*, since that depends entirely on your programming skills, but *lacking* these files means for certain that it won't work.

The .info.yml File

The .info.yml file is one file that Drupal always checks, so you will use it to tell Drupal about your projects and provide pointers so Drupal can find necessary files for your project to run. Here's an example of what a .info.yml file can contain:

```
name: My First Custom Module
type: module
description: This is the first module I've created.
core_version_requirement: ^9.5 || ^10
package: My Custom
dependencies:
  - views
  - book
```

And here's a breakdown of what each line means:

Line	Description
`name:  My First Custom Module`	The human-readable name. Required.
`type: module`	Drupal uses the same .info.yml format for themes as well, so you need to specify this one is for a module, and not a theme. (Required).
`core_version_requirement: ^9.5 || ^10`	New key added by Drupal 9. You can specify if your module works with either Drupal 8 or 9. (Required).
`description: This is the first module I've created.`	A description of what your module does, which users of the module will be able to see on the Modules page. (Optional).
`package: My Custom`	You can specify a new or existing Module category for grouping on the Extend page. If a package is not specified, your module will be displayed in the 'Custom' group. (Optional).

`dependencies:`
 	
`- drupal:views`
 	
`- drupal:book`
	A list of other modules your module depends on. Dependencies on Drupal core or contrib modules should be namespaced in the format {project}:{module}, where {project} is the project name as it appears in the Drupal.org URL (e.g. drupal.org/project/views) and {module} is the module's machine name.
	
	Note: Syntax is VERY important! You MUST indent by two spaces, then have a hyphen and another space before the machine name. Do not use tabs.
	(Optional).

More information about .info files can be found at this page, along with other options you can declare that aren't covered above: https://www.drupal.org/node/2000204.

Other .YML and Module Files

Depending on what your module does, you'll potentially need to create a few more files. Below is a table talking about them briefly; you'll learn more about them later in the next two learning modules.

All files listed below live in the root of your project's folder, and are prefaced before the first dot with the machine name of your module.

.routing.yml	This file is used to define what paths on a Drupal site map to what controllers or forms for your module.
.menu.links.yml	This file is used to define any menu links your module will need.
.services.yml	This file is used to define services your module uses or provides.
.libraries.yml	This file is where you define any asset libraries your module may need. For example, jQuery. Libraries defined here also need to be referenced in the .info.yml file.
.install	This file is where you'd implement hook_uninstall, hook_schema, and other related hooks if they are needed for your module.
.permissions.yml	This file is where you define permissions for your module.
.module	The .module file is a PHP file that typically contains all the implementations of Drupal hooks a module uses. When your module is enabled, Drupal will check your .module on each page load.
composer.json	Composer is the PHP package manager that Drupal uses. You can create a composer file for your module if needed. Learn more about them here: https://www.drupal.org/node/2514612

2.2 Create mymodule

Now that you've learned the minimum requirements you need to make a module, let's put it into practice.

Exercise 2.2.1: Create the Required Module Files

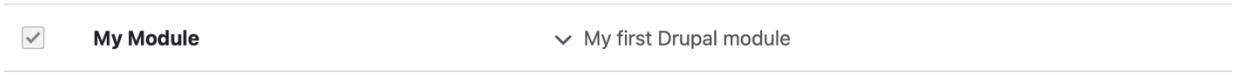
1. Navigate to your code base into the **modules** folder. Create a new folder **/ custom/mymodule** (VSCode will allow you to create sub-folders as well).
2. Right click on the **mymodule** folder and select **New File.**
3. **Name** your new file mymodule.info.yml.

   Note: Always check the breadcrumbs at the top of the editor to ensure the files are in the correct folder.

4. In your newly named and saved file, **add the following key: value pairs**.

```
name: My module
type: module
description: My first Drupal module
core_version_requirement: ^10 || ^11
```

5. **Save** your file.
6. **Visit** the *Extend* page. You should find your My Module project in the list!
7. **Check** the *checkbox* and then **click** *install* to install your module.



You'll see a verification that your module was installed. However, since no code has been added yet, you won't see anything else happen on your Drupal site.

At this point, you have the bare minimum Drupal needs to recognize that a custom module exists and should be loaded.

In the next sections, you'll flesh this module out a bit more.

2.3 Implementing a Hook in My Module

Now that you've made the .module file, something should be added to it so that it actually affects Drupal in some way.

You're going to add a hook that allows you to alter an existing form before the form is rendered. By altering forms, you can add additional form elements, change or remove existing elements, and even change the validation and submission handling of the form.

Because all Drupal forms use the Form API, any module can alter any form. Important forms like the node add/edit form are often altered by other modules.

About hook_form_alter()

Look up hook_form_alter() on api.drupal.org (you'll be able to do this with any hook you are thinking of implementing):

https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Form%21form.api.php/function/hook_form_alter/9.3.x

You'll see there are three parameters:

- $form - Nested array of form elements that comprise the form. &$form is a pointer to the current form. It's considered "passing by reference" - https://www.php.net/manual/en/language.references.pass.php
    - In PHP, arguments to a function can be passed by value or passed by reference. By default, values of actual arguments are passed by value to formal arguments which become local variables inside the function. Hence, modification to these variables doesn't change value of actual argument variable.
    - When arguments are passed by reference, change in value of formal argument is reflected in actual argument variable because the former is a reference to latter. Thus pass by reference mechanism helps in indirectly manipulating data in global space. It also helps in overcoming the fact that a function can return only one variable.
- $form_state - The current state of the form. The arguments that \Drupal::formBuilder()->getForm() was originally called with are available in the array $form_state->getBuildInfo()['args'].
- \Drupal\Core\Form\FormStateInterface – Forms that require a Drupal service or a custom service should access the service using a dependency injection.
- $form_id - String representing the name of the form itself. Typically this is the name of the function that generated the form.

At the bottom of the page, you'll also see some example code. You'll need to grab the first and last lines, change the name from hook_form_alter to mymodule_form_alter, and insert details for what you actually want this function to do for your module.

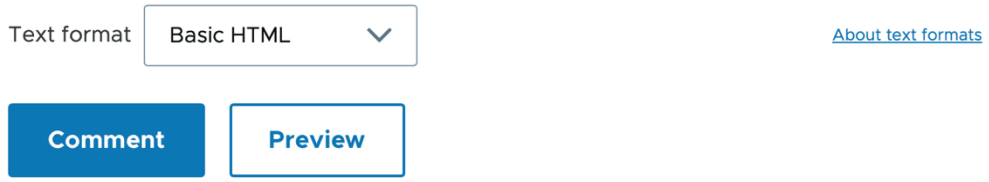
Renaming "hook" to your module's machine name is something you'll need to do each time you implement a hook. Try this out in the next exercise.

Exercise 2.3.1: Change the Comment Form

1. If you don't already have a node of the Article content type, **Create** a Test Article, then scroll to the comment form on the bottom and notice the label on the *Save button*. You will change that label with your module.
2. Now, **create a second new file** in the **mymodule** folder.
3. **Name** your new file **mymodule.module**
4. In it, **type** the following:

```
<?php
/**
 * Implements hook_form_alter().
 */
function mymodule_form_alter(&$form, \Drupal\Core\Form\FormStateInterface
$form_state, $form_id){
  dpm($form_id);
  if ($form_id == 'comment_comment_form') {
    $form['actions']['submit']['#value'] = t('Comment');
  }
}
```

5. The line: dpm($form_id); will print the machine names of any form on the page. "Drupal Print Message" is part of the Devel module, so you'll need to have enabled that on your site.
6. **Save** your file.
7. **Clear Drupal's cache**.
8. Navigate to your Test Article and scroll to the bottom to see the change to the Save button. It should now read as *Comment* instead of *Save*:



An Alternative to dpm:

Instead of using 'dpm' which relies on the devel module you can do the following:

At the top of the file (after the first docblock) add:

```
use Drupal\Core\Messenger\MessengerInterface;
```

then instead of 'dpm', use the built-in Messenger class:

```
\Drupal::messenger()->addMessage(t("form_id of this form is @id: ",
['@id' => $form_id]));
```

This will print: form_id of this form is comment_comment_form:

Troubleshooting Steps:

1. Flush the cache again.
2. If you get a PHP error in the browser, look for syntax issues including missing semicolons, brackets, or commas.
3. If you don't see any effect from your new code or you get a PHP error that mentions something is undefined in a different file, you probably have a misspelling - check spelling throughout your code.

Analyzing Your First Hook

Now that you have created a module, and have seen the hook_form_alter() hook in action, what questions come to mind about how the hook worked?

Here are some things that you might want to think about:

- How does the code you used in this exercise differ from the "Code" section on the hook_form_alter() API documentation page?
- The code section on the API documentation page needs to be modified before it will work in a custom module--it just illustrates possibilities of what CAN be done.
- Take a look at the comments for hook_form_alter(). Is there anything interesting there that will influence how you use hook_form_alter() in the future?

2.4 Add Pages and Menu Items

Next you're going to create a page at a specific path that you define with Drupal's Menu API by creating a controller, and then "wire it up" to Drupal by creating a .routing.yml file that directs Drupal to the controller you made.

Once you have those set up so a page appears at /mymodule, you'll link it to the main menu using a .links.menu.yml file.

Exercise 2.4.1: Add Routing and a Menu Link

1. In your /mymodule folder, **create** a new text file. **Save** the file as **mymodule.routing.yml**. You will use this file to define a path on your site.
2. Add the following six lines of code to **mymodule.routing.yml.**

```
mymodule.content:
  path: /mymodule
  defaults:
    _controller: \Drupal\mymodule\Controller\FirstController::content
    _title: My First Page and Menu Item
  requirements:
    _permission: access content
```

3. **Save** the **mymodule.routing.yml** file.
4. In your mymodule folder, **create** a new folder named src**.**
5. Inside the /src/ folder, **create** a new folder named Controller.

