WORDPRESS DEVELOPMENT

LECTURE NOTES

LUIS RAMIREZ

Getting Started

In this section, we get started with introductions, WordPress installation, and just some nice to know things. Glad to have you on board. 😄

Introduction

Welcome! 👋 Glad to have you on board! In this lecture, we talked about what you can expect from this course, its prerequisites, and some tips for getting the most out of this course.

What you can expect

- Custom Post Types
- Extending the REST API
- Translations
- Block Development
- WooCommerce
- Custom Database Tables
- Transients
- Data Sanitization
- Code Optimization
- And More!

Prerequisites

- Basic WordPress Administrative Tasks
- HTML
- CSS
- PHP (Optional)
- JavaScript (Optional)

Tips and Tricks

I've got two tips for you. The first of which is to download the PDF. The fact that you're reading this PDF means that you've already downloaded it. Good job! You should use this book to help you review lectures or revisit old topics.

Alternatively, you can treat this as your cheat sheet. 😲

My second tip is to ask questions in the Q&A section. I'm here to help you with your journey of becoming a WordPress developer. I usually respond within 24 hours.

Please refrain from asking questions related to personal projects. Keep your questions limited to the content of the course.

Free PDF Download

There are no notes for this lecture.

What is an environment?

In this lecture, we got into a discussion on what environments are and why they're important. An environment refers to the location where your WordPress site runs. There are two types of environments.

- **Production** - A live site that is publicly accessible to the world. Should store a stable copy of your site.
- **Development** - A site that is privately accessible to developers. Perfect for performing experiments and testing plugins/themes.

For this course, we're going to set up a development environment on our machine. Setting up an environment can be a tedious task.

Luckily, there are programs available for quickly installing WordPress with the necessary programs and configurations. The program I recommend is called Local. Installing Local is like installing any other program. Install Local before proceeding to the next lecture.

Resources

- Local

Installing WordPress

In this lecture, we installed WordPress with Local. There are no notes for this lecture. The content is meant to be consumed via video.

Local Quick Tour

In this lecture, we explored Local for managing WordPress. The content for this video is meant to be consumed via video. There are no lecture notes.

Text Editor

In this course, I'll be using a text editor called Visual Studio Code. Check it out here: https://code.visualstudio.com/

You don’t have to use the same editor, but I recommend it for consistency. If you’d like to use a different editor, be sure it supports HTML, CSS, JavaScript, and PHP. These are the languages we’ll be using in this course. As long as those languages are supported, you’re good to go.

PHP Fundamentals

In this section, we got started with the fundamentals of PHP. It's the language WordPress is written with. So it's an essential language to learn.

Introduction to PHP

In this lecture, we got our first taste of PHP by creating a **test.php** file in the root directory of our WordPress installation. In PHP, we are allowed to write HTML. As long as you know HTML, you've got the first few steps covered for writing PHP.

We were able to access the file via an HTTP URL. By installing a web server, like Nginx, it'll allow us to access the files in our project. Before a file is sent over to the browser, the server will process the PHP code in a file. This is why we're allowed to access PHP files from the browser.

Variables

In this lecture, we explored the first feature of PHP, which are variables. A variable can be thought of as a container for your data. You can store any type of data from numbers, names, file data, etc. In our PHP file, we wrote the following code:

```
<?php

$age = 28;

?>
```

We can write PHP code by adding a pair of PHP tags. This is where PHP starts to become different from HTML. Inside the PHP tags, we can start giving instructions to our machine. We are not allowed to write HTML. In addition, we're allowed to add as many PHP tags in our file as we'd like. It's common practice not to close a PHP tag if you don't intend on writing HTML after the PHP tag.

Variables

Variables can be declared by typing the `$` symbol followed by the name. There are rules for PHP names. They're the following:

- A variable starts with the $ sign, followed by the name of the variable

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive ($age and $AGE are two different variables)

Syntax

There are rules for writing PHP code called its syntax. Just like the English language, we must adhere to these rules. One of the rules is to end every line of code with a `;` character. This indicates we're finished with a line of code. PHP will run the next line of code.

Outputting Variables

We can output a variable with the following code:

```
echo $age;
```

The `echo` keyword instructs the language to output the value that's written after it. In this example, the `$age` variable will get outputted. PHP will search for this variable and output the value assigned to the variable. If we didn't include this line of code, nothing would get added to our HTML document.

Strings and Booleans

In this lecture, we discussed the concept of data types. A data type is the category system PHP uses for our variable's value. This system is completely automatic. PHP will automatically categorize the type of data in your variable. Here are the following data types:

- Integer
- Float
- String
- Boolean
- Array
- Object
- Null

Null

The `null` data type is a special type of data type for variables that aren't declared with a value. It's a great data type to use when you want to have a variable available without giving it a value.

String

The `string` data type is used for storing text. We can store names, addresses, gibberish, or anything that we can type on our keyboard. Strings can be created with single quotes or double-quotes. Here's an example of both syntaxes

```
$name = 'Luis';
$name = "Luis";
```

Booleans

Booleans are another type of data type that can only have two possible values, `true` or `false`. They're a useful feature for telling your program if something should be on or off. We'll come across practical situations throughout the course. Here's an example of both values:

```
$isLoggedIn = true;
$isLoggedIn = false;
```

Notice how we're using multi-worded variable names. There are two naming conventions for multi-worded variables in PHP called camel casing or snake casing. Camel casing is when every word in a variable is capitalized except the first word. Snake casing is when each word is separated with underscores.

Either convention is valid. It's all preference. Whichever you use, be consistent.

Functions

In this lecture, we learned about functions for reusing blocks of code. It's inefficient to copy and paste code. Rather than doing that, we can define a function once and reuse it as many times we'd like. Here's what a function definition looks like:

```
function greeting() {
  echo 'Hello';
}

greeting();
```

Functions are defined with the `function` keyword. Code inside the curly brackets makes up the function's body. By default, PHP will not run the code

inside a function. We must call it. Calling a function can be done by writing its name followed by parentheses.

Parameters/Arguments

Functions can have local variables called parameters. They can be added to a function's parentheses like so:

```
function greeting($message) {
  echo $message;
}

greeting('hello');
```

This feature can be useful for passing on custom data from the function call to the function. In this example, the string, `'hello'`, will be used as the value for the `$message` variable.

Arrays

In this lecture, we talked about another feature of PHP called arrays. They're a feature for storing a collection or group of data. Rather than creating multiple variables for storing a collection of data, we can define a single variable. Arrays are created with the `array()` function:

```
$food = array('pizza', 'tacos', 'hamburger');
```

Items in an array can be accessed by their index. Indexes are zero-based. This means the first item in the array has an index of `0`. The second item in the array has an index of `1`. So on and so forth. We can output a specific item in the array with the following syntax:

```
echo $food[0];
```

Loops

In this lecture, we took the time to look at how we can loop through an entire collection of arrays with loops. Outputting an individual item in an array can be useful. However, what if you want to output the whole array? It would be tedious to write multiple `echo` statements. Luckily, PHP has a feature called loops for iterating through an array.

```php
$count = 0;

while($count < count($food)) {
  echo $food[$count];

  $count = $count + 1;
}
```

In this example, we're looping through the `$food` array with the `while()` loop. The `while()` loop must be passed in a condition inside its parentheses. In this example, we're checking if the `$count` variable is less than the items in the `$food` array. We're counting the items in the array with the `count()` function, which is another PHP defined functions. If this condition evaluates to `true`, the code inside the block will execute.

Inside the curly brackets, we are outputting the item in the array by using the `$count` variable. After outputting the item, we are incrementing the `$count` variable by one.

Alternative syntax

Alternatively, you can use the increment operator to update a numeric value by one like so:

```php
$count++;
```

An alternative syntax for creating arrays is to use square brackets:

```php
$food = ['pizza', 'tacos', 'hamburger'];
```

Either syntax is valid. Beginner developers are encouraged to use the `array()` function over the square bracket syntax to make it easier to identify an array declaration.

Resources

- Comparison Operators

Constants

A constant is an immutable variable. Its value can never change after it's been defined. Constants can be defined with the `define()` function. It has two arguments, which are the name and value. Here's an example of a constant.

```php
define("NAME", "Luis");

echo NAME;
```

It's common practice for constant names to have all uppercase letters to differentiate them from regular variables.

Understanding Errors

In this lecture, we quickly reviewed the anatomy of an error. Let's take the following example:

```php
define('NAME', 'Luis');
define('NAME', 'test');

echo NAME;
```

It'll produce the following error in the browser:

```
Warning: Constant NAME already defined in test.php on line 2
```

The error produced by PHP tells us the following:

- **Type** - The type of error. Warnings will not stop the script from running.
- **Description** - A summary of the error and why PHP couldn't properly process your code.
- **File Location/Line Number** - Self-explanatory, the location where the error was produced.

With this critical information, you'll be able to quickly resolve your issues by carefully reading the error. Life won't always be this easy, but it's the best place to start when you run into trouble.

Comments

In this lecture, we quickly took a look at PHP comments, which serve the same purpose as HTML and CSS comments. They allow us to add notes and documentation to our code. PHP has two types of comments.

Single-line comments allow us to add a comment to a single line. Subsequent lines are unaffected.

```php
// Single-line example
```

Multiline comments allow us to write multiple lines of comments.

```
/*
 Multiline
 Example
*/
```

In some cases, developers may add an `*` on every line. This does not affect the comment. It's just a style preference.

```
/*
 * Multiline
 * Example
 */
```

WordPress and PHP

In this lecture, we talked about what's coming next in the course. The content for this video is meant to be consumed via video. There are no lecture notes.

Kickstarting a Theme

In this section, we got started with developing a custom for WordPress by bringing the static template to life with WordPress defined functions.

Exploring the WordPress Configuration

In this lecture, we took the time to discover what was inside WordPress's configuration file. WordPress has a configuration file for storing a site's database details, behavior, and security keys. These settings can be found in a file called **wp-config.php**.

WordPress uses constants for creating its configuration settings. In addition, constant names are written with all uppercase letters. It's a standard convention.

Adjusting the Configuration

In this lecture, we updated the configuration of WordPress by turning debug mode on. By turning this option on, WordPress will output error messages to help you debug your site during development. You can turn on debug mode by setting the `WP_DEBUG` constant to `true`.

```
define( 'WP_DEBUG', true );
```

Another setting worth turning on is called `WP_DISABLE_FATAL_ERROR_HANDLER`. By default, WordPress will produce a blank white page if a fatal error is encountered. WordPress can override this behavior by producing a friendlier page. By default, this behavior is turned off. You can turn it on by setting this constant to `true`.

```
define('WP_DISABLE_FATAL_ERROR_HANDLER', true);
```

Security Keys

In addition, the configuration file contains security keys for securely logging users in. In case of an emergency, you should reset these values. If you can't change these values, WordPress has a tool for generating new keys here

WordPress Files and Folders

In this lecture, we took the time to explore the files and folders that can be found in a standard WordPress installation. It can be overwhelming, but you won't have to deal with most of the files. Here's a summary of each directory

- **Root Directory:** These files will take care of initializing WordPress.
- **wp-admin (folder):** These files will take care of loading the admin dashboard.
- **wp-content (folder):** These files contain plugin, theme, and media files.
- **wp-includes (folder):** These files contain additional code for helping WordPress perform additional actions.

In most cases, you will be working inside the **wp-content** folder for creating the theme and plugin. You should never modify files outside this directory. Otherwise, your changes may be overridden. The **wp-config.php** file is an exception.

File Headers

In this lecture, we created a file header for our theme. WordPress has three requirements for creating a theme.

1. index.php
2. style.css
3. File Header

It's recommended you create your themes in the **wp-content/themes** directory. A separate folder should host your theme's files. For this course, we created a folder called **udemy**.

As for the file header, this should be created in the **style.css** file. File headers provide information to WordPress on your theme's details, such as the name, author, and license. Here's an example of a file header.

```
/*
Theme Name: Twenty Twenty
Theme URI: https://wordpress.org/themes/twentytwenty/
Author: the WordPress team
Author URI: https://wordpress.org/
Description: Our default theme for 2020 is designed to take full adva
Tags: blog, one-column, custom-background, custom-colors, custom-logo
Version: 1.3
Requires at least: 5.0
Tested up to: 5.4
Requires PHP: 7.0
License: GNU General Public License v2 or later
License URI: http://www.gnu.org/licenses/gpl-2.0.html
Text Domain: twentytwenty
This theme, like WordPress, is licensed under the GPL.
Use it to make something cool, have fun, and share what you've learne
*/
```

Additional File Headers

In this lecture, we began adding headers for WordPress to learn more about our theme. We added the following info:

```
/*
Theme Name: Udemy
Theme URI: https://udemy.com
Author: Udemy
Author URI: https://udemy.com
Description: A simple WordPress theme.
Version: 1.0
Requires at least: 5.8
Tested up to: 6.0
Requires PHP: 8.0
Text Domain: udemy
*/
```

Here's a complete list of headers you can add to a theme.

- **Theme Name**: Name of the theme.
- **Theme URI**: The URL of a public web page where users can find more information about the theme.
- **Author**: The name of the individual or organization who developed the theme. Using the Theme Author's wordpress.org username is recommended.
- **Author URI**: The URL of the authoring individual or organization.
- **Description**: A short description of the theme.
- **Version**: The version of the theme, written in X.X or X.X.X format.
- **Requires at least**: The oldest main WordPress version the theme will work with, written in X.X format. Themes are only required to support the three last versions.
- **Tested up to**: The last main WordPress version the theme has been tested up to, i.e. 5.4. Write only the number, in X.X format.
- **Requires PHP**: The oldest PHP version supported, in X.X format, only the number
- **License**: The license of the theme.
- **License URI**: The URL of the theme license.
- **Text Domain**: The string used for textdomain for translation.
- **Tags**: Words or phrases that allow users to find the theme using the tag filter. A full list of tags is in the Theme Review Handbook. Domain Path: Used so that WordPress knows where to find the translation when the theme is disabled. Defaults to /languages.

Say Hello to Full-Site Editing

In this lecture, we talked about the future of WordPress, which is full-site editing. Previously, you had to install a 3rd party package to build a site. While they were great, they did come with some downsides.

- 💰 Expensive
- 🧨 Gotta manage licenses
- 😕 Not Standardized

WordPress saw an opportunity to resolve these issues by extending the Gutenberg editor from a post content editor to a full-blown page builder. With the introduction of full-site editing (FSE), WordPress resolves the issues of past page builders. Completely free, no licenses, and standardized methodology of building pages.

Love it or hate it, FSE is here to stay. This course will cover full-site editing to its fullest extent. Don't worry, classic-site editing will be covered too in a future section.

The Index Template

In this lecture, we started to explore the index template. We have two options for creating the index template. The first option is to create an **index.php** file in the root directory of our theme. The second option is to create a file called **index.html** inside a directory called **templates**.

The HTML file has priority over the PHP file. If the HTML file does not exist, WordPress will fall back to the PHP file. By adding an **templates/index.html** file, WordPress will switch over to full-site editing.

You should keep in mind that full-site editing only supports blocks. If you were to attempt to write plain HTML, WordPress will throw an editor from within its editor.

Replicating the Index Template

In this lecture, we're going to recreate the base template for the **index.html** file. Previously, we would need to manually create a base template before working on a theme. With full-site editing, this process is expedited for you. Here's what the code that generates the base template looks like:

```
<!DOCTYPE html>
<html <?php language_attributes(); ?>>
<head>
  <meta charset="<?php bloginfo( 'charset' ); ?>" />
  <?php wp_head(); ?>
</head>
```

```
<body <?php body_class(); ?>>
<?php wp_body_open(); ?>

(template)

<?php wp_footer(); ?>
</body>
</html>
```

We're going to recreate this template so that we can fully understand what's going on behind the scenes. This is also a great opportunity to practice writing PHP.

Language Attribute

In this lecture, we dynamically generated the `lang` attribute. There are a couple of benefits to adding this attribute to your HTML document.

1. Helps search engines identify the language of your site.
2. Helps screen readers will be able to pick a language profile with the correct accent and pronunciation.
3. Can be used for styling a page based on a language.

For example, let's say a site is in Arabic; text should appear from right to left. CSS developers would use the following to change the direction:

```
:lang(ar) {
  direction: rtl;
}
```

Without the `lang` attribute set to the correct language code, the above code snippet would not work. For these reasons, you should always add the `lang` attribute to your documents.

Hardcoding this attribute is not recommended. Your theme should be compatible with various languages. Luckily, WordPress has a function called `language_attributes()` for checking the language of the current WordPress installation and outputting it onto the document. Here's an example:

```
<html <?php language_attributes(); ?>>
```

Resources

- Language Codes
- `language_attributes()` Function

Character Set

In this lecture, we dynamically loaded the character set of the document by using the `bloginfo()` function. WordPress stores the character set in the database. This function allows us to grab that information along with other pieces of information and displays it on the page.

You can refer to the documentation for a list of possible values. In our case, we can pass in the value `'charset'` to instruct this function to get the character set for the current site. Here's an example:

```
<meta charset="<?php bloginfo('charset'); ?>">
```

Resources

- `bloginfo()` Function

Loading Additional Tags

In this lecture, we began giving WordPress and 3rd party plugins locations to load additional content. By default, WordPress does not scan your document for locations to inject tags and content. We must provide a location.

There are two functions that we should use called `wp_head()` and `wp_footer()`. The `wp_head()` function can be added at the end of the `<head>` section of a document. Here's an example:

```
<head>
  <meta charset="<?php bloginfo('charset'); ?>">
  <?php wp_head(); ?>
</head>
```

It's recommended that you add this function after you've added tags for your theme. Your theme's tags should have priority over external tags.

As for the `wp_footer()` function, this function should be called before the closing `<body>` tag. Here's an example:

```
<body>
  ...

  <?php wp_footer(); ?>
</body>
```

Same as before, this function will allow plugins and the WordPress core to inject content at the bottom of the document.

The Body Tag

In this lecture, we got started with adding additional functions for modifying the `<body>` tag of the document. The first function is called `body_class()`. This function will add a series of classes to the `<body>` tag. We even have the option of adding our own classes. Here's an example:

```
<body <?php body_class(); ?>>
```

The second function is called `wp_body_open()`. This function will allow the WordPress core and 3rd party plugins to inject content into the opening tag of the `<body>` tag. This is the complete opposite of the `wp_footer()` function, which will inject content at the bottom of the `<body>` tag. Here's an example

```
<body>
  <?php wp_body_open(); ?>
</body>
```

Getting to know WordPress

In this lecture, we got to know WordPress by exploring the various resources at our disposal. Here's a rundown of the resources.

- **wordpress.com** - A hosting platform for WordPress. Buy servers, connect your domains, and manage your sites.
- **wordpress.org** - Hosts the original source code of WordPress along with basic usage information.
- **developer.wordpress.org** - A resource dedicated to programmers for learning about WordPress's functions and classes along with guides for developing themes/plugins.
- **codex.wordpress.org** - A resource for technical and non-technical users. It was the original resource for developers but is currently being moved over to **developer.wordpress.org**.

Resources

- Official WordPress Site (.org)
- WordPress Hosting Platform
- Developer Documentation
- Codex
- Automaticc

WordPress Coding Standards

In this lecture, we talked about WordPress's coding standards. These standards are completely optional unless you're planning on contributing to the original source code of WordPress. If you're planning on solely developing plugins and themes, you can skip these standards.

However, you should try your best to adhere to these standards. They can reduce confusion among teams and make the development cycle easier.

Resources

- Coding Standards

Global Styles

In this section, we modified WordPress's global styles to enhance the editing experience for our theme.

Aside: JSON

In this lecture, we took a moment to talk about JSON. If you're an experienced programmer, you can skip this lecture. Otherwise, let's get into it.

JSON was introduced as a way to store data in a separate file. It stands for JavaScript Object Notation. Heavily inspired by JavaScript but not required to know.

JSON Supports 5 data types: strings, numbers, booleans, arrays, and objects. Here's an example of some JSON code.

```
{
  "name": "John",
  "age": 20,
  "isHungry": true,
  "hobbies": ["hockey", "baksetball"],
  "job": {
    "company": "Udemy"
  }
}
```

We learned about a new data type called objects. The object data type allows us to group pieces of information together. A similar concept to array. The main difference is that objects follow a key-value format whereas arrays can just contain values.

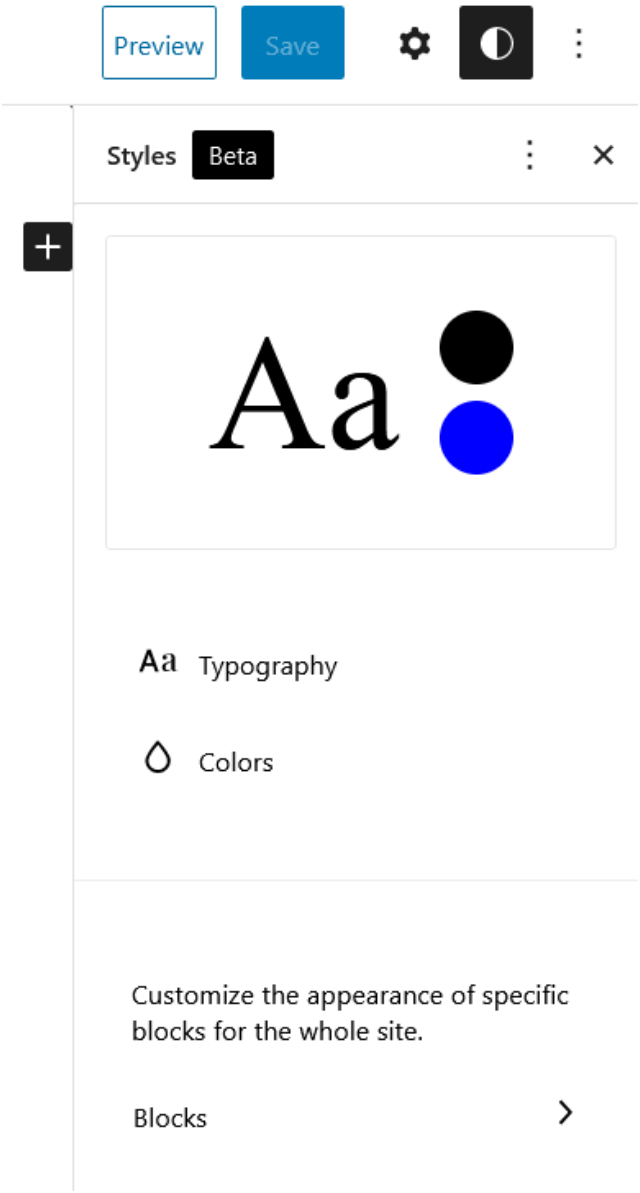
Resources

- JSON Playground

Introduction to Global Styles

In this lecture, we learned about global styles. The global styles sidebar is a fairly new feature of WordPress 5.9. It allows users to modify the theme's appearance on a global evel.

We can access the global styles in the full-site editor at the top left corner:

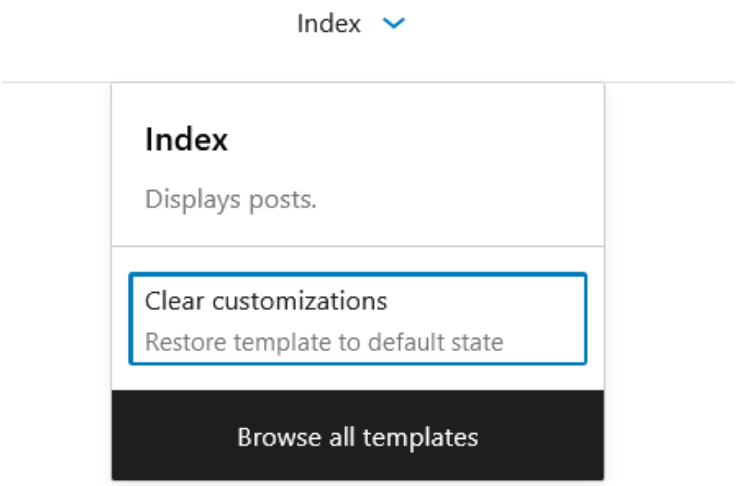


WordPress allows theme developers to modify the features available in the global styles through a file called **theme.json**. This file is optional, but if provided, WordPress will load its configuration settings.

Resetting the template

Any modifications you make to the template will not directly affect the template in your theme. However, a copy will be stored in the database, which will have priority over your theme's template.

In these cases, you can revert back to your template by clicking on the button at the top center of the page in the editor.



Loading a Schema

In this lecture, we added a schema to our **theme.json** file to help with debugging our file. This step is optional but extremely helpful for debugging JSON files. A schema is a link to a file that describes the structure of a JSON file. It'll help you correctly name your properties and set their values.

Schemas can be added by adding the `$schema` property. The value should be a URL to the schema. Here's an example of our schema:

```
{
  "$schema": "https://schemas.wp.org/trunk/theme.json",
  "version": 2
}
```

By adding this property to your JSON file, your editor should tell you that the `version` property is missing and that it should be an integer.

It's possible that you may get an error after adding the schema. If the error states the certificate has expired, try the following fixes:

- Change protocol from `https` to `http`.
- Change the URL to the following: https://raw.githubusercontent.com/WordPress/gutenberg/trunk/schemas/json/theme.json

Resources

- Schema

Default Color Palette

In this lecture, we disabled the color palette on a global level. The default color palette is a series of colors suggested by WordPress. If you're developing a theme, you may want to recommend your own set of colors. In that case, you might want to hide WordPress's recommendations.

The default palette can be toggled by setting the `color.defaultPalette` property. For example, here's how you would disable the default palette on a global level

```
{
  "settings": {
    "color": {
      "defaultPalette": false
    }
  }
}
```

Resources

- Global Styles
- Theme.json Reference

Block Color Settings

In this lecture, we enabled the default palette for the **Site Title** block. Block settings have priority over global settings. You can grab the name of the block from the developer resources. Check out the resource section for the link.

```
{
  "settings": {
    "blocks": {
      "core/site-title": {
        "color": {
          "defaultPalette": true
        }
      }
    }
  }
}
```

Keep in mind, that settings can only be enabled/disabled if the block originally supports the setting. If a block does not support a feature, forcibly enabling it will not do anything.