6. Inside the /Controller/ folder, **create** a new text file named **FirstController.php.**
7. **Add** the following code to the *FirstController.php* file:

```
<?php
/**
 * @file
 * Contains \Drupal\mymodule\Controller\FirstController.
 */
 namespace Drupal\mymodule\Controller;
 use Drupal\Core\Controller\ControllerBase;

 class FirstController extends ControllerBase {
   public function content() {
     return array(
       '#markup' => t('This is my menu-linked custom page'),
       );
   }
 }
```

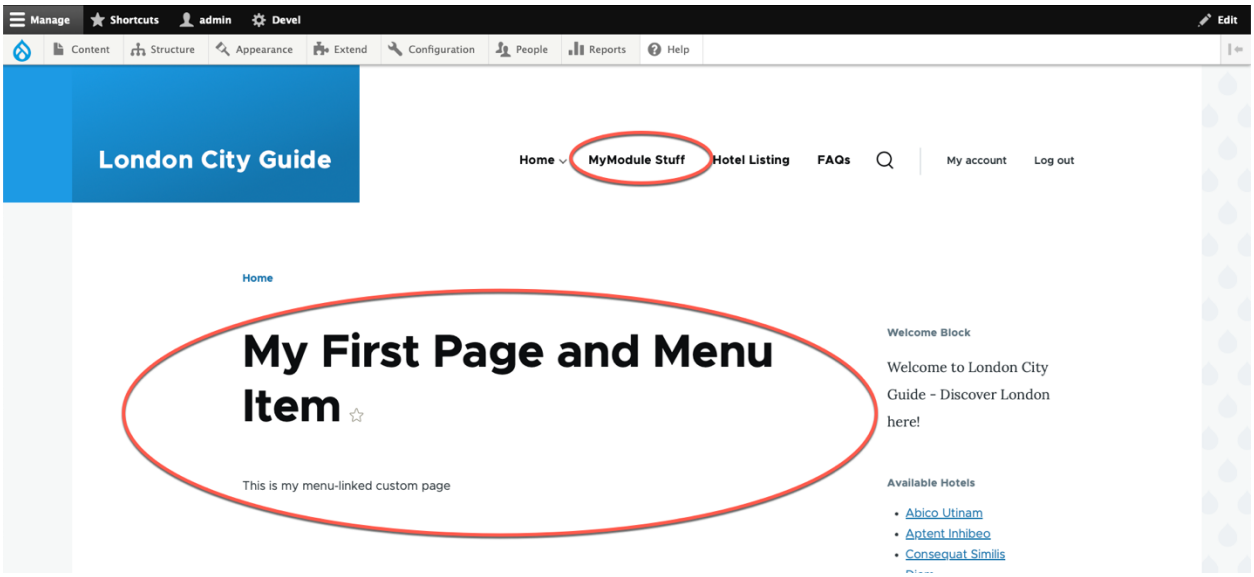
8. **Save** your file.
9. **Clear Drupal's cache** by going to **Configuration** > **Development** > **Performance** > **Clear all caches**
10. **Navigate** to /mymodule and see your new page.
11. Next, you will create a file to define a menu item linked to the new /mymodule page.
12. In your *mymodule* folder, **create** a new text file. Save the file as mymodule.links.menu.yml.
13. **Add** the following code to the mymodule.links.menu.yml file:

```
mymodule.newpage:
  title: MyModule Stuff
  description: Link to the page mymodule created
  route_name: mymodule.content
  weight: -100
  menu_name: main
```

14. **Save** your file.

15. **Clear Drupal's cache** by going to **Configuration** > **Development** > **Performance** > **Clear all caches**

16. You should see a result that looks something like this:



Two Important Reminders: Watch Your White Space and Clear Your Cache!

Any time you're editing a .yml file, watch the white space carefully. No tabs, and indent by two spaces per level. Drupal is very quick to complain or "break" if it doesn't find what it expects in a .yml file of any sort. You can go to Reports > Recent log messages to see details about what exactly it doesn't like in your .yml file.

Also, *always, always, always* clear your cache after a change. You'll get sick of being told to do this, but it's way easier to clear your cache again then it is to hunt and hunt for a change you think you did correctly but isn't applying to your module yet because the cache wasn't updated.

Configuration > Development > Performance > Clear all caches

2.5 Summary

- You learned the essentials of adding a new module to extend or alter Drupal's functionality.
- the project name you specify should be unique, it becomes the machine name for your module
- use the exact machine name to key your folder, file and function names
- your project folder must exist in a modules directory in your codebase
- best practices suggest you create custom and contrib subfolders to organize your projects
- two files are required to be recognized as a module, .info and .module
- .info.yml declares information to Drupal
- .module will hold many of your hook function definitions
- Understanding hooks is one of the keys to creating Drupal modules. Almost every phase of Drupal's page building process can be intercepted, and the data modified.
- Using Drupal's API as a reference helps you understand how to implement the hooks you need.

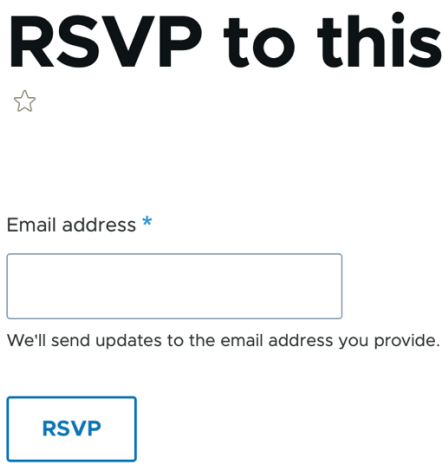
3.0 Building a Fully Functional RSVP List Module

In this learning module, you will focus on building a module called RSVP List. This will involve identifying and interacting with many key parts of the Drupal API to build a fully functional module from end-to-end. This will also give you more practice following Drupal's coding standards.

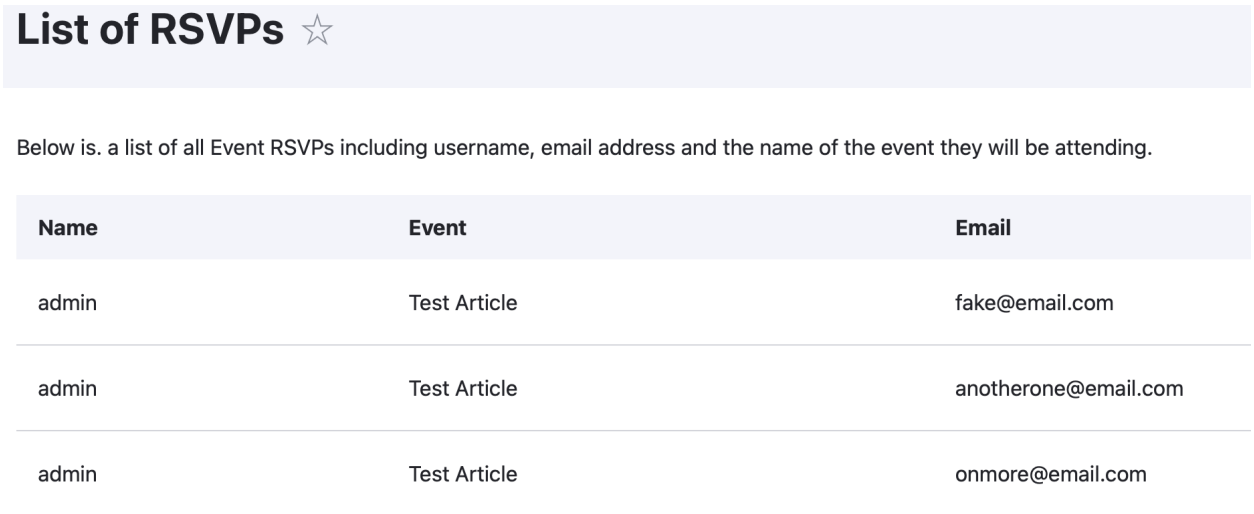
3.1 RSVP List: Module Requirements

Scenario: The client wants site visitors to be able to have an RSVP option available on Event nodes so that she can get a list of attendees from her site including their contact information and the event they are attending.

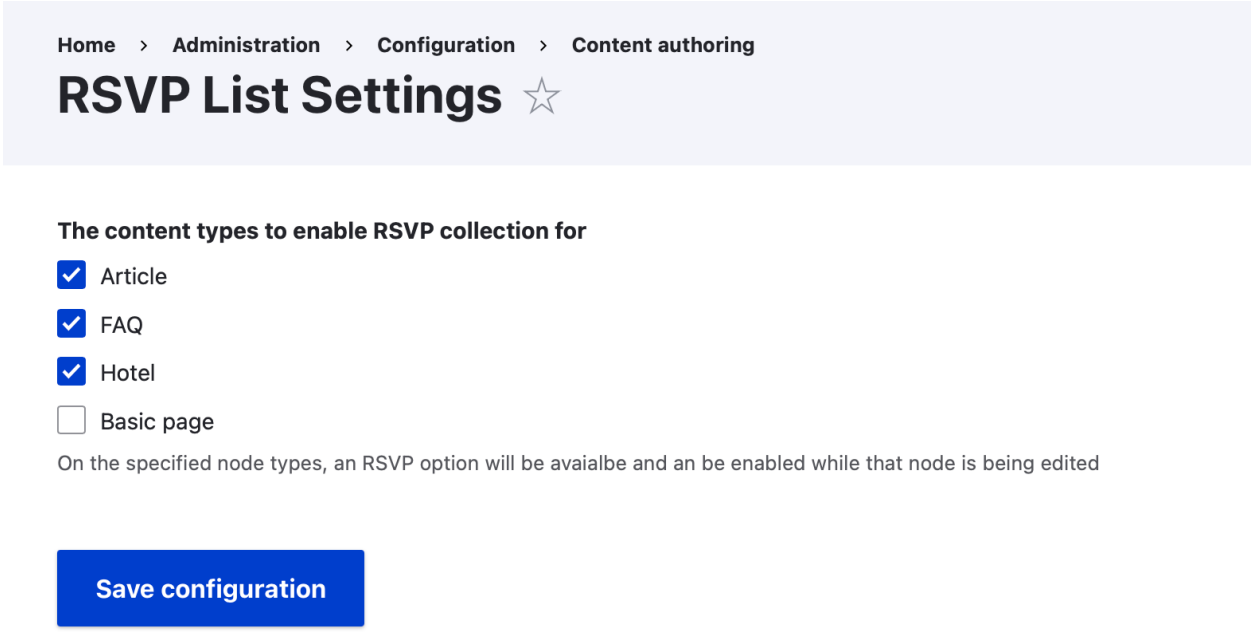
The RSVP List form will display as a block along with nodes when the content editor chooses to collect RSVPs. The form will collect the id for the node it's being displayed with when an RSVP is submitted.