Resources

- Core Blocks Reference

Adding Colors to the Palette

In this lecture, we began adding colors to match our theme. WordPress allows developers to recommend colors to users to help them design their site. You will always want to do this so that your users can make easy choices. Colors can be added through the `settings.color.palette` property. Here's an example:

```
{
  "settings": {
    "color": {
      "palette": [
        {
          "slug": "u-white",
          "color": "rgb(255,255,255)",
          "name": "Udemy White"
        },
        {
          "slug": "u-gray-100",
```

```
        "color": "rgb(243 244 246)",
        "name": "Udemy Gray 100"
      },
      {
        "slug": "u-gray-200",
        "color": "rgb(229 231 235)",
        "name": "Udemy Gray 200"
      },
      {
        "slug": "u-gray-300",
        "color": "rgb(209 213 219)",
        "name": "Udemy Gray 300"
      },
      {
        "slug": "u-gray-400",
        "color": "rgb(156 163 175)",
        "name": "Udemy Gray 400"
      },
      {
        "slug": "u-gray-500",
        "color": "rgb(107 114 128)",
        "name": "Udemy Gray 500"
      },
      {
        "slug": "u-gray-600",
        "color": "rgb(75 85 99)",
        "name": "Udemy Gray 600"
      },
      {
        "slug": "u-gray-700",
        "color": "rgb(55 65 81)",
        "name": "Udemy Gray 700"
      },
      {
        "slug": "u-gray-800",
        "color": "rgb(31 41 55)",
        "name": "Udemy Gray 800"
      },
      {
        "slug": "u-gray-900",
        "color": "rgb(17 24 39)",
        "name": "Udemy Gray 900"
      },
      {
        "slug": "u-primary",
        "color": "rgb(239 68 68)",
        "name": "Udemy Primary"
      },
      {
        "slug": "u-accent",
        "color": "rgb(67 56 202)",
        "name": "Udemy Accent"
      },
      {
        "slug": "u-success",
        "color": "rgb(52 211 153)",
        "name": "Udemy Success"
      },
      { "slug": "u-info", "color": "rgb(96 165 250)", "name": "Udemy Info" },
      {
        "slug": "u-warning",
        "color": "rgb(251 146 60)",
        "name": "Udemy Warning"
      },
      {
        "slug": "u-danger",
        "color": "rgb(236 72 153)",
        "name": "Udemy Danger"
      }
    ]
  }
```

```
  }
}
```

The `palette` property should be an array of objects. In each object you can add the following properties:

- `slug` - A unique ID for the color. It's recommended to add a prefix to help WordPress identify your colors among its own preset and plugin colors.
- `color` - A valid CSS color value. (Hex, RGB, RGBA, Color Names, Keywords, HSL, HSLA)
- `name` - A human readable name that will be displayed on the front end.

We can even add colors to specific blocks. Here's an example that adds the color pink to the **Site Title** block.

```
{
  "settings": {
    "blocks": {
      "core/site-title": {
        "color": {
          "palette": [
            { "slug": "u-pink", "color": "#f1c0e8", "name": "Udemy Pink" }
          ]
        }
      }
    }
  }
}
```

Keep in mind, WordPress will prioritize block-specific settings over global settings. This palette will override the global palette.

Resources

- Core Blocks Reference

Colors for backgrounds, text, and links

In this lecture, we learned how to enable colors for backgrounds, text, and links. The **theme.json** file can be updated to enable/disable these options in the editor on a global level or block level. Here's an example on both global/editor levels.

```
{
  "settings": {
    "color": {
      "background": true,
      "link": true,
      "text": true
    },
    "blocks": {
      "core/site-title": {
        "color": {
          "background": true,
          "link": true,
          "text": true
        }
      }
    }
  }
}
```

Custom Colors

In this lecture, we learned how to disable the color picker for disabling custom colors for blocks and the global styles sidebar. The `custom` property can be added to the `colors` object for either global styles or blocks. Here's an example of both:

```
{
  "settings": {
    "color": {
      "custom": true
    },
    "blocks": {
      "core/site-title": {
        "color": {
          "custom": true
        }
      }
    }
  }
}
```

Duotones

In this lecture, we learned how to apply duotones to images. A duotone is a filter that changes the color of an image with a shadow and highlight. It's similar to an Instagram filter but with one distinct difference. Instagram filters modify the color levels, whereas a duotone completely overrides the colors of an image.

Duotones are an experimental feature. There are some bugs, so be careful when using them. We can apply duotones to images and videos. Here's an example of how to add duotones to a theme.

```
{
  "settings": {
    "color": {
      "duotone": [
        {
          "slug": "u-pink-sunset",
          "colors": ["#11245E", "#DC4379"],
          "name": "Udemy Pink Sunset"
        }
      ]
    }
  }
}
```

A duotone can be added by adding a series of objects to the `duotone` array. The format for each object is the following:

- `slug` - A ID for the duotone
- `colors` - An array of colors where the first color is the shadow (dark) and the second color is the highlight (light)
- `name` - Human-readable name of the block

In some cases, you may want to disable the duotone feature for a block. You can do so by setting the `duotone` property to an empty array. WordPress will not recommend presets by doing so. However, the editor will allow users to select custom colors. We can disable custom selection by adding the `customDuotone` property to false. Here's an example of disabling duotones for the cover block.

```
{
  "settings": {
    "blocks": {
      "core/cover": {
        "color": {
          "duotone": [],
          "customDuotone": false
        }
      }
    }
  }
}
```

Gradients

In this lecture, we explored how to add custom gradient presets to our theme. Firstly, you may want to disable WordPress's default preset of gradients by setting the `defaultGradients` option to `false` like so.

```
{
  "settings": {
    "color": {
      "defaultGradients": false
    }
  }
}
```

Gradients can be added through the `gradients` array under the `colors` object. The format for adding gradients is the following:

- `slug` - A unique ID for the gradient
- `gradient` - A valid CSS value for the gradient
- `name` - A human readable name

```
{
  "settings": {
    "color": {
      "gradients": [
        {
          "slug": "u-summer-dog",
          "gradient": "linear-gradient(#a8ff78, #78ffd6)",
          "name": "Udemy Summer Dog"
        }
      ]
    },
    "blocks": {
      "core/site-title": {
        "color": {
          "gradients": [],
          "customGradient": false
        }
      }
    }
  }
}
```

In some cases, you may want to completely disable the gradients option. There are two settings you will need to add. Firstly, the `gradients` property should be an empty array to prevent WordPress from recommending colors. Secondly, the `customGradient` property must be set to `false` to prevent users from selecting colors. Here's an example of disabling the gradients option from the **Site Title** block.

```
{
  "settings": {
    "blocks": {
      "core/site-title": {
        "color": {
          "gradients": [],
          "customGradient": false
        }
      }
    }
  }
}
```

Applying Colors

In this lecture, we began applying colors to our theme. We're not limited to adding options to WordPress styles; we can apply them to blocks immediately. To get started, we changed the text and background colors of our theme. Keep in mind, these colors can be modified. We're just setting the initial colors.

We can modify the styles by adding the `styles` object to our JSON file. This object is separate from the `settings` property, which is primarily used for enabling/disabling features.

```
{
  "styles": {
    "color": {
      "text": "var(--wp--preset--color--u-gray-700)",
      "background": "var(--wp--preset--color--u-white)"
    }
  }
}
```

In the above example, we're changing the colors of the text and background to CSS variables. Any valid CSS value will work.

WordPress will store custom colors in variables. Their values are extracted for further use in our CSS or options. Colors are prefixed with `--wp--preset--color--` followed by the slug of the color.

Duotones and gradients are also converted to variables. The prefix for duotones is `--wp--preset--duotone--`. The prefix for backgrounds are `--wp--preset--gradient--`.

Applying Gradients

In this lecture, we learned how to apply gradients to a block. As a reminder, the prefix for gradients is `--wp--preset--gradient--` followed by the slug. Styles can be applied to a block. For this example, we applied the gradient to the button block. Here's how we did it.

```
{
  "styles": {
    "blocks": {
      "core/button": {
        "color": {
          "gradient": "var(--wp--preset--gradient--u-summer-dog)"
        }
      }
    }
  }
}
```

Similar configuring the settings for a block. The `blocks` object can be added to the `styles` object with the name of the block as a child. The gradient can be configured through the `gradient` property.

Applying Styles to Elements

In this lecture, we can change the styles of elements through the `elements` property in the `styles` object. We're not limited to global or block styles. Specific elements can be styled. At the moment, we're limited to updating links or heading tags(h1,h2,h3,etc).

```
{
  "styles": {
    "elements": {
      "link": {
        "color": {
          "text": "var(--wp--preset--color--u-primary)"
        }
      }
    }
  }
}
```

In this example, we're changing the color for the `link` element, which is the selector for `<a>` tags. For headings, we must use `h1`, `h2`, etc.

Borders

In this lecture, we learned how to add support for borders and styling the borders. By default, changing the borders for most blocks is disabled. However, you have the option, as a theme developer, to change these settings. CSS has various properties for borders. Since that's the case, WordPress has different options for borders.

```
{
  "settings": {
    "border": {
      "color": true,
      "radius": true,
      "style": true,
      "width": true
    }
  }
}
```

Border settings can be enabled by adding the `border` object to the `settings` object. However, you may enable/disable this feature for specific blocks too. There are four options we can enable/disable. They're the color, radius, style, and width of a border.

After making those changes, you have the option of setting a default border through the `styles` property like so.

```
{
  "styles": {
    "blocks": {
      "core/pullquote": {
        "border": {
          "width": "2px",
          "radius": "10px",
          "style": "solid",
          "color": "var(--wp--preset--color--u-primary)"
        }
```

```
        }
      }
    }
}
```

In this example, we're modifying the pull-quote block. The values for the width, radius, style, and color must be valid CSS values.

Font Families

In this lecture, we started adding custom font families to the editor. We added two fonts called **Rubik** and **Pacifico**. The **Rubik** font is available as a global option. We can add font families by adding the `typography` object to the `settings` object. In your file, add the following:

```
{
  "settings": {
    "typography": {
      "fontFamilies": [
        {
          "fontFamily": "Rubik, sans-serif",
          "slug": "u-rubik",
          "name": "Udemy Rubik"
        }
      ]
    }
  }
}
```

The `fontFamilies` is an array of objects where each object represents a font. There are three properties that must be added for each font

- `fontFamily` - A valid CSS value for the font family name. Can be multiple fonts (recommended)
- `slug` - A unique ID for the font
- `name` - A human-readable name for the font that will be displayed in the editor.

Alternatively, you can modify the font family options for a specific block. Here's an example of a set of custom font options for the **Site Title** block.

```
{
  "settings": {
    "blocks": {
      "core/site-title": {
        "typography": {
          "fontFamilies": [
            {
              "fontFamily": "Pacifico, sans-serif",
              "slug": "u-pacifico",
              "name": "Udemy Pacifico"
            }
          ]
        }
      }
    }
}
```

The `typography` object can be added to any blocks with the same settings for the global-level settings. Keep in mind, the font family for blocks will override the global blocks even if they're custom font families. If you would like the same fonts to appear, you will need to read them for the block.

This solution is not complete. You must also load the font families, which are not supported through the **theme.json** file. This topic is covered in the next section. As long as the font families appear as an option in the sidebar, you're good to go.

Resources

- Pacifico
- Rubik

Font Sizes

In this lecture, we configured the font sizes for our theme. By default, WordPress will recommend default font sizes which can be overridden. However, we're gonna take a different approach by overriding the values instead of generating new sizes. WordPress generates four sizes, all of which are overridable. They're the following:

```
--wp--preset--font-size--small: 13px;
--wp--preset--font-size--medium: 20px;
--wp--preset--font-size--large: 36px;
--wp--preset--font-size--x-large: 42px;
```

We can override these values by updating the `theme.json` file with the following:

```
{
  "settings": {
    "typography": {
      "fontSizes": [
        { "slug": "small", "size": "0.75rem", "name": "Small" },
        { "slug": "medium", "size": "1.25rem", "name": "Medium" },
        { "slug": "large", "size": "2.25rem", "name": "Large" },
        { "slug": "x-large", "size": "3rem", "name": "Extra Large" },
        { "slug": "gigantic", "size": "3.75rem", "name": "Gigantic" }
      ]
    }
  }
}
```

New font sizes can be made by adding the `fontSizes` array. In this array, we must add objects that represent each font size. The following properties must be present.

- `slug` - The unique ID for the font size
- `size` - A valid CSS value for a font size
- `name` - A human readable name

Disabling Font Settings

In some cases, you may want to disable the font size options from the typography panel of a block. You can do so by setting the `fontSizes` property to an empty array. In addition, the `customFontSize` property must be set to `false` to disable the option for adding a custom font size. In this example, we're disabling the font size settings for the preformatted block.

```
{
  "settings": {
    "blocks": {
      "core/preformatted": {
        "typography": {
          "fontSizes": [],
          "customFontSize": false
        }
```

```
      }
    }
  }
}
```

Resources

- Standardizing Theme.json Font Sizes

Various Font Settings

In this lecture, we enabled various typography settings for our theme. Most of these settings are enabled, but we added them in for example purposes. All of these settings are toggleable from within the `typography` object.

```
{
  "settings": {
    "typography": {
      "lineHeight": true,
      "dropCap": true,
      "fontWeight": true,
      "fontStyle": true,
      "textTransform": true,
      "letterSpacing": true,
      "textDecoration": true
    }
  }
}
```

- `lineHeight` - Allow users to set custom line height.
- `dropCap` - Allows users to increase the font size of the first letter in a paragraph. (Only available for the paragraph block)
- `fontWeight` - Allow users to set custom font weights.
- `fontStyle` - Allow users to set custom font styles.
- `textTransform` - Allow users to set custom text transforms.
- `letterSpacing` - Allow users to set custom letter spacing.
- `textDecoration` - Allow users to set custom text decorations.

Applying Typography Styles

In this lecture, we began applying basic typography styles to our theme by adding the `typography` object to the `styles` property. Here's the code we added:

```
{
  "styles": {
    "typography": {
      "fontFamily": "var(--wp--preset--font-family--u-rubik)",
      "fontSize": "16px",
      "fontStyle": "normal",
      "fontWeight": "normal",
      "lineHeight": "inherit",
      "textDecoration": "none",
      "textTransform": "none"
    }
  }
}
```

Most of these are self-explanatory. For the `fontFamily` property, we're setting the value to a variable called `--wp--preset--font-family--u-rubik`. All custom font families will have

variables that are prefixed with `--wp--preset--font-family--` followed by the slug name.

In some cases, you may want to add font families for a specific block. Here's how we added one for the **Site Title** block.

```
{
  "styles": {
    "blocks": {
      "core/site-title": {
        "typography": {
          "fontFamily": "var(--wp--preset--font-family--u-pacifico)"
        }
      }
    }
  }
}
```

Content Width

In this lecture, we added content width settings for the layout of our theme. Without these settings, users may assume their content stretches the entire width of the page. The editor should accurately represent what the content will look like on the front end of your site. One way to do that is by setting the width. You can set the width by adding the `contentSize` property to your `layout` object like so:

```
{
  "settings": {
    "layout": {
      "contentSize": "840px",
      "wideSize": "1100px"
    }
  }
}
```

The `contentSize` property should be set to the maximum width of your content. The value must be a valid CSS value. In addition, you have the option of adding the `wideSize` property for supporting wide alignment. If added, WordPress will add the option to the alignment toolbar. The value for this property should be a size bigger than the `contentSize` property. The `wideSize` option will allow blocks to stretch outside the width of the content.

Margin and Padding

In this lecture, we enabled the option for modifying the margin and padding settings from the full-site editor. By default, these options are disabled. Inside the `settings` object, you can enable margin and padding by adding the `margin` and `padding` properties with a value of `true` like so:

```
{
  "settings": {
    "spacing": {
      "margin": true,
      "padding": true
    }
  }
}
```

You can also configure the margin and padding values for the document or for specific blocks. Here's an example of how we can apply margin and padding:

```
{
  "styles": {
    "spacing": {
      "margin": {
        "top": "0px",
        "right": "0px",
        "bottom": "0px",
        "left": "0px"
      },
      "padding": {
        "top": "0px",
        "right": "0px",
        "bottom": "0px",
        "left": "0px"
      }
    }
  }
}
```

Custom Units

In this lecture, we customized the units that users can select for customizing the margin and padding. You can add the `units` property to the `spacing` object to configure the units. Here's an example of configuring the units that exclude the `em` unit.

```
{
  "settings": {
    "spacing": {
      "units": ["px", "rem", "%", "vw", "vh"]
    }
  }
}
```

Block gaps

In this lecture, we enabled the block gap settings for blocks that add spacing content. For example, the columns block has spacing between each column. By default, these settings are disabled. You can enable them by adding the `blockGap` property to the `spacing` object.

```
{
  "settings": {
    "spacing": {
      "blockGap": true
    }
  }
}
```

You can configure the gap by updating the `blockGap` property to your desired space. The value must be a valid CSS value.

```
{
  "styles": {
    "spacing": {
      "blockGap": "4rem"
    }
  }
}
```

Enabling Everything

In this lecture, we quickly learned how to enable all settings with one single property. In your `settings` property, you can add the `appearanceTools` to enable everything. In some cases, it may be a good idea to enable everything all at once and then disable settings for specific blocks.

```
{
  "settings": {
    "appearanceTools": true
  }
}
```

Section 5: Managing Asset Files

In this section, we got started with loading asset files for our theme. It's one of the next steps you'll take to create a theme.

Aside: Git and GitHub

In this lecture, we got a brief introduction to Git and GitHub. What is Git? Git a version control system for recording changes to our codebase and maintaining a history of it. In addition, it makes collaboration easy by syncing projects across your team. There are various programs available for tracking the history of your code. Git is considered the most popular solution. Over 90% of devs use it.

So, what about GitHub? Git runs on your machine, but you may want to publish your code online. There are various platforms that have integration with Git. The most popular platform is GitHub.

Why use Git/GitHub?

Up until this point, programming has been pretty easy. Things are about to ramp up. If you run into problems with the course, you can ask for help in the Q&A section. I may ask you to your see your code. Udemy doesn't have the best tools for sharing code, so GitHub is the preferred way of sharing your work for me to check.

I highly recommend watching the video to learn how to upload your project to GitHub to share it with me.

Resources

- GitHub Desktop
- Git and GitHub Tutorial

Adding the Template

In this lecture, we got started with creating the theme by adding the HTML for the homepage. The **index.html** file is responsible for displaying the homepage on the front side of the site. For this example, we transferred the static template over to the index template.

Problems with hardcoding paths

Transferring the template was not completely foolproof. The template will be broken because the paths to the CSS and JS files are incorrect. The browser is unable to load these assets. We can fix this problem by updating the hardcoded paths, but that presents additional problems.

1. Users are able to change the WordPress directory structure, which can render absolute links useless.
2. Hard coding links doesn't allow for checking if SSL is enabled. (https:// & http://)
3. You'll end up loading all scripts and stylesheets on every page, even if you don't need them.

WordPress introduces some solutions for avoiding these issues, which we'll explore in the upcoming lecture.

Understanding Hooks

In this lecture, we discussed what hooks are. It's common to execute code during events. PHP does not have an event system built-in. Therefore, WordPress has its own event system called the hooks API. The hooks API will allow us to run functions during events.

Events can range from form submissions to sending emails. WordPress offers hundreds of hooks. On top of the hooks defined by WordPress, we can create custom hooks.

We're discussing hooks because WordPress offers a hook for loading stylesheets and script files. Let's try using the hook API for loading our assets.

The functions.php File

In this lecture, we created a file called **functions.php** inside the theme folder. The purpose of the **functions.php** file is to contain the logic of our theme. Its recommended to keep the logic and template of a theme separate. WordPress automatically runs the **functions.php** file. It'll always run before the **index.php** file.

Organization

We added a couple of sections to our **functions.php** file to keep things organized. We'll discuss each section as we need them. We're adding them ahead of time.

```
<?php
```

```
// Variables


// Includes


// Hooks
```

The Enqueue Hook

In this lecture, we added our first hook called `wp_enqueue_scripts`. It's recommended we register and load our scripts/styles during this hook. Below the `hooks` section in the **functions.php** file, we added the following code:

```
// Hooks
add_action('wp_enqueue_scripts', 'u_enqueue');
```

A hook can be registered by using the `add_action()` function. This function has two arguments, which are the name of the hook and the name of the function to run during the hook. In this example, we are prefixing the name of our function with the letter `u`, which is short for Udemy.

Prefixing function names is a common practice in the WordPress community. PHP will throw an error if two functions have the same name. It's possible site owners may install plugins, which can cause duplicate function names. To avoid this issue, it's recommended to prefix **ALL** your function names.

If you're new to PHP, functions can accept multiple values. If it does, like the `add_action()` function, you can separate each value with a comma.

Aside: PHP Function Return Values

In this lecture, we learned about return values. Functions can return values that can be stored in variables. This can be useful if you need to store a value from a function for future use. We can return values from a function with the `return` keyword. Here's an example:

```
function sum($a, $b) {
  return $a + $b;
}
```

One thing to keep in mind is that PHP will stop running functions if a value is returned. If you want to run additional logic, it should be performed before returning a value.

Including Files

In this lecture, we learned how to include PHP files. Rather than typing all your code in a single file, it's recommended to split your code into separate files for readability and management. Before using this function, we created a new directory:

- **includes/front** - Files for the front end

PHP has a function called `include()` that performs this task. It has one argument, which is the path to the file. First, we need to create a file that we want to include. We created a new file in the **includes/front** directory called **enqueue.php**. It contains the following code:

```
function u_enqueue() {

}
```

In the **functions.php** file, we include this file by using the `include()` function like so:

```
// Includes
include( get_theme_file_path('/includes/front/enqueue.php') );
```

We're using a function called `get_theme_file_path()` that's defined by WordPress. This function will generate a full path to the current activated theme. We can pass in an optional path relative to the theme directory that'll get appended to the final path.

By using the `include()` function, the contents of the **enqueue.php** file are added to the **functions.php** file, which gives us access to the `u_enqueue()` function.

Registering Styles

In this lecture, we registered styles by using a function called `wp_register_style()`. Registering a stylesheet will help WordPress become aware of the file, which can later be loaded into the document. We call this function inside the `u_enqueue()` function.

```
function u_enqueue() {
  wp_register_style(
    'u_font_rubik_and_pacifico',
    'https://fonts.googleapis.com/css2?family=Pacifico&family=Rubik:w
  );
  wp_register_style(
```

```
    'u_bootstrap_icons',
    get_theme_file_uri('assets/bootstrap-icons/bootstrap-icons.css')
  );
  wp_register_style(
    'u_theme',
    get_theme_file_uri('assets/public/index.css')
  );
}
```

The first two arguments of this function are required. Firstly, we must provide a handle name, which can be thought of as an ID for the file. Handle names should be prefixed since plugins are capable of registering styles too. To prevent naming conflicts, prefixing can avoid this issue.

The second argument must be a valid HTTP URL. In this example, we're loading files from external and local sources. To get a valid URL for a local source, you can use the `get_theme_file_uri()` function. This function is not to be confused with the `get_theme_file_path()` function, which returns a system path. The `get_theme_file_uri()` functions returns an HTTP URL.

It's advantageous to use this function because it'll always point to the correct directory and check for SSL. It has an optional argument, which is a path relative to the theme directory. In this example, we're passing in a path to a CSS file.

Resources

- wp_register_style() Function

Enqueuing Styles

In this lecture, we queued styles by calling the `wp_enqueue_style()` function. If a file is registered, WordPress will become aware of it. It doesn't do anything with the file. It just stores the location of the file in memory. Queuing a file will tell WordPress to load the file in the browser. Calling the `wp_enqueue_style()` function is the next step after registering a style.

At the end of the `u_enqueue()` function, we're calling this function to queue all our stylesheets.

```
wp_enqueue_style('u_font_rubik_and_pacifico');
wp_enqueue_style('u_bootstrap_icons');
wp_enqueue_style('u_theme');
```

This function requires the handle name of the file that should be loaded. It should correspond to the value passed into the first argument of the `wp_register_style()` function.

Resources

- wp_enqueue_style() Function

Fixing the Google Font

In this lecture, we fixed our Google Font. WordPress is modifying the URL passed into the `wp_register_style()` function. This is because WordPress is adding a query parameter called `ver`, which filters existing query parameters. If duplicate query parameters exist, the duplicate is removed. Therefore, a font family gets removed.

A query parameter is a feature for adding values to the URL. This is useful for sending data to the server without the user having to submit a form. Query parameters are added to a URL by adding a `?` character followed by key-value pairs. Multiple key-value pairs can be separated with the `&` character.

Example: **example.com?key=value&anotherKey=anotherValue**

We can fix this issue by preventing the `ver` parameter from being added. The 4th argument of the `wp_register_style()` function is a custom version. You can set this to a string. Alternatively, you can pass in `null` to disable WordPress from adding a parameter. Therefore, our parameters are never filtered. Here's the updated registration:

```
wp_register_style(
  'u_font_rubik_and_pacifico',
  'https://fonts.googleapis.com/css2?family=Pacifico&family=Rubik:wgh
  [],
  null
);
}
```

Loading Additional Head Tags

In this lecture, we added pre-connect tags to the `<head>` section of the document. This is to boost the performance of our site by telling the browser to connect to Google's servers before it needs to. Additional tags can be added to the document by running a function during the `wp_head` hook like so:

```
include( get_theme_file_path('/includes/front/head.php') );

add_action('wp_head', 'u_head', 5);
```

We are defining the function. Before looking at the definition, we are taking advantage of the third argument of the `add_action()` function, which is the priority. We are running the function early so that the tags can appear earlier than the style links.

```
function u_head() {
  ?>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin
  <?php
}
```

In our function, we're outputting the tags by leaving PHP mode. This is preferable to using the `echo` statement so that we can benefit from syntax highlighting.