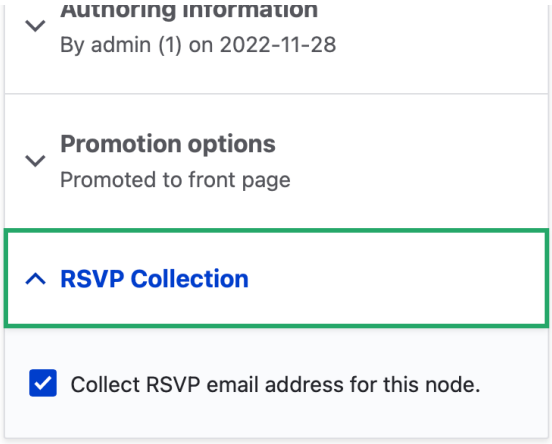
You'll need the module to collect the data submitted and display a report for the content managers, so you'll be working with the database and creating a list controller:



Site administrators will need to be able to configure the module, so you'll create an administrative settings page:



Content editors will need to be able to choose whether or not they want the node they are working on to collect RSVPs, so you'll be editing the node add/edit form.



As you can see, this project will touch on many different aspects of Drupal custom module development. Forms, permissions, blocks, routes, controllers, services--these are all parts of what Drupal module builders need to be able to work with, and you're about to get hands on experience extending them.

So let's get started!

Exercise 3.1.1: Create the "RSVP List" Module

1. Close any open files in your text editor.
2. Create a new folder **inside /custom** called **rsvplist**. This is the name of your project.
3. Right click on rsvplist and click **New File.** Name your new file rsvplist.info.yml and **add** the following coded key: value pairs (**note: the manual wraps the text on the description line. Do NOT do that).**

```
name: RSVP List
type: module
description: Allows users to RSVP for Events and provides a contact list
for administrators.
version: 1.0
core_version_requirement: ^10 || ^11
package: Custom
dependencies:
  - drupal:block
```

4. **Save** your file.

5. **Visit** the Extend page. You should find your **RSVP List** project in the list. Select it and click **Install** to enable it.

3.2 RSVP List: Create a Form

The RSVP List module allows users to RSVP to an event. To collect RSVP information, you will need to create a form.

There are two things to think about when you're creating a form.

- First, how does Drupal locate a form, validate it on submission and store the data?
- Second, how do you define the form itself, including the fields, widgets, user-facing text, etc?

Handling these tasks is simplified by the Drupal Form API.

The Drupal Form API provides a framework for building forms in Drupal. Forms in Drupal are described in multidimensional arrays - sometimes referred to as an array of arrays. This structure handles all of the details to tell Drupal how a form should be rendered.

If you've created forms in HTML, you know that there is a ton of typing involved because there is so much redundancy in the code. Drupal's Form API has all of the form elements and form-related functionality you might need already predefined so you can reuse elements as needed. This makes form building a lot less time consuming.

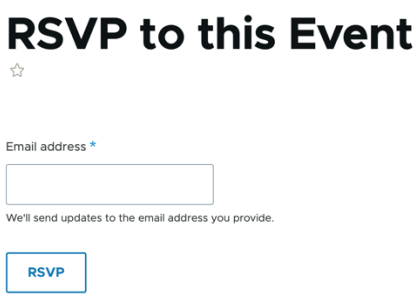
Each form element's properties, called attributes, are held in a subarray for that element. In some cases, form elements can be nested inside other form elements. So attributes are always prefixed with the '#' symbol to allow Drupal to distinguish between a form element and an element's attributes.

Types	
checkbox	textfield
checkboxes	hidden
fieldset	button
password	submit
radio	value
radios	markup
select	textarea

Attributes	
#type	#title
#access	#weight
#attributes	#tree
#default_value	#value
#description	#options
#disabled	#required
#empty	#collapsed

Building the RSVP Email Submission Form

You will build a form to collect your user RSVPs. This form will eventually be rendered in the RSVP Block you'll create with your module. For now, you'll display the form on a custom page so you can see it working. It will look like this:



In Drupal Forms are defined by implementing the \Drupal\Core\Form\FormBuilderInterface and the basic workflow of a form is defined by buildForm, validateForm, and submitForm methods on the interface.

Drupal provides base classes you can extend for easier form creation. There are a few different ones to choose from depending on the type of form you're creating. In most cases you'll start by extending one of these:

- **ConfirmFormBase** - For providing *users* with a form to confirm an action such as deleting a piece of content.
- **ConfigFormBase** - For creating system configuration forms like the one found at admin/config/system/site-information.
- **FormBase** - The most generic base class for generating forms.

For your purposes, you're going to extend FormBase.

Every form in Drupal has an ID which is assigned with getFormId()

After naming the form, you'll define an array of fields using buildForm.

In this exercise, you need to create an array to describe each form element, or field. You are adding each element to the $form array.

The Form API provides us with many form element types to choose from, in this case the first field you need on your form collects an email address, so you'll use the type email. You'll use a few other attributes to describe the email form element.

Use this Drupal Form and render elements page as a guide:

https://api.drupal.org/api/drupal/elements/

Form Submission

The FormBase contains another method for form submission. Since the database isn't set up yet to accept form submissions, you will build a stub function that calls Drupal Messenger Service.

Messenger service

This service helps us communicate to the user. In this case it will indicate that the form is working.

This service provides the ability to produce different types of messages like status, error, warning etc. This service's function takes a message as a parameter and adds it to the top of the page in green. Below are some examples:

```
\Drupal::messenger()->addError(t("migration failed"));

\Drupal::messenger()->addStatus(t("Status changed."));

\Drupal::messenger()->addMessage(t("Form saved."));
```

Learn More About Drupal Messenger Service

https://www.drupal.org/node/2774931

Make sure that you pass the message through the **t() function** to make your message friendly to multilingual websites.

Stub functions

When you need a function or method to exist for other methods to run, but you don't have all the pieces together yet, create a stub function.

In this case, you need to tell your submit handler what to do even though you're not ready to take the next steps yet. You'll eventually want the submission stored in the database, but for now you'll just use the messenger service to hold the method's place.

Forms can be called directly via the routing system which will take care of calling \Drupal::formBuilder()->getForm(). The following example demonstrates the use of a

routing.yml file to display a form at the given route. This will give us a page so you can see your form before continuing to build your module's functionality.

Exercise 3.2.1: Build the Email Submission Form for RSVP List

1. **Create** a folder inside /rsvplist called /src/Form.
2. **Create** your form in a new file called **RSVPForm.php** in your text editor and make sure it is saved in the /src/Form folder.
3. In this file you're going to create an *RSVPForm* class to extend the *FormBase* class available from the Form API.
4. **Enter** the following to start your new file:

```
<?php
/**
 * @file
 * Contains \Drupal\rsvplist\Form\RSVPForm
 */

namespace Drupal\rsvplist\Form;

use Drupal\Core\Database\Database;
use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Messenger\MessengerInterface;

/**
 * Provides an RSVP Email form.
 */
class RSVPForm extends FormBase {

  /**
   * {@inheritdoc}
   */
  public function getFormId() {
    return 'rsvplist_email_form';
  }
```

5. Now you'll use the buildForm() method to **define** your new form.

```
  /**
   * {@inheritdoc}
   */
  public function buildForm(array $form, FormStateInterface $form_state) {
    $node = \Drupal::routeMatch()->getParameter('node');
    $nid = $node ? $node->nid->value : false;
    $form['email'] = [
      '#title' => t('Email address'),
      '#type' => 'textfield',
      '#size' => 25,
      '#description' => t("We'll send updates to the email address you provide."),
      '#required' => TRUE,
    ];
    $form['submit'] = [
      '#type' => 'submit',
      '#value' => t('RSVP'),
    ];
    $form['nid'] = [
      '#type' => 'hidden',
      '#value' => $nid,
    ];
    return $form;
  }
```

6. After building the form array, you'll **add** a stub function for the submitForm() method.

```
  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state) {
    $this->messenger()->addMessage(t('The form is working.'));
  }
}
```

7. **Save** the file.

8. Now, you'll **create** a routing file for your module named rsvplist.routing.yml in your text editor. **Save** the file to your modules/custom/rsvplist directory.

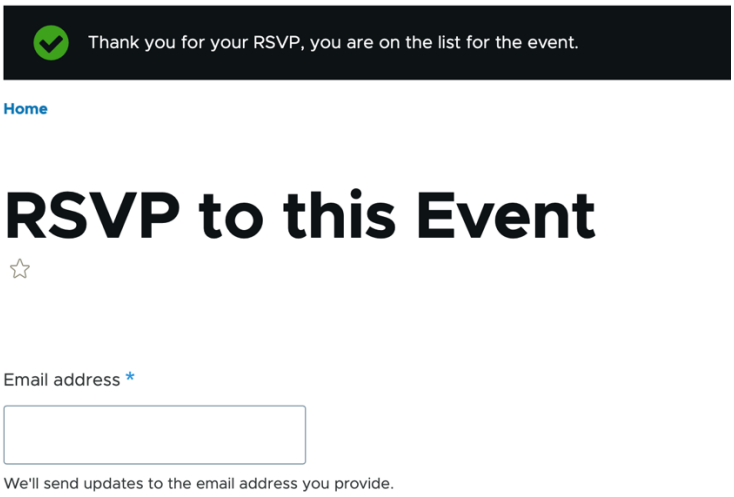
9. The routing file will tell Drupal where your form should appear. You will temporarily place the form on a page called */rsvplist* so you can see how it looks and works as the module is built.

10. Enter the following code:

```
rsvplist.form:
  path: /rsvplist
  defaults:
    _form: \Drupal\rsvplist\Form\RSVPForm
    _title: RSVP to this Event
  requirements:
    _permission: access content
```

11. **Save** your file.
12. Return to the site and **flush the cache**.
13. **Navigate** to /rsvplist and take a look at your new form. You should see your email collection form appear.
14. **Enter** an email and **click** RSVP. You should see a message with a green checkmark when you **submit** an entry.

The code: $nid = $node ? $node->nid->value : false; is a ternary operator that condenses "if / then" statements. See https://www.geeksforgeeks.org/php-ternary-operator/ for more information.



More About the Drupal Form API

Drupal\Core\Form\FormInterface interface states that any form object should have getFormId, buildForm, validateForm, and submitForm methods. It turns out that this matches up nicely with the *build -> validate -> submit* workflow.