Section 6: Templates

In this section, we got started with creating templates for our theme, from the home page to single posts.

Creating a Header and Footer Template Part

In this lecture, we got started with developing our theme's templates by splitting our index template into separate files. WordPress has support for creating sections of a template called a template part. In these template parts, we can add blocks that will be included with a template.

The main benefit of separating your code into separate files is that it makes it easier to read and maintain. Template parts can also be used in multiple templates.

The first step is to create a folder called **parts** in your theme folder. By creating template parts in this folder, WordPress will autoload them when users try adding template blocks to their template.

Afterward, we created two template parts called **header.html** and **footer.html**. You can use whatever filename you prefer. However, it's recommended to keep your names short, concise, and descriptive. Inside these files, you must insert blocks, not raw HTML.

Here are the blocks we added to the **header.html** file.

```
<!-- wp:html -->
<header class="shadow">...</header>
<!-- /wp:html -->
```

Here are the blocks we added to the **footer.html** file.

```
<!-- wp:html -->
<footer class="bg-gray-700 p-16">...</footer>
<!-- /wp:html -->
```

Lastly, the **index.html** file was updated to include these template parts.

```
<!-- wp:template-part {"slug":"header","theme":"udemy"} /-->

<!-- wp:html -->
<!-- Main Content -->
<main class="container !mx-auto pb-5 mb-4 mt-12">...</main>
```

```
<!-- /wp:html -->

<!-- wp:template-part {"slug":"footer","theme":"udemy"} /-->
```

We're using the **template part** block to help us load our template parts into the template.

Brief Anatomy of a Block

In this lecture, we briefly looked at the markup of a block. There are two types of blocks you'll come across which are blocks with content and blocks without content. A block with content will look like this:

```
<!-- wp:html -->
<header class="shadow">...</header>
<!-- /wp:html -->
```

A block is surrounded by HTML comments that contain the name of the block. The closing block has `/` character before the name to indicate the end of the block. Inside the block, you will find static content that will be rendered in the browser.

Blocks without content look like this:

```
<!-- wp:template-part {"slug":"header","theme":"udemy"} /-->
```

Same as before, the comment contains the name of the block. However, the comment is self-closing by adding the `/` at the end of the comment. In addition, the block has its settings stored in a JSON object. Normally, you will never need to edit this info. In most cases, you can edit these settings from the editor.

Optimizing the Header and Footer

In this lecture, we optimized the header and footer by updating the template parts. Template parts will surround your blocks with a tag that can be modified. Classes can be added to them as well. You can modify these settings from the editor.

We updated the **header.html** and **footer.html** template parts by letting the **template part** block generate the root element. Here's what the **header.html** file looks like:

```
<!-- wp:html -->
<!-- Topbar -->
<div class="py-3 text-sm bg-gray-700"></div>
<!-- Header midsection -->
<div class="container !mx-auto"></div>
<!-- Primary Menu -->
<div class="primary-menu" id="primary-menu"></div>
<!-- /wp:html -->
```

Here's what the **footer.html** file looks like:

```
<!-- wp:html -->
<div class="container !mx-auto"></div>
<!-- /wp:html -->
```

Last but not least, we updated the **index.html** file. The header block has changed to:

```
<!-- wp:template-part {"slug":"header","theme":"udemy","tagName":"hea
```

And the footer block has changed to:

```
<!-- wp:template-part {"slug":"footer","theme":"udemy","tagName":"foo
```

The classes and tag names will appear in both comments. WordPress will take care of the rest of applying these settings to the template part. By doing this, we reduce bloat in our document with unnecessary markup. Whenever possible, you should let WordPress generate the markup for your template instead of using raw HTML.

Adding Dummy Content

In this lecture, we began adding dummy content to our WordPress to help us with testing dynamic content. There are no lecture notes for this lecture. This content is meant to be consumed via video.

Creating the Topbar

In this lecture, we recreated the top bar of the header with WordPress's blocks. Instead of raw HTML, we should always attempt to recreate UI elements with blocks. This allows users to modify them without knowledge of HTML and CSS. To begin, we added the row block to act as the wrapper for the top bar. We're essentially trying to recreate this element.

```
<div class="py-3 text-sm bg-gray-700"></div>
```

There are three classes that do the following:

- `py-3` - Adds padding to the top and bottom.
- `text-sm` - Decreases the font size.
- `bg-gray-700` - Changes the background color.

We have two options at our disposal. We can apply these classes directly to the block. Alternatively, we can manipulate the block's settings to achieve the same effects. Either solution is valid. It entirely depends on the flexibility you want to give your clients. Applying classes are fast and easy, but they're not going to be easy to modify for clients. Whereas modifying the block settings is slower but allows clients to modify them without knowledge of HTML and CSS.

For this course, we decided to use block settings. We modified the following settings of the **row** block.

- Set the background color to **Udemy Gray 700**.
- Set the font size to **0.875rem**.
- Set the line height to **1.25**.
- Set the top and bottom padding values to **0.75rem**. The left and right padding values are set to **0**.

These are the same values we had in our CSS.

Process

Throughout this course, we're going to use the following procedure for converting a static template into a block.

1. Add a block that best represents the UI element.
2. Add styles that are used in the classes.
3. Add classes to apply styles that can be done through the text editor.

The Group Block

In this lecture, we added the **group** block, which is a generic block for adding content. It can be thought of as the equivalent of the `<div>` tag. This tag can be useful when any other block is not suitable for the job.

We inserted this block into the **row** block with the following classes: `container`, `!mx-auto`, `flex`, `items-center`, and `justify-between`. These classes will center the blocks inserted into the block, which will align them

with the rest of the page. In addition, for testing purposes, we added a paragraph block to test the centering of the content.

Loading Editor Styles

In this lecture, we began loading our theme's CSS into the Gutenberg editor. At the moment, the CSS is only loaded on the front end. For the best possible editing experience, it's considered good practice to load these styles into the editor. This way, clients can accurately view the final result before saving their changes.

The first step to loading our styles in the editor is to hook into the right event. The name of the event recommended for loading styles in the Gutenberg editor is called `after_setup_theme` This hook runs after the theme has been loaded. WordPress recommends this hook for setting up your theme.

```
include( get_theme_file_path('/includes/setup.php') );

add_action('after_setup_theme', 'u_setup_theme');
```

In the above example, we're running a function called `u_setup_theme()`, which will be responsible for handling the setup of our theme. Instead of defining this function in the same file, we're going to define it in a file called **includes/setup.php**. Here's what the code looks like for this file:

```
function u_setup_theme() {
  add_theme_support('editor-styles');
  add_editor_style([
    'https://fonts.googleapis.com/css2?family=Pacifico&family=Rubik:w
    'assets/bootstrap-icons/bootstrap-icons.css',
    'assets/public/index.css'
  ]);
}
```

We're running two functions. The first function is called `add_theme_support()`, which will enable support for custom styles for the Gutenberg editor. We must always enable this feature if we want to load styles. Otherwise, we may run into problems.

WordPress has various features that can be enabled. Therefore, we must pass in the name of the feature we'd like to enable. In this example, the string `editor-styles` can be passed in to enable custom styles.

Afterward, we're calling the `add_editor_style()` function, which can accept a single or array of CSS files to load. External and local paths are supported. If an HTTP URL is passed in, WordPress will load it as-is. Otherwise, WordPress

will assume the file can be found in our theme's folder. You do not need to prepend the path or generate the URL. It'll be generated for you.

Encapsulation

Behind the scenes, WordPress performs encapsulation on your CSS code. This feature only applies to styles added to the Gutenberg editor, not the frontend. Encapsulation is the idea of isolating your styles to the blocks inserted into the template. This is to prevent your styles from affecting the entire Gutenberg editor.

WordPress achieves this by adding the `.editor-styles-wrapper` select to your styles. For example, this:

```
.example {
  color: red;
}
```

Changes to this:

```
.editor-styles-wrapper .example {
  color: red;
}
```

Adding a Navigation Menu

In this lecture, we added a navigation menu to complete the top bar section of the header. WordPress has a block called **Navigation** to add a menu to our theme. We added this block with the following settings:

- Mobile option turned on
- Changed the color to **Udemy Gray 200**
- Changed the block spacing to **1rem**
- The `secondary-menu` class has been added

In addition, the **paragraph** block was modified to remove the margins. The following class was added: `!mt-0`

Adding the Midsection

In this lecture, we began working on the midsection of the header by adding four blocks. The first block we added was the **row** block with the following settings:

- Changed justification to space between

- Disabled the wrap option
- Top and bottom padding set to **1.25rem**
- All margin properties reset to **0**
- Block spacing set to **0**
- The following classes were added: **container !mx-auto relative w-full**

Up next, we added the **Site Title** block with the following changes:

- Link color set to **Udemy Primary**
- Font size set to **1.875rem**
- Appearance set to **bold**
- Line height set to **2.25**

Afterward, the **Search** block was added with the following changes:

- The following classes were added: **header-search-form**

Lastly, an HTML block was added for handling the login and cart links. Unfortunately, it's not possible to recreate every UI element with WordPress's blocks. In these cases, you can resort to the HTML block or create a custom block. In the future, we are going to create a custom block, but for now, we'll stick to HTML blocks.

For this section of the header, we added everything under the `<!-- Header Tools -->` HTML comment.

Fixing the Editor Styles

In this lecture, we fixed the issues with the editor by loading custom styles. Loading the theme's CSS in the template editor is not always enough. This is because the markup generated for the editor can be different from the markup generated on the front end. Therefore, you will need additional stylings to get an exact match for your theme.

To quickly resolve these issues, we updated the `add_editor_style()` function in the `u_setup_theme()` function to the following:

```
add_editor_style([
  'https://fonts.googleapis.com/css2?family=Pacifico&family=Rubik:wgh
  'assets/bootstrap-icons/bootstrap-icons.css',
  'assets/public/index.css',
  'assets/editor.css'
]);
```

The **editor.css** file does not exist in the **assets** directory. You can find a copy of it in the resource section below.

Resources

- Editor CSS

Finishing the Header

In this lecture, we finalized the theme by adding two more blocks to the header template part. First, we added the **Row** block with the following settings:

- Disable the wrap option
- Bottom padding set to **1rem**
- Apply the following classes: **container !mx-auto**

Afterward, we added a **Navigation** block with five dummy links to random pages. The following settings were applied:

- Block spacing option set to **2.5rem**

After making those changes, we transferred the blocks into the **parts/header.html** file. In total, there should be three root blocks with children blocks in each of them.

Tailwind

Hey everyone! Some of you have asked me if I use a CSS framework for creating templates. It just so happens that I do! I use Tailwind for creating static templates before transforming them into a block template. You can check out Tailwind here: https://tailwindcss.com/

It's not necessary to know Tailwind. As long as you have a good grasp of HTML and CSS, the static template should be easy to understand with the developer tools.

Inserting the Columns Block

In this lecture, we inserted the **Columns** block to help us recreate the layout for our theme. It's similar to the **Row** and **Group** blocks that allow blocks to be inserted as the content. We applied the following settings to the **Columns** block:

- Two Columns
- 70/30 Width
- Applied the **container !mx-auto** classes
- Top and bottom margins set to 4rem

Query Loop Block

In this lecture, we added the **Query Loop** block for rendering a list of posts. The **Query Loop** block will take on the responsibility of querying the database for a list of posts. In regards to WordPress, a query is a request for data from the database. This data will then become available to your children blocks.

Along with the **Query Loop** block, a **Post Template** block will be added. This block will be responsible for displaying the template of each post requested by the query.

By default, the query will be performed by the block. However, behind the scenes, WordPress will **always** perform a query on every page request. The URL is scanned by WordPress to determine what posts should be displayed. We can configure the **Query Loop** block to use the query from the page instead of creating another query.

- Enabled "Inherit query from loop" option

Resources

- Query Loop Block

Adding Post Blocks

In this lecture, we added a few blocks to the **Post Template** block to render the data that can be found in the static version. These blocks will be able to grab the data from the query to dynamically render a post. These are the following blocks added to the **Post Template** block:

- Columns (66.66/33.33 width)
  - Column
    - Post Author
    - Post Title
    - Post Date
  - Column
    - Post Featured Image
    - Row
      - Post Tags
      - Post Comment Count
    - Post Excerpt

Styling Post Blocks

In this lecture, we started styling the blocks added to the **Post Template** block. For your convenience, here's an overview of all the changes applied to each block.

Post Author

- Enable **show avatar** option
- Set image size to **24x24**
- Set the text color to **Udemy Gray 700**
- Set the bottom margin to **0.5rem**
- Set padding on all sides to **0**
- Applied the **post-author** class

Post Title

- Enable **make title a link** option
- Set the text color to **Udemy Gray 700**
- Set the font size set to **1.25rem**
- Set the appearance to **Bold**
- Set the line height to **1.75**
- Set the bottom margin to **0.5rem**
- Set the margin top to **0**.

Post Date

- Enable **link to post** option
- Set the link color to **Udemy Gray 500**
- Applied the **!mt-0** class.

Post Featured Image

- Enable **link to post** option
- Set the width to 100%
- Set the bottom margin to **1rem**
- Applied two classes: **rounded-lg overflow-hidden**

Row

- Set the justification to **space between**
- Disable the **wrap** option
- Set the text color to **Udemy Gray 500**
- Set the font size to **0.875rem**
- Set the bottom padding to **1rem**
- Set margins to **0**.

Post Tag

- Set the link color to **Udemy Gray 500**

Post Comment Count

- Applied the following classes: **bi post-comment-count**

Post Excerpt

- Set the font size to **0.875rem**

Columns

- Applied the following classes: **border-b**
- Set the border color to **Udemy Gray 200**
- Bottom margin and padding set to **2.5rem**

Second Column

- Applied the following class: **!mt-0**

Resources

- Boostrap Icons

Pagination

In this lecture, we took a look at the **Pagination** block that was added by the **Query Loop** block. By default, this block will search for the **Query Loop** block as a parent to determine the links. We configured the following settings on this block:

- Set justification to **space between**
- Disable wrap option
- Changed color to **Udemy Gray 500**

In addition, the **Pagination** block acts as a wrapper for the actual links. The actual links are rendered with the **Previous Page**, **Page Numbers**, and **Next Page** links. For this example, we removed the **Page Numbers** block to be consistent with our theme. As for the other two classes, we applied the following classes to add a hover effect: **rounded-md py-2 px-4 block transition-all hover:bg-gray-100**

Sidebar Blocks

In this lecture, we got started with creating the sidebar by creating a template part and inserting blocks into the template part. Inside the **parts** folder, we created a file called **sidebar.html**. The following blocks were added:

Heading

- Set the font size to **1.25rem**
- Set the appearance to **Medium**
- Set the bottom margin to **1.25rem**
- Add Text: Blog Tags
- A duplicate heading was added with the same settings, but the text says: Blog Categories

Tag Clouds

- Set the number of tags to 15
- Applied a class called **sidebar-tags**

Custom HTML

- The contents of this block was the `<div>` tag under the comment that says **Highly Rated Posts**

Categories

- Enable the **Show only top level categories** option
- Applied a class called **sidebar-categories**

Exercise: Footer Blocks

In this lecture, we worked on the footer section of the template as an exercise. To save time, I've provided the completed block, which you can find in the resources section. Check it out for the completed code.

Resources

- Footer Template

Template Hierarchy

In this lecture, we talked about the template hierarchy, which is the mechanism that WordPress uses for loading a template. Before loading the

page, it'll search your theme for an appropriate template to load. For example, here are the files that WordPress will search for in your theme if a visitor is viewing the front page.

1. **front-page.html** – Used for both “your latest posts” or “a static page” as set in the front page displays section of Settings → Reading.
2. **home.html** – If WordPress cannot find **front-page.html** and “your latest posts” is set in the front page displays section, it will look for **home.html**. Additionally, WordPress will look for this file when the posts page is set in the front page displays section.
3. **page.html** – When “front page” is set in the front page displays section.
4. **index.html** – When “your latest posts” is set in the front page displays section but **home.html** does not exist, or when the front page is set, but **page.html** does not exist.

Note: I've replaced the **.php** extension with the **.html** extension. While the documentation shows examples with PHP, the same rules apply to block themes.

By default, all pages will load the **index.html** file if an appropriate template doesn't exist. In most cases, you won't need to define a template for every page. Only define what's necessary. For the rest of this section, we're going to explore the most common templates to create.

Resources

- Template Hierarchy

404 Template

In this lecture, we got started with creating a new template by creating a **404.html** file in the **templates** directory. The template for this page is simple. To save time, I've provided the template for you in the resources section.

According to the documentation, WordPress will search for the following templates in this order.

1. 404.html
2. index.html

Resources

- 404 Template

Category Template

In this lecture, we created a template for categories. WordPress will check for several templates for displaying posts for a category. Here's the official list:

1. **category-{slug}.html** – If the category's slug is **news**, WordPress will look for **category-news.html**.
2. **category-{id}.html** – If the category's ID is **6**, WordPress will look for **category-6.html**.
3. **category.html**
4. **archive.html**
5. **index.html**

In some cases, you may see a portion of the URL wrapped with curly brackets. The brackets indicate a placeholder. For example, the **category-{slug}.html** has a placeholder for the slug of a category. This placeholder must be replaced.

We decided to go with the **category.html** template to act as a generic template for all categories. The category template is nearly identical to the index template with the addition of the page header. We added the following block above the columns.

```
<!-- wp:html -->
<!-- Page Header -->
<div class="bg-gray-100 py-8 mt-0">
  <div class="container mx-auto">
    <h1 class="text-3xl font-medium">Category: Cats</h1>
  </div>
</div>
<!-- /wp:html -->
```

Search Template

In this lecture, we added a search template to our theme for displaying search results. Here are the templates WordPress will search for in our theme.

1. **search.html**
2. **index.html**

We went with the **search.html** option. The search template is identical to the index template with the addition of a search form. We added this form with an HTML block. Here's the following block:

```
<!-- wp:html -->
<!-- Search Block -->
<div class="bg-red-400 rounded-lg p-8 mb-16 shadow-lg">
  <h1 class="font-bold text-white text-2xl mb-4">
    Search: Your search term here
  </h1>
```

```
  <form
    class="sm:flex items-center bg-white rounded-lg overflow-hidden p
  >
    <input
      class="text-base text-gray-400 flex-grow outline-none px-2"
      type="text"
      placeholder="Search"
    />
    <div class="ms:flex items-center px-2 rounded-lg space-x-4 mx-aut
      <button class="bg-red-400 text-white text-base rounded-lg px-4
        Search
      </button>
    </div>
  </form>
</div>
<!-- /wp:html -->
```

Single Post Template

In this lecture, we created a templated for single posts. Unlike the other templates, this template is vastly different. As usual, here is the list of templates WordPress will search for from within your theme:

1. **single-{post-type}-{slug}.html** – (Since 4.4) First, WordPress looks for a template for the specific post. For example, if the post type is a product and the post slug is **dmc-12**, WordPress would look for **single-product-dmc-12.html**.
2. **single-{post-type}.html** – If the post type is product, WordPress would look for **single-product.html**.
3. **single.html** – WordPress then falls back to **single.html**.
4. **singular.html** – Then it falls back to **singular.html**.
5. **index.html** – Finally, as mentioned above, WordPress ultimately falls back to **index.html**.

For our theme, we went with the **single.html** template to act as a generic template for all posts. As a base, we used the index template. We've made several modifications to the template.

Row

- This block will contain the **Post Author**, **Post Date**, and **Post Comment Count** blocks
- Set the block spacing to **1rem**

Post Author

- Removed Margins

Post Comment Count

- Add the following classes: !ml-auto

Post Title

- Set the font size to 2.25rem

Post Tags

- Added a class called post-content-tags

Row for Pagination

- Set top and bottom margin to 2rem
- Set block spacing to 0
- Set border styles to solid
- Set border width to 1px
- Set border radius to 0.5rem
- Set border color to Udemy Gray 200

Previous Post

- Applied the following classes: basis-0 grow max-w-full py-5 px-3 transition-all hover:bg-gray-100 text-center border-r border-r-gray-200

Next Post

- Applied the following classes: basis-0 grow max-w-full py-5 px-3 transition-all hover:bg-gray-100 text-center !mt-0

Heading

- Set the font size to 1.5rem
- Set the appearance to Medium

Post Comments

- Add a class called comments-section

Query Loop Block

This block was removed from our template. It's not necessary to keep this block around since most post blocks will grab data from the query created by WordPress. Without the Query Loop block, WordPress will not loop through

the results if there are multiple. Luckily, single posts are always guaranteed to have one post.

Page Template

In this lecture, we added the page template, which is the minimum version of the **single** template. WordPress will search for the following template in our theme:

1. **custom template file** – The page template assigned to the page. See **get_page_templates()**.
2. **page-{slug}.html** – If the page slug is recent-news, WordPress will look to use **page-recent-news.html**.
3. **page-{id}.html** – If the page ID is 6, WordPress will look to use **page-6.html**.
4. **page.html**
5. **singular.html**
6. **index.html**

For our theme, we went with the **page.html** template file that'll act as a generic template for all pages. The page template is nearly identical to the index template with the following blocks removed:

- Removed both **Row** blocks
- Removed the **Post Tags** block

Custom Templates

In this lecture, we created a custom template for a full-width page. WordPress allows theme developers to offer additional templates for various designs. Firstly, the custom template must be registered through the **theme.json** file. We added the following array to our theme file.

```
{
  "customTemplates": [
    {
      "name": "full-width-page",
      "title": "Full Width Page",
      "postTypes": ["page"]
    }
  ]
}
```

The `customTemplates` property is an array of objects where each object represents a custom template. Each object can have the following properties.

- `name` - The filename without the file extension

- `title` - A human-readable name that will be displayed to the user
- `postType` - An array of post types that the template can be used on

We made a replica of the single template into a file called **full-width-page.html**. We modified the template by updating the number of columns in the **Columns** block to 1 column.

Templates can be applied by editing a specific page. On the sidebar, there will be a panel called **Templates** with a dropdown of custom templates in the theme.

JavaScript and React Fundamentals

In this section, we learned about the JavaScript programming language and React library for building interactive interfaces on the web.

What's to come

In this lecture, we talked about what you can expect from the rest of this course. There are no lecture notes for this lecture.

Introduction to JavaScript

In this lecture, we got an idea of what JavaScript is and why it was introduced. There are no lecture notes for this lecture.

First Taste of JavaScript

In this lecture, we got our first taste of JavaScript. For starters, we explored two areas for running JavaScript. Under the developer tools, the **Console** panel is available for quickly running code in the browser without creating a file.

Another area to run JavaScript is to use the **Snippets** section under the **Sources** Panel. Unlike the console, snippets are files that are temporarily stored in the browser, not our project, that can be executed. This allows us to write more complex code.

We created a snippet that runs the following code:

```
alert("Hello world!");
```

This will make a popup appear with the text `Hello World`.

Functions

JavaScript supports functions. The idea of a function is to allow developers to write reusable blocks of code. In JavaScript, there are three types of functions

- Functions defined by the language
- Custom defined functions

- Functions defined by the environment

As opposed to PHP, JavaScript can run on the desktop or mobile device. It's not restricted to the browser. Each of these locations is considered an environment. Most environments will define additional functions that will be available from our script. In the above example, we're using a function called `alert()`, which is available in most browsers.

Strings

JavaScript supports strings, which can be written with single or double-quotes. Either style is suitable. The program doesn't care. Strings allow us to store random text such as names, addresses, or content.

Semicolons

Unlike PHP, semicolons are completely optional in JavaScript. JavaScript is more than capable of ending a line of code for you. If we were to take the previous code example without ending it in a semicolon, it would look like this:

```
alert("Hello world!");
```

Data Types

In this lecture, we talked about data types that are supported in JavaScript. They're the following:

- String
- Number
- Boolean
- Null
- Undefined
- Objects

There are more data types, but these are the most common data types and the ones we'll be working with throughout the course. Like PHP, JavaScript is a dynamically typed language. Therefore, we don't have to explicitly assign data types to our variables. This process is handled for us behind the scenes.

Variables

In this lecture, we explored a core feature of JavaScript called variables. Variables allow us to store data in memory. We can define a variable by using

the `let` keyword. This is followed by the name of the variable. Here are rules for variable names

- Must start with a letter, underscore(`_`), or dollar sign (`$`) character.
- Can only contain letters, numbers, underscores, or dollar signs
- Spaces are not allowed
- Reserved keywords are not allowed
- Case-sensitive

Here's an example of a variable definition and its usage:

```
let myName = "John";
alert(myName);
```

Constants

In this lecture, we explored another way of creating variables called constants. A constant is a variable that cannot have its value changed. Before we check out how a constant is created, here's how you would update a regular variable.

```
let myName = "John";
myName = "Jane";
```

Retyping the `let` keyword is not necessary. It's only necessary for initializing variables.

Here's an example of a constant definition.

```
const myName = "John";
```

If you were to attempt to update the `myName` variable, the browser would throw an error at you. Constants can be reliable for storing data that should never change.

One more thing

There's another keyword available for creating variables called the `var` keyword. Before the `const` and `let` keywords, the `var` keyword was the only way to create variables. It's considered outdated compared to the `const` and `let` keywords, and most developers prefer them over the `var` keyword.

Template Literals

In this lecture, we learned about a third option for creating strings called template literals, AKA string templates. Unlike regular strings, template literals allow us to inject values into the string. Here's an example of a string literal

```
const age = 28;
alert(`My age is ${age}`);
```

A template literal can be written with a backtick character. Inside the string, we can inject values by adding a placeholder with the following syntax: `${}`. Inside the curly brackets, you can enter a valid expression.

Expressions are lines of code that evaluate to a value. Refer to the video in the resource section for a further explanation

String Concatenation

Another syntax is available for adding values to strings called string concatenation. Same goal, different syntax. Ultimately, most developers prefer template literals over string concatenation. Just in case, here's what string concatenation looks like:

```
const age = 28;
alert("My age is " + age);
```

The `+` operator is not limited to adding numbers. It can be used for adding strings together too.