When a user visits a URL on a site, */contact* for example, Drupal needs to return the HTML representation of the required form so that it can be displayed in the user's browser. In order to get that form definition, Drupal loads the required form object. Then Drupal calls the buildForm() method on that object. This returns a Form API array that Drupal can turn into HTML. Often this HTML also includes a button that a user can click. Clicking the button generates an HTTP Post request to the URL defined as the action of the form. In Drupal, this is the same URl in which the form is displayed (i.e., /contact).

This time, however, when Drupal gets the request for */contact* it notices also that the request contains $_POST data. This means that the form being requested has actually just been submitted, and it should proceed to the next step in the workflow prior to actually submitting, which is *validation*.

So Drupal instantiates your form object and calls the validateForm() method, which it knows is present because you're implementing the FormInterface. If the validation handler determines there are any errors in the data it flags them, and Drupal halts processing. Then it displays the form to the user to get errors fixed and waits for the user to submit the form again before proceeding.

If no errors are found, Drupal moves on to the submission step of the workflow by calling your form objects submitForm(). Here you perform whatever logic is necessary with the data you collect in the form, like save it to the database or a config file.

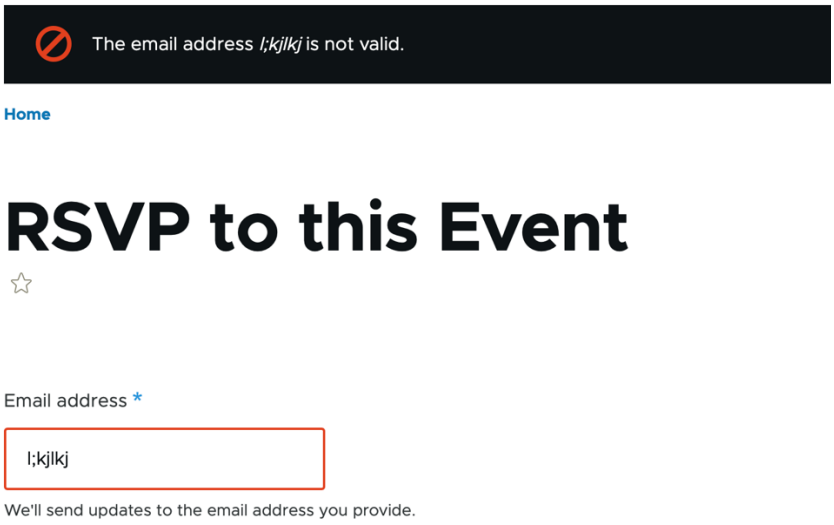
Form Validation

Whenever you create a form with Drupal's form api, you have additional validation and submission functionality tools already built for you. The validation tools allow you to tap into the form API for pre-built validation rules, and then stop processing if the submission does not pass the test.

For the RSVP List module, you will need to validate your form in order to ensure that the email addresses you are collecting are emails, and not phone numbers or other data that can't be used to email.

Validation Handlers

If you try submitting the form right now, you'll see that you can enter any text into the form, and it will validate. You want the form to reject email addresses that are invalid and return an error like this:



The validateForm method can be added to handle validation of any aspect of your form submissions.

```
\Drupal::service('email.validator')
```

Email validator is a service included in core. It uses a predefined set of rules to verify the syntax of a submitted email address. The code for this service can be found in the *vendor/egulias/* directory.

setErrorByName()

Depending on the situation, both setError()and setErrorByName() can be called to handle invalid submissions. These functions allow you to specify an error message for display and stop submission of the form. Be sure to pass the error message through the t function so it will be translatable on multilingual sites.

Exercise 3.2.2: Add a Validation Handler to RSVP List

1. In **RSVPForm.php**, in between the buildForm() and submitForm() functions, **add** this:

```
/**
 * {@inheritdoc}
 */
  public function validateForm(array &$form, FormStateInterface $form_state)
{
    $value = $form_state->getValue('email');
    if (!(\Drupal::service('email.validator')->isValid($value))) {
      $form_state->setErrorByName('email', $this->t('The email address %mail
is not valid.', ['%mail' => $value]));
      return;
    }
  }
```

2. **Save** your file.
3. **Refresh** your browser.
4. **Navigate** to /rsvplist and try **submitting** a bogus email address. You should see your form error message appear in red directly below the menu.

Now that you have the form set up, you need to create a place to store submissions.

For Further Study: Drupal Standards

Drupal provides a standard, easy to use, easy to extend and secure way of adding forms to your Drupal website: Form API or FAPI for short.

https://www.drupal.org/docs/drupal-apis/form-api

Specific list of form and render elements:

https://api.drupal.org/api/drupal/elements/11.x

3.3 RSVP List: The Install File

Install files are run the first time your module is enabled and are used to run setup procedures as required by the module. The most common task of the .install file is to

create the necessary database tables and fields for your module. Since the .install only runs when a module is enabled, you will need to go through the process of uninstalling rsvplist in Drupal. Then, when you enable the module again - Drupal will consider it the FIRST time.

There can be some confusion over these terms. When you add a project to your codebase, you typically think of that process as "installing" - but that's not the usage in this case.

Term	Definition
Install (Enable)	Enable a module in the codebase for use by Drupal.
Uninstall	Remove all traces of a module in Drupal's database, but does not remove any files from the codebase.

There are four Drupal hooks commonly associated with .install files. Remember that when invoking them, you replace "hook" with the name of your module. So hook_schema() becomes rsvplist_schema(), etc.

hook_schema()

This hook is used to define a module's database tables and fields.

http://api.drupal.org/api/function/hook_schema/

hook_install()

This hook allows you to add any additional setup tasks necessary during installation.

http://api.drupal.org/api/function/hook_install

hook_uninstall()

This hook allows you to add any additional tasks necessary during uninstall.

http://api.drupal.org/api/function/hook_uninstall

hook_update_N()

You use this hook to perform a single update between minor versions of a module.

http://api.drupal.org/api/function/hook_update_n

Implementing hook_schema() for RSVP List

In the rsvplist.install file, you need to implement hook_schema to define the database table needed to store rsvplist subscriptions. You will define a table called 'rsvplist' with the following fields:

- An ID, which will serve as the primary key for the data (type 'serial')
- The user ID to keep track of which user created the subscription (type 'int')
- The node ID, to track which node the user signed up to (type 'int')
- The email address submitted (type 'varchar')
- The created date, to track the timestamp when the user subscribed (type 'int')

These fields are added to a serialized array for the rsvplist table.

The new schema will only be loaded when the module is first installed. Because the module is already installed, you will need to uninstall and re-enable it to see the changes to the database.

See the Schema API Handbook at https://www.drupal.org/node/146843 for details on schema definition structures. Note that foreign key definitions are for documentation purposes only; foreign keys are not created in the database, nor are they enforced by Drupal.

Now that you've been introduced to hook_schema(), go ahead and create an install file that sets up your database schema by following the directions in the next exercise.

Exercise 3.3.1: Create an Install File for RSVP List

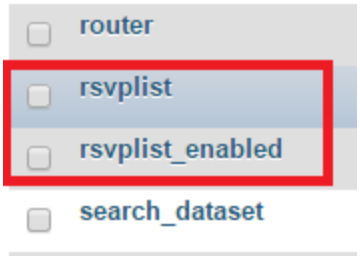
1. **Use** *PHPMyAdmin* to view your Drupal database, by clicking the **link** labeled *Local database:*
2. **Navigate** through the tables to see there is no *rsvplist* table available.
3. **Create** a new file called rsvplist.install in your rsvplist project folder.
4. **Add** the following code. Notice we're invoking hook_schema by replacing the word hook with our project name in the function name. (We'll actually just upload this file from the zip file. It is here for reference)

```
<?php
/**
 * Implements hook_schema().
 *
 */
function rsvplist_schema() {
  $schema['rsvplist'] = [
    'description' => 'Stores email, timestamp, nid and uid for an rsvp',
    'fields' => [
      'id' => [
        'description' => 'The primary identifier for the record.',
        'type' => 'serial',
        'unsigned' => TRUE,
        'not null' => TRUE,
      ],
      'uid' => [
        'description' => 'The {users}.uid that added this rsvp.',
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
      ],
      'nid' => [
        'description' => 'The {node} .nid for this rsvp.',
        'type' => 'varchar',
        'unsigned' => TRUE,
        'not null' => FALSE,
        'default' => 0,
      ],
      'mail' => [
        'description' => 'User\'s email address.',
        'type' => 'varchar',
        'length' => 64,
        'not null' => FALSE,
        'default' => '',
      ],
```

```
      'created' => [
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
        'description' => 'Timestamp for when rsvp was created.',
      ],
    ],
    'primary key' => ['id'],
    'indexes' => [
      'node' => ['nid'],
      'node_user' => ['nid', 'uid'],
    ],
  ];
 $schema['rsvplist_enabled'] = [
    'description' => 'Tracks whether rsvplist is enabled for a node.',
    'fields' => [
      'nid' => [
        'description' => 'The {node}.nid that has rsvplist enabled.',
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
      ],
    ],
    'primary key' => ['nid'],
  ];
  return $schema;
}
```

Later in the project, you will need to store data for node settings. To avoid uninstalling again, you can specify the schema for that table now.

5. Save rsvplist.install
6. **Uninstall** RSVP List and **then re-enable** it on the *Extend* page.
7. **Use** *PHPMyAdmin* to view your database again. **Navigate** to the tables and **find** *rsvplist* and *rsvplist_enabled* tables are now among them.



8. **Check** the structure of the tables. They should be ready to receive data.

Checking the database with DDEV (if you're using DDev to take the class).

1. In the terminal type: `mysql -u db -pdb db`
2. Next, enter `show tables;` and look for the two tables above.
3. Later we'll use: `select * from rsvplist;` to view all the entries.

3.4 RSVP List: Database Integration 1

The database abstraction layer allows users of different database servers to use the same base code to manage data. Drupal's database abstraction layer provides a unified database query API that can query databases running on several popular platforms.

It is built upon PHP's PDO (PHP Data Objects) database API, and inherits much of its syntax and semantics from there. Besides providing a unified API for database queries, the database abstraction layer also provides a structured way to construct complex queries.

Drupal's database abstraction layer provides us with a number of useful functions to interact with the currently active database.