Resources

- Expressions

Creating Functions

In this lecture, we learned how to create functions in JavaScript. Functions support argument lists and return values. Here's an example of a function we wrote

```
function sum(a, b) {
  return a + b;
}

sum(10, 20);
```

Functions are defined with the `function` keyword followed by the name of the function, argument list, and body. Multiple arguments can be added by separating them with a comma.

Lastly, we can call a function by writing its name followed by an argument list.

Arrays

In this lecture, we took a quick look at how to write arrays. Arrays are a feature for storing a collection/group of data within a single variable. Arrays can contain any type of value. We're not restricted to the type of data that can be stored in an array.

Arrays are written with `[]` syntax. Here's an example of an array:

```
const states = ["NJ", "NY", "CA"];

alert(states[1]);
```

Arrays are zero-based indexed. This means the first item in the array can be accessed with the index of `0`. The second item can be accessed with an index of `1`. So on and so forth.

Objects

In this lecture, we got into a discussion on objects. Objects are another data type for storing a collection/group of data. JSON is heavily inspired by object syntax in JavaScript. However, JavaScript objects offer more features than a JSON object. Let's explore what those are.

Here's an example of an object

```
const checkingAccount = {
  name: "John",
  balance: 1000,
};

alert(checkingAccount.balance);
```

Objects are created with `{}` syntax. The syntax is very similar to JSON, with a few exceptions. We don't need to surround property names with quotes. They're completely optional. We can access an object's properties with dot syntax.

An interesting feature of objects is being able to define functions like so:

```
const checkingAccount = {
  name: "John",
  balance: 1000,
  withdraw: function (amount) {
    checkingAccount.balance = checkingAccount.balance - amount;
  },
};

checkingAccount.withdraw(200);
alert(checkingAccount.balance);
```

There's a shorthand way of writing functions by omitting the `function` keyword like so:

```
const checkingAccount = {
  name: "John",
  balance: 1000,
  withdraw(amount) {
    checkingAccount.balance = checkingAccount.balance - amount;
  },
};
```

Another useful feature for writing less code is the `this` keyword. The `this` keyword will always point to the object it's used in. This way, you don't have to type the full name of the object. However, it can only be used in objects. You can't use it outside of an object. Here's an example:

```
const checkingAccount = {
  name: "John",
  balance: 1000,
  withdraw(amount) {
    this.balance = this.balance - amount;
  },
};
```

Another term used for functions defined inside an object is called a method. The words function and method can be interchangeable, but there is a difference.

The Console Object

In this lecture, we got started with the console object for debugging an application. Throughout this section, we've been relying on the `alert()` function to view data from our scripts. However, this function was not intended for this purpose. It blocks the visitor from interacting with the page until the alert box is closed.

A better solution is to use the `console` object, which provides methods for adding messages to the console. The most common method for logging messages is the `console.log()` method. Here's an example:

```
console.log("Hello World!");
```

Resources

- Console

Loading JavaScript

In this lecture, we loaded JavaScript by adding the `<script></script>` tags to the document. These tags can be added anywhere. Inside these tags, we can write JavaScript like so:

```
<script>
  console.log("Hello world!");
</script>
```

Alternatively, we can link to an external script by adding the `src` attribute:

```
<script src="main.js"></script>
```

Both options are viable, but it's recommended to keep your JavaScript code separate from your HTML code for maintainability. Regardless, you should always load a script at the bottom of the document. Problems can arise by loading your JavaScript first. If you were to load a script early, your elements may not be ready by the time you need to access them. Therefore, you should always load your scripts last unless you really need to load them early.

Working with the DOM

In this lecture, we talked about the document object model (DOM). Behind the scenes, the browser will convert your document's elements into a tree-like structure. This tree contains a series of objects that map your entire document. Through these objects, you can interact with the elements by reading/writing to them, like change their content, styles, or add/remove tags.

The entire tree can be found under the `document` object, which represents the document. If you would like to select a specific element, you can use the `querySelector()` function.

```
document.querySelector("h1");
```

This function accepts a valid CSS query, which means you can select elements by classes, IDs, or other attributes. You can even use nested selectors like so:

```
document.querySelector("ul li");
```

In some cases, you may want to select multiple elements. You can use the `document.querySelectorAll()` function to do so.

```
document.querySelectorAll("ul li");
```

This function will return an array of elements. Overall, these two methods should cover all your use cases. There are alternative methods like the `getElementById()`, `getElementsByTagName()`, and `getElementsByClassName()` functions. As their names suggest, you can grab elements by their ID, tag name, or class name.

Reading elements is not your only option. You can change an element's properties too. Here's an example of changing an element's `color` property.

```
const li = document.querySelectorAll("ul li");

li[1].style.color = "red";
```

Check out the resource section for a complete list of properties and methods available for objects selected by the DOM API.

Resources

- DOM API

Conditional Statements

In this lecture, we looked at how to create conditional statements. In JavaScript, the idea of conditional statements is similar to conditional statements in PHP. They allow us to control the flow of logic by executing code if a condition is true. Conditional statements can be written with the `if`, `else if`, and `else` keywords.

Here's an example:

```
const li = document.querySelectorAll("ul li");

if (li[1].innerHTML === "Jane") {
  console.log("Jane found!");
} else if (li[1].innerHTML === "John") {
  console.log("John found!");
} else {
  console.log("No one was found!");
}
```

A couple of things worth mentioning. Firstly, the `innerHTML` property can be found on all DOM objects. It contains the inner contents of an element, including children elements.

Secondly, we're using the **strict equal** operator to compare two values but also comparing their data types. Data types can be tricky in JavaScript. In some cases, two values may get matched even if they're not necessarily a complete match. To avoid errors like these, consider using the **strict equal** operator over the **equal** operator. For a list of complete comparison operators, check out the resource section of this lecture.

Thirdly, we're chaining the conditional statements in a specific order. The `if` statement must come first, followed by a series of `else if` statements. Lastly, the `else` statement must be last. You can have as many conditions as you'd like. It's optional to add an `else if` statement if you only plan on checking for a single condition.

Resources

- Comparison Operators

Understanding Scope

In this lecture, we talked about scope, which is the idea of how blocks of code can affect the accessibility of a variable. If a variable is defined inside a block of code, it won't be accessible to outside code. Think of a block of code as a fence that blocks outside intruders from accessing the code inside.

Check out the video for a complete breakdown of scope with examples.

Arrow Functions

In this lecture, we learned about another option for writing functions called arrow functions. Arrow functions are written without the `function` keyword, name, and include a fat arrow(`=>`). Here's an example.

```
const hello = () => console.log("hello world");
```

Arrow functions are anonymous. So, you'll have to assign it to a variable or pass it into another function. Unlike regular functions, adding `{}` is optional. Arrow functions can be written on the same line. An arrow function with multiple lines would require the `{}` characters like so:

```
const hello = () => {
  console.log("hello world");
};
```

Parameters

Arrow functions can have parameters just like regular functions. Multiple parameters can be added by separating them with a comma. In some cases, you can add/remove the parentheses based on the number of parameters in your functions.

- If your function has 0 parameters, parentheses are required.
- If your function has 1 parameter, parentheses are optional.
- If your function has 2 or more parameters, parentheses are required.

Scope

Arrow functions do not have a scope. Instead, they inherit their parent's scope. Take the following example:

```
const foo = {
  num: 10,
  logNum() {
    console.log(this.num);
  },
};
```

This is an object with a regular function called `logNum`. It'll log the `num` property without a problem. If we used an arrow function, things would be different.

```
const foo = {
  num: 10,
  logNum: () => {
    console.log(this.num);
  },
};
```

This wouldn't work because the `this` keyword no longer references the object. Therefore, we would receive `undefined` from JavaScript. If we would like to grab the `num` property, we would need to reference it through the `foo` property.

```
const foo = {
  num: 10,
  logNum: () => {
    console.log(foo.num);
  },
};
```

Overall, arrow functions are considered popular because they're easier to read and they can borrow from the parent scope. It might not seem like it, but these features are incredibly helpful in larger applications. Once we get into Gutenberg, it'll become clearer as to why this feature is popular.

Destructuring

In this lecture, we learned about destructuring for quickly making our code readable by extracting properties from an object. Oftentimes, we may need to access a property deeply nested inside an object. It would be annoying to constantly type out the full path to the object. One solution would be to assign a property to a variable like so:

```
const foo = {
  num: 10,
  logNum: () => {
    console.log(foo.num);
  },
};

const num = foo.num;
```

However, if you would like to extract more properties, you would need to create a variable for each one. This can become tedious. Luckily, we can use destructuring to shorten the process. The previous example can be modified to the following:

```
const { num, logNum } = foo;
```

The variable name is wrapped with `{}` where the name of the properties are listed inside. The value for the variable is where the properties can be found. Additional properties can be destructured by separating them with commands.

Destructuring from arrays is also possible. Take the following:

```
const names = ["John", "Jane", "James"];
const [firstName] = names;
```

The first name gets mapped to the first item in the array. If we had a second name, the second item from the array would get mapped to it. So on and so forth.

What is React?

In this lecture, we talked about React, which is a JavaScript library for helping us build rich interactive user interfaces. Similar to WordPress, React takes care of the more complex technical details for building a user interface. Technically, you can build an application without React, but you'll be writing a lot more code than usual.

Overall, React is a great library to learn. On top of that, React is supported by WordPress. One of the benefits of using React is that there are tools for optimizing your code. React will help make sure your code is ready for production.

For this course, we're going to learn about React along with the tools necessary for running React apps.

Resources

- React

Getting Started with React

In this lecture, we got started with React by launching a starter project with Stack Blitz. Stack Blitz is a site that provides development environments for testing features in the browser instead of committing to your own environment. Inside the starter project, you'll find the following:

- **public** - A folder containing the HTML of your project.
- **src** - A folder that contains the code for your application code.
- **package.json** - A file containing the settings of your project.

We dived deeper into the **package.json** file. This file will contain various settings for configuring your project. Most importantly, you'll find a list of dependencies. You can download packages created by other developers to expose features to your application.

There are objects for listing dependencies. They're `dependencies` and `devDependencies`. The `dependencies` object should contain packages that should be shipped with your project for production. Whereas the `devDependencies` should contain a list of packages that should be for development.

There's another property called `scripts` that we can safely ignore and will be revisiting in a future section.

Resources

- React Stackblitz
- Package File Docs
- NPM

Creating a React App

In this lecture, we created a minimal React application. To get started, we imported React with the following code:

```
import React from "react";
```

We're using the `import` statement to include code from external files. This is followed by a name to reference the functions and variables exported by a file. Lastly, we added the `from` keyword to specify the location. In this example, we're specifying the package name. Node will be intelligent enough to understand this package is installed with your machine. It doesn't need a full path.

Afterward, we created an element with the `React.createElement()` function.

```
const h1 = React.createElement("h1", null, "Hello world!");
```

This function will create an element in memory. It has three arguments.

1. The name of the element
2. Properties/attributes to add to the element
3. Inner content

This won't insert the element into the document. That would be next step. First, we must import the `ReactDOM` object like so:

```
import ReactDOM from "react-dom";
```

React supports various platforms like desktop and mobile apps. The core `React` package handles creating and updating elements. Whereas the `ReactDOM` package will handle inserting and updating elements in the DOM. We can run the `render()` function to add the element to the DOM like so:

```
const app = document.querySelector("#app");

ReactDOM.render(h1, app);
```

The `render()` function has two arguments.

1. The element to insert into the document.
2. A location for rendering an element.

Rendering Multiple Elements

In some cases, you may want to render multiple children elements. You can pass in an array to the third argument of the `createElement()` function instead of a string to insert children elements like so:

```
const div = React.createElement("div", null, [
  React.createElement("h1", null, "Hello world!"),
  React.createElement("p", null, "This is a paragraph"),
  React.createElement("p", null, "This is another paragraph"),
]);

ReactDOM.render(div, app);
```

Dynamic Content

In this lecture, we looked at how to render dynamic content with React. This is where React shines. First things first, we converted our element to be returned by a function rather than storing it in a variable like so:

```
function Page() {
  return React.createElement("div", null, [
    React.createElement("h1", null, `Hello ${new Date().toLocaleStrin
    React.createElement("p", null, "This is a paragraph"),
    React.createElement("p", null, "This is another paragraph"),
  ]);
}
```

Two things worth noting. Firstly, we're choosing to use a function to prevent memory leaks from happening. A memory leak can happen whenever a variable occupies memory when it's no longer needed anymore. Memory leaks can cause a visitor's machine to become sluggish.

Secondly, we're using a function called `Date()`, which is defined by the JavaScript language to return an object that has information on the current. From this object, we're calling the `toLocaleString()` to return a human-readable time. This time is inserted into the `h1` element.

Next, we wrapped the `render()` function with the `setInterval()` function.

```
setInterval(() => {
  ReactDOM.render(Page(), app);
}, 1000);
```

We're using the `setInterval()` function to run a function every second. The arguments are the following:

1. The function to execute.
2. The interval, which is measured in milliseconds.

In this example, we're running the `Page()` function every second, which is responsible for displaying a timer. Even though the `Page()` function returns the entire document, only the `h1` element gets updated. React is smart enough to only update what's necessary. Thus, saving us from having to manage what needs to be updated.