Using Database Abstraction Layer

For simple SELECT queries that do not involve entities, the Drupal database abstraction layer provides the functions **\Drupal::database()->query()** and **\Drupal::database()->queryRange()**, which execute SELECT queries (optionally with range limits) and return result sets that you can iterate over using foreach loops.

INSERT, UPDATE, and DELETE queries need special care in order to behave consistently across databases; you should never use \Drupal::database()->query() to run an INSERT, UPDATE, or DELETE query. Instead, use functions **\Drupal::database()->insert()**, **\Drupal::database()->update()**, and **\Drupal::database()->delete()** to obtain a base query on your table, and then add dynamic conditions).

> *For further study*
>
> *Check the detailed documentation about using Database Abstraction Layer at: https://www.drupal.org/docs/8/api/database-api/database-api-overview*

Now that the rsvplist table exists in the database, the next step is to add data to it when a user RSVPs to an Event. In the next exercise, you'll need to return to the submitForm method that you created earlier as a stub and update it to insert the form values into your rsvplist table.

Exercise 3.4.1: Add RSVP Submissions to the Database

1. In RSVPForm.php, **return** to your submitForm()stub function and **edit** it so that it matches the code below:

```
  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state) {
    $user = \Drupal::currentUser();
    $query = \Drupal::database()->insert('rsvplist');
    $query->fields([
      'mail',
      'nid',
      'uid',
      'created',
    ]);
    $query->values(
      [ $form_state->getValue('email'),
        $form_state->getValue('nid'),
        $user->id(),
        time(),
      ]
    );
    $query->execute();
    $this->messenger()->addMessage(t('Thank you for your RSVP, you are on the
list for the event.'));
  }
}
```

2. **Save** your file.
3. **Flush** the cache.
4. **Navigate** to /rsvplist and submit a valid email address.
5. **Go** to your database and **check** the *rsvplist table*. You should see your submitted email recorded.
6. **You may shorten the query** into a compact form: https://www.drupal.org/docs/drupal-apis/database-api/insert-queries . **Note**: $this is **a reserved keyword in PHP that refers to the calling object**. It is usually the object to which the method belongs. For more on this read: https://www.phptutorial.net/php-oop/php-this/

3.5 RSVP List: Permissions

So far, anyone, registered user or anonymous visitor, can sign up using the rsvplist module who has permission to 'access content'. To refine your module and provide more granular control over who can subscribe, you'll need to add some custom permissions.

Drupal's role-based permissions system is typically used to determine who can access menu items. Many modules define their own permissions.

rsvplist.permissions.yml

A .permissions.yml file is used to define permissions that site administers can use to configure your module with. It works similarly to other .yml files you've created for your module prior. For each permission key, you'll add a title and description that will appear to users on the People > Permissions page. You can set a key attribute of *restrict access* to *TRUE* if you'd like admins to know the permission is intended for administrators only.

Defining RSVP List Permissions

Here are the titles for permissions you'll need to define:

- View RSVP Form
- Access List of submitted RSVPs
- Administer RSVP Settings

Once permissions are defined, the key can be used to control access to any functionality in your module. We will add our *view rsvplist* key to the permissions attribute of our */rsvplist* route, so only users with permission can see the form at that path.

Exercise 3.5.1: Define RSVP List Permissions

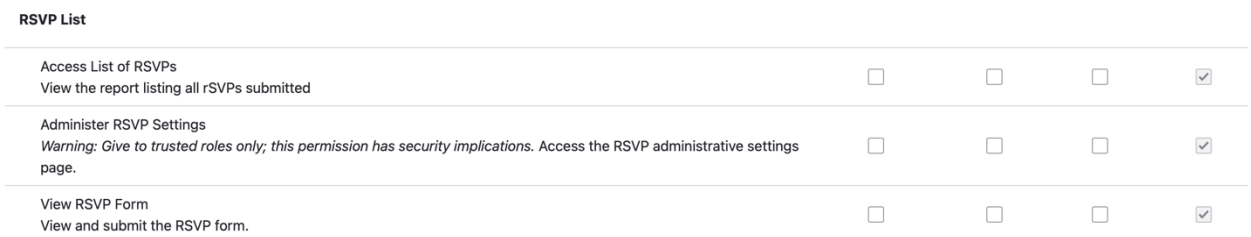
1. **Create** a new file called In rsvplist.permissions.yml in the rsvplist folder and **add** the following code:

```
view rsvplist:
  title: View RSVP Form
  description: View and submit the RSVP form.
access rsvplist report:
  title: Access List of RSVPs
  description: View the report listing all RSVPs submitted.
administer rsvplist:
  title: Administer RSVP Settings
  description: Access the RSVP administrative settings page.
  restrict access: true
```

2. **Save** your file.
3. **Log out** and **navigate** to */rsvplist*. You should see the form because it is not yet restricted. Once verified, **log back in**.
4. Now, return to your routing file and add the *View RSVP Form* permission to your rsvplist.routing.yml permissions:

```
rsvplist.form:
  path: /rsvplist
  defaults:
    _form: \Drupal\rsvplist\Form\RSVPForm
    _title: RSVP to this Event
  requirements:
    _permission: view rsvplist
```

5. **Return** to the site and **flush** the cache.
6. **Log out** and navigate to */rsvplist* as an anonymous user. You should no longer have access to the form.
7. **Login** and **navigate** to *admin/people/permissions*. Find the *RSVP List* section and look for the permissions you defined in code.



8. Give *View RSVP Form* to Anonymous and *all three RSVP List permissions* to Authenticated users. Test your settings.
9. Now that you have permission keys defined you can use them to control access to the form block, report and administrative settings page as you build them.

3.6 RSVP List: Create the Block

You want to add a block that contains your RSVPForm so that site administrators can have maximum display flexibility when using your module. Once the block has been added, it will show up on Drupal's block configuration page *admin/structure/block*.

Adding a New Block Instance for RSVP List

Blocks in Drupal are instances of the block plugin.

Most of the plugins in Drupal will use annotations to register themselves and describe their metadata. Some of the plugins types provided by core are:

- Blocks (see /src/Plugin/Block/ for many examples)
- Field formatters, Field widgets (see /src/Plugin/Field/ for many examples)
- All Views plugins (see /src/Plugin/views/ for many examples)
- Conditions (used for Block visibility in core)

The Drupal block manager scans your module's classes for a class that contains the @Block annotation (a special comment right above your class declaration).

In the next exercise, you'll add a new block and annotate it with @Block.

Exercise 3.6.1: Creating a RSVP List Subscription Block

1. Create new folders inside /src. Add /Plugin/Block/
2. Create a new text file called RSVPBlock.php inside /src/Plugin/Block.
3. **Open** RSVPBlock.php in your text editor.
4. At the very top, **add** this code to declare it's a PHP file, and what it will contain:

```
<?php
/**
 * @file
 * Contains \Drupal\rsvplist\Plugin\Block\RSVPBlock
 */
```

5. Next **declare the namespace**, and any other classes you will use. Since you are creating a block, you want to declare the BlockBase class you will be extending your class from. We'll also need AccountInterface and AccessResult when we control access to the block later.

```
namespace Drupal\rsvplist\Plugin\Block;


 use Drupal\Core\Block\BlockBase;
 use Drupal\Core\Session\AccountInterface;
 use Drupal\Core\Access\AccessResult;
```

6. After that, you will **insert an annotation** comment, so that Drupal can register your Block with the block manager. This annotation is necessary to have when creating custom blocks for your module.

```
/**
 * Provides an 'RSVP' List Block
 *
 * @Block(
 *   id = "rsvp_block",
 *   admin_label = @Translation("RSVP Block"),
 * )
 */
```

7. Finally, **add your class**. Again, you are extending this from BlockBase. BlockBase implements the BlockPluginInterface, which means you need to define the build()method which is expected to generate a render array.

```
class RSVPBlock extends BlockBase {
  /**
   * {@inheritdoc}
   */
  public function build() {
    return ['#markup' => $this->t('My RSVP List Block')];
    }
  }
```

8. **Save** your file.
9. **Flush the cache**.
10. **Navigate** to admin/structure/block
11. **Scroll** down to the *Sidebar* region.
12. **Click** on *Place block*.
13. **Find** *RSVP block* and **click** on *Place block* again.

14. Keep the defaults and **click** *Save block*.
15. Your block is placed! Click *Back to site* and take a look. In the sidebar, you should see something like this:



With this working, you'll add the actual form display in the next exercise.

\Drupal::routeMatch()->getParameter()

One of the many benefits of tapping into Drupal's form API is utilizing all of the functionality that already exists. One handy function is called getParameter. You can use this function to get information from the current path. You will use it to get the id of the node currently displayed.

\Drupal::formBuilder()->getForm()

Another handy function is called getForm. You will often use this function as a menu callback. When supplied with a form name of a class (or instance of a class) that implements FormInterface, this function collects all of the data and returns the form array for display.

AccessResult::allowedIfHasPermission

You will call the blockAccess function to set who gets to see the block. This function will return the results of AccessResult::allowedIfHasPermission. *AcessResult* is a value object used for passing an access result. *isAllowed()* will be TRUE if the account has permission.

Exercise 3.6.2: Replace Block Placeholder Text with RSVPForm and Apply Access Control

1. In RSVPBlock.php alter the build() method to match the code below:

```
  /**
   * {@inheritdoc}
   */
  public function build() {
    return \Drupal::formBuilder()->getForm('Drupal\rsvplist\Form\RSVPForm');
    }
```

2. **Add** the following function beneath your build() to control when the block form will be displayed. You will need to match the permission key *'view rsvplist'* that was defined in exercise 3.5.1.

```
  /**
   * {@inheritdoc}
   */
  public function blockAccess(AccountInterface $account) {
    /** @var \Drupal\node\Entity\Node $node */
    $node = \Drupal::routeMatch()->getParameter('node');
    $nid = $node ? $node->nid->value : false;
    if(is_numeric($nid)) {
      return AccessResult::allowedIfHasPermission($account, 'view rsvplist');
    }
    return AccessResult::forbidden();
  }
}
```

3. **Save** your file.
4. **Flush** the cache.
5. **Navigate** to your home page. You will not see the block at all. Navigate to any node and you will see the block display because you are logged in as admin and on a node. Log out and you should NOT see the block display.
6. With the blockAccess function working, **log back into your site**.

You just added a block containing the RSVP Form to the block admin page using an implementation of the build() method from BlockBase and made sure only people with

permissions to view and use the form have access with the blockAccess() method. This took care of the functional aspects of the block. However, you may also want to address the aesthetic aspects of your module output, to do so you would need to register a theme function and template for your block.

Theming Block Display

To create a theme template for your block, you'll need to add your module to the theme registry and create a .html.twig template file using Twig syntax.

The Theme Registry

The theme registry is where Drupal keeps track of all theming functions and templates. Every themeable item in Drupal is controlled by either a template or a function. When the theme registry is built, Drupal looks for hook_theme()implementations in modules to discover theme functions and to discover information about each themeable item.

hook_theme()

Modules must implement hook_theme()to declare their themeable functions and templates. The hook declares how a particular render array should be rendered in HTML. The name of the element you create as *'element_name'* will automatically be called as the template name.

element-name.html.twig

To create your new template, you simply create a file in your rsvplist module directory and give it the same name as the template you registered using hook_theme() in the previous step. The naming difference is that the hook element has an underscore and the template has a dash. Make sure you include the .html.twig extension.

3.7 RSVP List: Build Administrative Settings Page

You've created an RSVP form and it currently displays in a block with any node on the site. To make your module more useful, you can set up configuration settings for administrators. The requirement for this project is to allow administrators to configure which content types have RSVP List functionality available.

Administrative Menu Callback Files

To display your module's configuration form in the administrative interface, you'll want to give it a path that begins with admin. You set the path and provide menu links to a page with two YML files that are added to your project's root directory:

4. [modulename].routing.yml The routing file you created earlier will get a new entry to represent the location of the admin settings form. Most module setting/ configuration forms can be found at admin/config/... You will see how the paths work in the next exercise.

5. [modulename].links.menu.yml file allows you to define a menu link in the Drupal menu system. For module setting/configuration pages, you declare the parent menu: system.admin_config to represent where you want your module's configuration page to sit in the administrative menu.

First you'll start building your configuration page by defining the route and menu link. With that information in place, you can also update your *.info.yml* file with a configuration key:value.

Exercise 3.7.1: Add Routing and Menu Link for RSVP List Administrative Settings Form

1. **Open** rsvplist.routing.yml. Since we have our RSVPForm displaying in a block, we no longer need the *rsvplist.form* route. Comment out the route by selecting all the text and pressing Ctl + / (Windows) or Command + / (Mac).

2. Add the following:

```
rsvplist.admin_settings:
  path: /admin/config/content/rsvplist
  defaults:
    _form: \Drupal\rsvplist\Form\RSVPSettingsForm
    _title: RSVP List Settings
  requirements:
    _permission: administer rsvplist
```

3. **Save** your file.
4. **Create** a new file in your project's root directory called rsvplist.links.menu.yml.
5. Add the following menu link definition:

```
rsvplist.admin_settings:
  title: RSVP List Settings
  description: Control display of RSVP form
  route_name: rsvplist.admin_settings
  parent: system.admin_config_content
  weight: 10
```

6. **Save** your file.
7. Normally you would create your configuration page before declaring it in your .info file. You can **open** the rsvplist.info.yml and add the following route at the end of the file.

```
configure: rsvplist.admin_settings
```

8. **Save** your file.
9. You'll need to create the form (next exercise) before anything will appear at that route.

ConfigFormBase

You have already created a form with the Drupal Form API. In that case, you extended the standard FormBase to inherit methods to build, validate and submit your form. To create forms for configuration pages on your website, you use ConfigFormBase. This type of form allows you to utilize $config objects to communicate with your modules settings file. When using a $config object you can use ::get(), ::set(), and ::save() methods to handle storage and retrieval of your module settings. For more information see: https://www.drupal.org/docs/drupal-apis/configuration-api/working-with-configuration-forms.

Module Configuration Files

The configuration API introduces two additional files you'll create to hold and maintain settings for your module. They are both saved in a config directory:

[modulename].settings.yml The settings file is saved in your *config/install* directory. It contains any config objects that you need for your module. You can have as many configuration objects as needed.

[modulename].schema.yml The schema file for your module is saved in your *config/schema* directory. It holds definitions and mapping for each of your configuration objects.

node_type_get_names()

When you are populating the options on a configuration form, and you want to have a current list of all content types on the site node_type_get_names() conveniently grabs the current list for. You'll use that function to populate checkbox options on your form.

Exercise 3.7.2: Create RSVPSettingsForm

1. **Create** your form in a new file called **RSVPSettingsForm.php** inside the **src/Form** directory/
2. In this file you're going to create an *RSVPSettingsForm* class to extend the Config*FormBase* class available from the Form API. There is a new function introduced here *getEditableConfigNames()* where you will name the settings file you will be creating next.
3. **Enter** the following to start your new file:

```
<?php
/**
 * @file
 * Contains \Drupal\rsvplist\Form\RSVPSettingsForm
 */

namespace Drupal\rsvplist\Form;

use Drupal\Core\Form\ConfigFormBase;
use Drupal\Core\Form\FormStateInterface;

/**
 * Defines a form to configure RSVP List module settings
 */
class RSVPSettingsForm extends ConfigFormBase {
  /**
   * {@inheritdoc}
   */
  public function getFormID() {
    return 'rsvplist_admin_settings';
  }
   /**
```

```
   * {@inheritdoc}
   */
  protected function getEditableConfigNames() {
    return ['rsvplist.settings'];
  }
```

4. Now you'll use the buildForm() method to **define** your new form.

```
  /**
   * {@inheritdoc}
   */
  public function buildForm(array $form, FormStateInterface $form_state) {
    $types = node_type_get_names();
    $config = $this->config('rsvplist.settings');
    $form['rsvplist_types'] = [
      '#type' => 'checkboxes',
      '#title' => $this->t('The content types to enable RSVP collection
for'),
      '#default_value' => $config->get('allowed_types'),
      '#options' => $types,
      '#description' => $this->t('On the specified node types, an RSVP option
will be available and can be enabled while that node is being edited.'),
    ];
    return parent::buildForm($form,$form_state);
  }
```

5. After building the form array, you'll **add the** submitForm() method.

```
  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state) {
    $allowed_types = array_filter($form_state->getValue('rsvplist_types'));
    sort($allowed_types);
    $this->config('rsvplist.settings')
      ->set('allowed_types', $allowed_types)
      ->save();
    parent::submitForm($form, $form_state);
  }
}
```

6. **Save** the file.

7. Notice that you referenced a settings file called rsvplist.settings and used *get()* and *set()* functions on a *$config* object called *allowed_types*.
8. **Create** two new folders inside rsvplist: config/schema and then inside config add a folder called install.
9. **Create** that *$config* object in a settings file for your module named rsvplist.settings.yml inside the **config/install** directory. This would automatically select article IF the module was re-installed.

```
allowed_types:
  - article
```

10. **Save** the file.
11. Now, to tell Drupal more about your $config object, **create** a new schema file for your module named rsvplist.schema.yml inside the **config/schema** directory.

```
# Schema for the configuration file of rsvplist module.
rsvplist.settings:
  type: config_object
  label: RSVP List Content Type Settings
  mapping:
    allowed_types:
      type: sequence
      label: Content types RSVP Form can display on
      sequence:
        type: string
        label: Content type
```

12. **Save** the file.
13. Return to the site and **flush the cache**.
14. **Navigate** to Configuration -> Content Authoring -> RSVP List Settings and take a look at your new form. You should see the RSVPSettingsForm appear.
15. **Note:** You will see a large error at the top of the page. This is due to installing the module *before* the configuration object was created. Once you click Save the first time, this error will disappear.
16. **Select** a checkbox and **click** Save configuration. You should see a green message to confirm your configuration options have been saved.

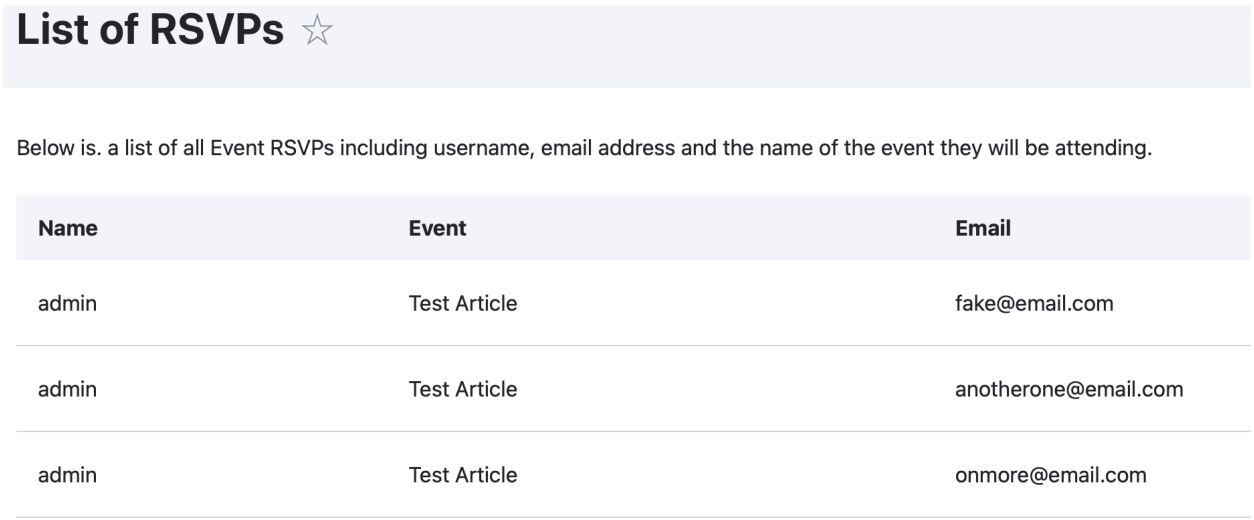
An excellent resource on schema files is at https://www.drupal.org/docs/drupal-apis/configuration-api/configuration-schemametadata - make sure you download the cheat sheet!

3.8 RSVP List: Reporting Results

You want to provide site administrators with a list of all the RSVP List signups so they know who is attending their events. First, since you'll be building another page, you'll need to add another route for the path *admin/reports/rsvplist*. You will build your report page by extending the Controller Base.

Creating a Page to List Results

The reporting page will include a table listing all the RSVPs on the website. The table will appear like this:



List of RSVPs ☆

Below is. a list of all Event RSVPs including username, email address and the name of the event they will be attending.

Name	Event	Email
admin	Test Article	fake@email.com
admin	Test Article	anotherone@email.com
admin	Test Article	onmore@email.com

Exercise 3.8.1: Add Routing and Menu Link for RSVP Report Page

1. **Open** rsvplist.routing.yml. Add your new path to the bottom of the file:

```
rsvplist.report:
  path: /admin/reports/rsvplist
  defaults:
    _controller: \Drupal\rsvplist\Controller\ReportController::report
    _title: List of RSVPs
  requirements:
    _permission: view rsvplist
```

2. **Save** your file.

3. **Open** rsvplist.links.menu.yml and add the following menu link definition:

```
rsvplist.report:
  title: List of RSVP submissions
  description: View listing of RSVPs
  route_name: rsvplist.report
  parent: system.admin_reports
  weight: 0
```

4. **Save** your file.

5. Nothing will appear at that route until you create the report.

ControllerBase

When Drupal gets a request, it produces a response object that is sent back to the client. The output is typically generated by a piece of code called the controller. Although a controller can be a PHP function, a method or an object - in Drupal the best practice is to use a controller class.

When creating your Controller, you will use the ControllerBase to get access to some utility methods and the Container, which reduces the amount of code written to get the results out of the database and into a table display.

Exercise 3.8.2: Create an RSVP List Reports Page

1. **Create** your report in a new file called **ReportController.php** in the **src/Controller** directory.
2. In this file you're going to create a *ReportController* class to extend the *ControllerBase*.
3. **Enter** the following to start your new file:

```
<?php
/**
 * @file
 * Contains \Drupal\rsvplist\Controller\ReportController.
 */
namespace Drupal\rsvplist\Controller;


use Drupal\Core\Controller\ControllerBase;
```

```
use Drupal\Core\Database\Database;


/**
 * Controller for RSVP List Report
*/
class ReportController extends ControllerBase {
```

4. Now you'll create a load() method to **query** the database.

```
  /**
   * Gets all RSVPs for all nodes.
   *
   * @return array
   */
  protected function load() {
    $select = Database::getConnection()->select('rsvplist', 'r');
      // Join the users table, so we can get the entry creator's username.
    $select->join('users_field_data', 'u', 'r.uid = u.uid');
      // Join the node table, so we can get the event's name.
    $select->join('node_field_data', 'n', 'r.nid = n.nid');
    // Select these specific fields for the output.
    $select->addField('u', 'name', 'username');
    $select->addField('n', 'title');
    $select->addField('r', 'mail');
    $entries = $select->execute()->fetchAll(\PDO::FETCH_ASSOC);
    return $entries;
  }
```

5. After querying the *rsvplist $entries*, you'll **add** a report()

```
  /**
   * Creates the report page.
   *
   * @return array
   *   Render array for report output.
   */
  public function report() {
    $content = [];
    $content['message'] = [
      '#markup' => $this->t('Below is a list of all Event RSVPs including
username, email address and the name of the event they will be attending.'),
    ];
    $headers = [
      t('Name'),
      t('Event'),
      t('Email'),
    ];
    $content['table'] = [
      '#type' => 'table',
      '#header' => $headers,
      '#rows' => $this->load(),
      '#empty' => $this->t('No entries available.'),
    ];
    // Don't cache this page.
    $content['#cache']['max-age'] = 0;
    return $content;
  }
}
```

6. **Save** the file.
7. Return to the site and **flush the cache**.
8. **Navigate** to /admin/reports/rsvplist and take a look at your new table.
9. **Submit** an RSVP and return to the report. You should see your new record has been included.

3.9 RSVP List: Altering the Node Edit Form

You need to add a checkbox that allows content editors to enable or disable RSVP collection on a node-by-node basis. This checkbox will appear on the node add/edit form of RSVP List-enabled content types. To add this setting, you will implement hook_form_alter(). When you add hooks to your module, you will put them in a file called the .module.

Using hook_form_alter

Many hook_form_alter implementations start off with a switch to determine which form you're working with (usually based on the form's id). You can also implement hook_form_FORM_ID_alter() to target a specific form. This approach is generally the best practice. See http://api.drupal.org/api/function/hook_form_FORM_ID_alter/8

You tried a hook_form_alter() in your mymodule project earlier. Now you'll try the hook_form_FORM_ID_alter() syntax to alter the node form for RSVP List.

Exercise 3.9.1: Alter the Node Form to Add RSVP List Settings

1. **Create** your hook in a new file called **rsvplist.module** in your text editor. Save the file to your rsvplist directory.

2. In this file you're going to use *FormStateInterface* to manage the form object.

3. **Enter** the following to start your new file:

```
<?php
/**
 * @file
 * RSVP Module hooks.
 */
use Drupal\Core\Form\FormStateInterface;
```

4. Now, you'll **add** your form_alter hook to add RSVP options for content administrators:

```
/**
* Implements hook_form_FORM_ID_alter().
*
```

```
* Alter the node add/edit form to include admin setting for displaying RSVPBlock with
content.
*/
function rsvplist_form_node_form_alter(&$form, FormStateInterface $form_state,
$form_id) {
  $node = $form_state->getFormObject()->getEntity();
  $current_node_type = $node->getType();
  $config = \Drupal::config('rsvplist.settings');
  $types = $config->get('allowed_types', array());
  // RSVP Options for administrators
  if (in_array($current_node_type, $types)){
    $form['rsvplist'] = [
      '#type' => 'details',
      '#title' => t('RSVP Collection'),
      '#access' => \Drupal::currentUser()->hasPermission('administer rsvplist'),
      '#group' => 'advanced',
      '#weight' => 100,
    ];
```

5. To manage the database queries required to check and set if the node is enabled, we'll **introduce a service** that will be defined in the next exercise:

```
    /** @var \Drupal\rsvplist\EnablerService $enabler */
    $enabler = \Drupal::service('rsvplist.enabler');
    $form['rsvplist']['rsvplist_enabled'] = [
      '#type' => 'checkbox',
      '#title' => t('Collect RSVP e-mail addresses for this node.'),
      '#default_value' => $enabler->isEnabled($node),
    ];
    foreach (array_keys($form['actions']) as $action) {
      if ($action != 'preview' &&
        isset($form['actions'][$action]['#type']) &&
        $form['actions'][$action]['#type'] === 'submit') {
           $form['actions'][$action]['#submit'][] =
           'rsvplist_form_node_form_submit';
      }
    }
  }
}
```

6. Now that the RSVP field is added to the node form, you'll need to define what happens when the node form is saved. **Add** the following hook to the end of

your .module file:

```
/**
 * Form submission handler for RSVP item field on the node form.
 *
 * @see rsvplist_form_node_form_alter()
 *
 * @param array $form
 * @param \Drupal\Core\Form\FormStateInterface $form_state
 */
function rsvplist_form_node_form_submit(array $form, FormStateInterface
$form_state) {
  /** @var \Drupal\rsvplist\EnablerService $enabler */
  $enabler = \Drupal::service('rsvplist.enabler');
  $node = $form_state->getFormObject()->getEntity();
  if ($form_state->getValue('rsvplist_enabled')) {
    $enabler->setEnabled($node);
  }
  else {
    $enabler->delEnabled($node);
  }
}
```

7. **Save** the .module file.

8. Before this will work, you need to add the rsvplist.enabler service. You'll do that in the next lesson.

3.10 RSVP List: Database Integration II

Remember, in exercise 3.3.1, you added a .install to your project which defined the schema for two tables. The first table, called rsvplist stores the RSVP submissions. The second table, called rsvplist_enabled will be used to store the nid of any nodes that are configured to collect RSVP List subscriptions.

Now, you're going to learn about some of the other database API functions you need to use to update the rsvplist_enabled table in the database.

The Database Layer

When creating the ReportController, you used the Database service (Database::getConnection) to query the rsvplist table for RSVPs.

Database::getConnection

This public static function gets the connection object for a specified database key and target. Here is the syntax:

```
Database::getConnection($target = 'default', $key = NULL)
```

The $target parameter is set to your database table, while $key represents the database connection key and defaults to NULL which means the active key.

Creating a Service

You are going to define some database methods to manage the rsvplist_enabled setting per node in the database. These methods will be called when you create, edit, and delete nodes to keep the table up-to-date. When methods will be called multiple times, it's useful to create them once as a service.

Defining Custom Services

When you create a module and wish to add it to Drupal as a service, you create a file called modulename.services.yml, and define an ID for your service. In this case the ID has already been called in your node_alter hook: rsvplist.enabler

You will also define a path to the location of the default class for the service (note that you use the namespace), and any arguments and tags.

Here is an excerpt from the .services.yml for the core Book module. Following the naming convention, this file is called book.services.yml.

```
services:
  book.breadcrumb:
    class: Drupal\book\BookBreadcrumbBuilder
    arguments: ['@entity_type.manager', '@current_user']
    tags:
      - { name: breadcrumb_builder, priority: 701 }
  book.manager:
    class: Drupal\book\BookManager
```

```
    arguments: ['@entity_type.manager', '@string_translation',
'@config.factory', '@book.outline_storage', '@renderer']
  book.outline:
    class: Drupal\book\BookOutline
    arguments: ['@book.manager']
```

For more information on the structure of a service file see: https://www.drupal.org/docs/drupal-apis/services-and-dependency-injection/structure-of-a-service-file

Argument(s) to be used with the factory method (in case of 'factory_class') or with the class constructor (in case of 'class'). A '@' indicates another service, and then what is put in here is the name of the service, as defined in its services.yml file. An argument wrapped in '%' characters represents a parameter, which can be defined in the 'parameters' section of the file. A '@?' indicates that the dependency is optional.

Exercise 3.10.1: Create rsvplist.enabler Service

1. **Create** your service declaration in a new file called **rsvplist.services.yml** inside the rsvplist directory (project root).
2. **Enter** the following to name your new service:

```
services:
  rsvplist.enabler:
    class: Drupal\rsvplist\EnablerService
    arguments: []
```

3. Now, you'll **add** your EnablerService.php file in the rsvplist/src directory.
4. You'll use a Constructor method __construct (two underscores) to initialize your new class.
5. **Enter** the following to start your new file:

```
<?php
/**
 * @file
 * Contains \Drupal\rsvplist\EnablerService.
 */
namespace Drupal\rsvplist;
use Drupal\Core\Database\Database;
use Drupal\node\Entity\Node;
```

```
/**
 * Defines a service for managing RSVP list enabled for nodes.
 */
class EnablerService {
  /**
   * Constructor.
   */
  public function __construct() {
  }
}
```

6. **Save** file.

Defining the RSVP List Database Queries

You will define three methods for your class to handle help handle any situation:

- The isEnabled() will check whether the node is in the rsvplist_enabled table.
- The setEnabled() will add the node to the rsvplist_enabled table..
- The delEnabled() will remove the node from the rsvplist_enabled table.

Exercise 3.10.2: Add Methods to rsvplist.enabler Service

1. **Open** your EnablerService.php file.
2. **Create** the setEnabled() method, by adding the following to the end of your file - but before the last curly bracket:

```
  /**
   * Sets an individual node to be RSVP enabled.
   * @param \Drupal\node\Entity\Node $node
   */
  public function setEnabled(Node $node) {
    if (!$this->isEnabled($node)) {
      $insert = Database::getConnection()->insert('rsvplist_enabled');
      $insert->fields(array('nid'), array($node->id()));
      $insert->execute();
    }
  }
```

3. **Create** the isEnabled() method, by adding the following to the end of your file - but before the last curly bracket:

```
  /**
   * Checks if an individual node is RSVP enabled.
   * @param \Drupal\node\Entity\Node $node
   * @return bool
   *  Whether the node is enabled for the RSVP functionality.
   */
  public function isEnabled(Node $node) {
    if ($node->isNew()) {
      return FALSE;
    }
    $select = Database::getConnection()->select('rsvplist_enabled', 're');
    $select->fields('re', array('nid'));
    $select->condition('nid', $node->id());
    $results = $select->execute();
    return !empty($results->fetchCol());
  }
```

4. **Create** the delEnabled() method, by adding the following to the end of your file:

```
  /**
   * Deletes enabled settings for an individual node.
   * @param \Drupal\node\Entity\Node $node
   */
  public function delEnabled(Node $node) {
    $delete = Database::getConnection()->delete('rsvplist_enabled');
    $delete->condition('nid', $node->id());
    $delete->execute();
  }
}
```

5. **Save** file.
6. Since these methods are already called in your node_form_alter, you only need Drupal to recognize your new class. simply **return** to the site and **flush** your cache.
7. Test by creating a piece of content that is enabled for RSVP collection (see /admin/config/content/rsvplist to check your configuration).
8. You should **see** the RSVP Collection settings on the node form and selecting the checkbox should populate the rsvplist_enabled table with the nid.

3.11 RSVP List: Conditional Display of RSVP Block

Now that the rsvplist_enabled table is kept up-to-date, you can use it to determine whether or not to show the RSVP block with a node. You will call isEnabled() to verify that a nid should have the RSVP Block displayed to collect emails.

Exercise 3.11.1: Display Signup Form Conditionally

1. **Open** the *RSVPBlock.php* file.
2. **Add** a condition to the blockAccess method to check whether the nid is in the *rsvplist_enabled* table and shorten the code since $nid is not required.
3. Alter the blockAccess method so it matches this code snippet (the lines to modify/ add are highlighted).

```
  /**
   * {@inheritdoc}
   */
  public function blockAccess(AccountInterface $account) {
    /** @var \Drupal\node\Entity\Node $node */
    $node = \Drupal::routeMatch()->getParameter('node');
//    $nid = $node ? $node->nid->value : false; (this line can be deleted)
    /** @var \Drupal\rsvplist\EnablerService $enabler */
    $enabler = \Drupal::service('rsvplist.enabler');
    if($node) {
      if($enabler->isEnabled($node)) {
        return AccessResult::allowedIfHasPermission($account, 'view rsvplist');
      }
    }
    return AccessResult::forbidden();
  }
}
```

4. **Save** the file and clear your cache.
5. You can **test** your conditional displays and settings by trying different configurations.

4.0 Drush and Composer

Once you're familiar with creating modules in Drupal, some of the routine tasks may start to feel repetitive or boring. Luckily, Drupal has two command line interface (CLI) tools available to help speed module development up: Drush and Drupal Console.

In this learning module, you'll learn a little about the advantages of each one, and how to install and use both.

4.1 Different Tools for Differing Needs

Drush and Drupal Console share some capabilities, but offer different advantages depending on what you want to do.

Drush

Drush is a tool that's been around for many years, and through several major Drupal versions. It provides Drupalists with a CLI for common Drupal administration tasks--and also makes it much easier to access parts of Drupal that a server admin or developer might need to touch, but a regular content administrator might not be interested in. You can backup and restore sites using it, clear site cache, create users and assign user roles, install/uninstall modules and more with Drush.

Composer

Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.

Comparing Drush and Composer

Here's a brief breakdown of the capabilities of each CLI tool:

Composer:

- Download Drupal
- Download contributed modules
- Install Drupal
- Update Drupal and contrib module versions

Drush:

- Run updatedb
- Clear the cache
- Run cron
- Run Drupal with a lightweight web server
- Import, export and merge configuration
- Add users and set their roles
- Add permissions to roles
- Back up and restore Drupal
- Copy your database and files to a remote server
- Compile twig templates

If your interactions with Drupal lean towards Drupal server administration or site building, you might end up using Drush more frequently than Composer, as it gives some great commands to speed up common administration tasks.

4.2 Downloading and Installing Drush

Drush is not a module but a tool that provides command-line scripts to speed up development tasks.

Read more about **en** here: http://drushcommands.com/drush-8x/pm/pm-enable/

4.2.1 Exercise: Check that Drush Is Installed

1. In the terminal window, type in the word Drush and the system will respond with a list of possible Drush commands.

4.3 Using Drush

Useful Drush Commands

The power of Drush comes from being able to quickly use the command line to make changes that would take longer using the Administration Toolbar.

Here are some useful and common commands.

Command Name	Description
drush status	Check if Drush is working.
drush cr	Clears all caches.
drush en modulename	Enable a selected module.

You can also reference the following resources:

- http://drushcommands.com/
- https://www.drupal.org/documentation/modules/drush
- http://api.drush.org/

Some commands have aliases, so make sure to do a search if you're looking for more information on a command and you don't immediately see it.

4.3 Using Composer

Thoughout this course we've used composer to install modules.

Here are some helpful links:

https://www.drupal.org/docs/develop/using-composer/using-composer-to-install-drupal-and-manage-dependencies

https://www.drupal.org/docs/develop/using-composer/starting-a-site-using-drupal-composer-project-templates

4.4 Summary

- Command-line tools can help you develop modules more quickly and easily.
- Drush is one tool that simplifies many Drupal site and server administrative tasks.
- Composer is the tool you'll use to download Drupal and contributed modules.

Appendix: How Would You Do That In Drupal?

When you're new to Drupal, you'll probably need to make a few sites to get a feel for all the possibilities Drupal offers you. In this session, you'll learn about real-life practices and how you can get started with your first theme, by looking at some websites you like, and then brainstorming how you'd configure Drupal to do the same thing.

Real-Life Design Considerations

Choose a Case Study

- Working in small groups or pairs, choose an inspiring site from http://drupal.com/showcases
- Here are a couple of prominent sites using Drupal (as of Feb 2016):
- https://www.mskcc.org/ Memorial Sloan Kettering Cancer Center
- http://plbint.com/ PLB International (Pet Food) - Multilingual Site

After you've selected a site, also select one particular page or section. For example, you can choose:

- A landing page
- A top-level category or section
- A full node page

Content Type Planning and Design

Every site has data that needs to be stored somewhere. In Drupal, it's important to design your content types to take into consideration the type of data used for the site's audience.

A content type can be explicitly a type of content. For example, an article with a title body and tags, or an event with a time, date, and location. However, you can also have content types to collect data that will not be used on it's own, like images, galleries, banners, or ads.

On your example site, identify content types appearing on the page.

- Take note of elements from different content types appearing on the page. You may see images or titles from different content types, such as blog articles in one spot, and new products in another.
- Write down as many content types as you can find. Name them however makes sense to you!
- Choose one particular content type to explore.
- Once you've selected a content type, think of the following:
- Content type planning
- Write down the fields you think are in use for the content type.
- What kind of data is being stored in each field? Numbers? Names? Links to other sites? Images? Large amounts of text? And how long is the content? Is it always 10 characters or shorter, or is someone likely to write a novel? Write this down as well. This will influence the field type you select for the field.
- Content type design
- How does a node of this content type appear in different places on the site?
- Are you able to identify a teaser display that only shows you a little of the content?
- Are you able to click through to see an expanded display of content?
- How do the fields display in different locations?
- How many view modes does it have? You may need to poke around to figure this out.

Wireframe

After you have an idea of all the content types in use, consider the content regions.

- What regions do you see being used in a layout?
- Identify what blocks appear consistently across the site on every page, and what blocks only appear on some pages (such as the front page, or on a contact page).
- Identify what contributed modules might be used to control display.
- Examples: Panels, Display Suite, etc.?
- Draw a wireframe of your chosen page.
- Identify how elements of the page may be assembled from static content, or generated from Views or other modules.

Planning Your Theme

After you know what content types you need, and the rough layout of the site, it's time to think of the overall look of your site, and what you might need to do with your theme to accomplish your goals.

- Is there a base theme or theme system you might choose to start with?
- For example, if you resize the browser while looking at a site, does the site resize with that, or stay static?
- Identify where you might need to do the following:
- Override a template file
- Override with a preprocess function
- Use a theme function
- Override a string

Share Your Results

After you get time as a pair or a small group, you will come back together to discuss your findings, and debate the best approaches. Try to offer examples of multiple options that were considered, and talk about why one approach might be better.

Summary

- When learning Drupal, you can select a site that appeals to you, and see if you can make a mockup of it in Drupal.
- Make sure to begin with identifying your content types and the fields in them.
- From there, look at how the site is laid out and where those different content types display, and how their display changes depending on where you are on the site.
- List out what modules might help you with your layout.
- Create a wireframe to guide your layout design later.
- Once you've configured Drupal's content types and layout, explore what base themes would be good candidates to help you manage the finer details of your site's look and style.
- Finally, create your theme to put the finishing touches on your site.

Acknowledgements

Writing and Editing Credits

First edition: Published in November 2020, by Prasad Shirgaonkar, Acquia.

Based on previous Drupal 8 editions. Published in February 2016 and March 2020.

Text and code updates in January 2025 by Rod Martin, Tyto Learning.