Introduction to JSX

In this lecture, we got introduced to JSX. It's an extension for JavaScript files that allows us to write HTML-like syntax in an HTML file. Since we're using React with Webpack, we can immediately start adding JSX into our codebase. For example, we can replace the value returned by the `Page()` function with the following:

```
function Page() {
  return (
    <>
      <h1 className="orange">Hello world!</h1>
      <p>This is a paragraph</p>
      <p>This is another paragraph</p>
    </>
  );
}
```

To be absolutely clear, JSX is not HTML. There are differences.

1. There may only be one root element.
2. If you don't want to clutter your document with a root element, you can use the fragment element, which will create an invisible wrapper without introducing another element.

3. The `class` attribute is not supported. If you would like to add classes to your elements, you can apply the `className` attribute.

For demonstration purposes, we created a **style.css** file with the following contents:

```
.orange {
  color: orange;
}
```

At the top of the **index.js** file, we imported the CSS with the following code:

```
import "./style.css";
```

Notice how we're importing the CSS file. Normally, JavaScript does not support CSS file imports. Thanks to Webpack, we get the benefit of being able to import CSS.

Components

In this lecture, we explored the idea of components. Components are a feature for teaching the browser new tags. A component is created by defining a function that returns JSX. For example, our `Page()` function is a component. We can use our custom elements by using the syntax for creating tags with the name of the function inside.

For example, we updated the `ReactDOM.render()` function by swapping the `Page` function with its tag counterpart like so:

```
ReactDOM.render(<Page />, app);
```

As extra practice, we outsourced the heading to a component.

```
function Header() {
  const clock = Date().toLocaleString();

  return <h1 className="orange">Hello World! {clock}</h1>;
}
```

In this example, we are creating another component called `Header`. Inside this component, we are injecting a variable into our HTML by using `{}`. Inside these brackets, we can add a valid JavaScript expression. The value evaluated from the expression will appear in the contents of the template.

Lastly, we updated the `Page` component by swapping the `<h1>` tag with our `<Header>` component:

```
function Page() {
  return (
    <>
      <Header />
      <p>This is a paragraph</p>
      <p>This is another paragraph</p>
    </>
  );
}
```

Extending Components with Props

In this lecture, we extended the behavior of a component by adding custom attributes, which are known as props in react. Attributes allow us to manipulate the behavior of HTML elements like the `width` and `height` attributes for `<img>` tag.

In the `Page` component, we updated our use of the `Header` component by passing on a name:

```
function Page() {
  return (
    <>
      <Header name="John" />
      <p>This is a paragraph</p>
      <p>This is another paragraph</p>
    </>
  );
}
```

Custom prop names can be whatever you want. Props can be accepted by adding the `props` parameter to your component's function like so:

```
function Header(props) {
  const clock = Date().toLocaleString();

  return (
    <h1 className="orange">
      Hello {props.name}! {clock}
    </h1>
  );
}
```

In some cases, you may want to pass on dynamic data. You can do so by setting the value of a prop to a pair of curly brackets like so:

```
function Page() {
  const name = "John";

  return (
    <>
      <Header name={name} />
      <p>This is a paragraph</p>
      <p>This is another paragraph</p>
    </>
  );
}
```

ES6 Modules

In this lecture, we talked about how to import/export data. For organization purposes, you should split your code into separate files. Luckily, JavaScript introduces modules for sharing code between files. In an external file called **Page.js**, we exported the `Page` component like so:

```
import React from "react";

const age = 28;

export default function () {
  return (
    <>
      <Header />
      <p>This is a paragraph</p>
      <p>This is another paragraph</p>
    </>
  );
}
```

A couple of things worth mentioning. Firstly, we importing `React` since Webpack will convert JSX elements into the `React.createElement()` function. Without this function, our code wouldn't work. So, we need to import it even though we're not writing it out in our function.

Secondly, we're creating a variable called `age`. This variable will never be exported. Data must be explicitly exported with the `export` keyword like the function. Our function has been exported under the default namespace. A namespace is a programming concept where data is exported under a specific location.

The default namespace is a location that allows you to export data without assigning a name. This means you can export an anonymous function. After exporting the function, we can import it like so:

```
import Page from "./Page";
```

Local files can be imported by providing a local path. We're not finished yet. The `Home` component should be exported from a separate file.

```
import React from "react";

export function Header(props) {
  const clock = Date().toLocaleString();

  return (
    <h1 className="orange">
      Hello {props.name}. {clock}
    </h1>
  );
}
```

In this example, we're creating a named export. We can import this file by using the following syntax:

```
import { Header } from "./Header";
```

By using a named export, we must import the data by its name. By splitting our code into separate files, our project will be more maintainable and easier to read.

Adding State to a Component

In this lecture, we added state to our component by utilizing the `useState()` function. Our application is relying on the `setInterval()` function wrapped around the `render()` function to update the timer. However, this is not good practice as all elements have to be checked. Its considered good practice to only update a specific section without bothering the entire application.

Rather than updating the heading from the **index.js** file, we should update it from the `Header` component. In the **index.js** file, remove the `setInterval()` function:

```
const app = document.querySelector("#root");

ReactDOM.render(<Page />, app);
```

Next, we updated the `Header` component to the following:

```
import React from "react";

export function Header(props) {
```

```
  let [clock, setClock] = React.useState(Date().toLocaleString());

  setInterval(() => {
    setClock(Date().toLocaleString());
  }, 1000);

  console.log("Component updated");

  return (
    <h1 className="orange">
      Hello {props.name}. {clock}
    </h1>
  );
}
```

We're using the `useState()` function to add state to our component. State refers to the data of an application. Its data that React should care about and update the component if the state changes. This function has one argument, which is the default value.

It'll return an array where the first item is the current value and the second value is a function to update the state. It's considered good practice to destructure the items for readability.

From within our component, we're updating the state a combination of the `setInterval()` and `setClock()` functions. React will automatically rerun the function to render the latest template to the page.

Events

In this lecture, we talked about handling events in JavaScript. By default, JavaScript has support for handling events, which can be triggered by clicks, keyboard typing, or scrolling on the page. React extends this behavior by adding syntax for easily listening for events.

We created a `Counter` component for this example. This component was added to the `Page` component.

```
import React from "react";
import { Header } from "./Header";
import Counter from "./Counter";

export default function () {
  const name = 28;

  return (
    <>
      <Header name={name} />
      <p>This is a paragraph</p>
      <p>This is another paragraph</p>
      <Counter />
```

```
    </>
  );
}
```

Our `Counter` component looks like the following:

```
import React from "react";

export default function Counter() {
  let [count, setCount] = React.useState(0);

  function handleClick(event) {
    event.preventDefault();

    setCount((prevCount) => prevCount + 1);
  }

  return (
    <a href="#" onClick={handleClick}>
      Count: {count}
    </a>
  );
}
```

An event can be listened to by adding an attribute to the element that should have the event. You can refer to the resource section for a complete list of events. In this example, we're using the `onClick` event.

Next, we're setting the event to a function called `handleClick`. Take note that we're not adding the `()` characters after the function name. We want to pass on a reference to the event, not call the function. React will be able to call our function on our behalf.

The `handleClick()` function is performing two actions. Firstly, it's accepting the `event` object, which comes with every event. It'll contain information on the current event. We're using it to prevent the default behavior, which is to update the URL in the address bar.

Secondly, we're updating the current click count by using the `setCount()` function. We have the option of passing in a value, but we can also pass in a function that'll be provided the previous value. The return value of this function will be the new value of the `count` variable.

Resources

- JavaScript Events
- React Events
- Event Object

Using useEffect

In this lecture, we persisted the click count by storing the count in the Local Storage. Local Storage is an API provided by the browser. It allows you to store data on the browser that'll persist across page refreshes.

In addition to the Local Storage, we're using a function called `useEffect()`, which will allow us to run a function whenever data changes from within our component. It has two arguments, a function to call when data changes and a list of dependencies.

```
React.useEffect(() => {
  localStorage.setItem("count", count);
}, [count]);
```

In the above example, we're watching the `count` variable for updates. If it's updated, we're storing an item in the Local Storage API with the `localStorage.setItem()` function. The `localStorage` exposes to the browser's local storage with various methods. One of the methods is called `setItem()`, which has two arguments. The first argument is the name of the item, and the second argument is the value.

Afterward, we created another `useEffect()` function. This time, we're not supplying a list of dependencies. This tells React to execute the function immediately after the component has been rendered:

```
React.useEffect(() => {
  if (localStorage.getItem("count")) {
    setCount(parseInt(localStorage.getItem("count")));
  }
}, []);
```

In the above example, we're grabbing the `count` item from local storage by using the `localStorage.getItem()` method. If the item exists, we'll update the state with the `setCount()` method. Before doing so, we're wrapping the value with the `parseInt()` function to convert the data type to an integer. Otherwise, we may receive unexpected behavior if we're working with the wrong data type.

Resources

- Local Storage

The Command Line

In this lecture, we explored the command line. Before interfaces existed, everything we wanted to do was done through the command line. Such as sending emails, downloading files, or playing audio. Developers prefer to use the command line since a user interface can bog down the performance. Most tools are only executable through the command line.

You can open the command line by searching for a program called **Powershell** on a Windows machine. If you're on a Mac/Linx, you can search for a program called **Terminal**.

There are dozens of commands available. Luckily, it's not required to be a master of the command line. You can get away with the following commands:

- `pwd` - Short for **Present Working Directory**. This command will output the full path you're currently in.
- `ls` - Short for **List**. This command will output a full list of files and folders that are in the current directory.
- `cd` - Short for **Change Directory**. This command will change the current directory. You can use two dots (`..`) to back up a directory instead of moving into a directory.

In Visual Studio Code, you can open the command line by going to **Terminal > New Terminal**. By default, the command line will point to your project directory, which can make things easier. This saves you time from moving the command line to your project.

Getting Started with Node

In this lecture, we got started with Node. Node is a program for executing JavaScript on your machine without using a browser. We must install Node as the tools for optimizing your code are built with JavaScript and Node.

You should download the latest version of Node. You may find older versions, which are available for developers who developed their apps on older versions and need support. Installing Node is like installing any other program.

After installing Node, a program will be available for running JavaScript, but it's not necessary to use this tool. It was provided for quickly testing JavaScript on your machine. In most cases, you will want to execute code from your JS files.

Let's say we had a file called **index.js** with the following code:

```
console.log("test");
```

We can execute this file by running the following command: `node index`

By installing Node, a new command will become available called `node`, which will tell Node to run a JavaScript file. After this command, we can provide the name of the file without the extension. Node will assume the file is already a JS file.

The most important step to remember is that the file and command must be in the same directory. If your command line is in a different directory than your file, the command will not work. Luckily, if you're using VSC, the editor should point to your project's directory already.

Node can do so many things, from starting a server to interacting with a file system. In our case, we're going to be installing tools for optimizing our JavaScript codebase.

Resources

- Node.js

Block Development Fundamentals

In this section, we created our first block for the Gutenberg editor. Along the way, we created a plugin with tooling for proper development.

Creating a Plugin

In this lecture, we got started with creating a plugin. A can be registered by creating a folder called **udemy-plus** inside the **wp-content/plugins** directory. This directory is dedicated to all the plugins installed on your WordPress site. The minimum requirement for creating a plugin is a PHP file with a header.

The PHP file can be called whatever you want. In most cases, developers will either use **index.php** or name the plugin file after the name of the plugin. For our case, we'll use **index.php**. Inside this file, we added the following code:

```
<?php
/**
 * Plugin Name: Udemy Plus
 * Plugin URI: https://udemy.com
 * Description: A plugin for adding blocks to a theme.
 * Version: 1.0.0
 * Requires at least: 5.9
 * Requires PHP: 7.2
 * Author: Udemy
 * Author URI: https://udemy.com
 * Text Domain: udemy-plus
 * Domain Path: /languages
 */
```

The file header will be scanned and extracted from the main plugin file. This information is displayed to the user on the plugin page. We can add the following headers to our file:

- **Plugin Name**: The name of your plugin, which will be displayed in the Plugins list in the WordPress Admin.
- **Plugin URI**: The home page of the plugin, which should be a unique URL, preferably on your own website. This must be unique to your plugin. You cannot use a WordPress.org URL here.
- **Description**: A short description of the plugin, as displayed in the Plugins section in the WordPress Admin. Keep this description to fewer than 140 characters.
- **Version**: The current version number of the plugin, such as 1.0 or 1.0.3.

- **Requires at least**: The lowest WordPress version that the plugin will work on.
- **Requires PHP**: The minimum required PHP version.
- **Author**: The name of the plugin author. Multiple authors may be listed using commas.
- **Author URI**: The author's website or profile on another website, such as WordPress.org.
- **License**: The short name (slug) of the plugin's license (e.g., GPLv2). More information about licensing can be found in the WordPress.org guidelines.
- **License URI**: A link to the full text of the license (e.g., https://www.gnu.org/licenses/gpl-2.0.html).
- **Text Domain**: The gettext text domain of the plugin. More information can be found in the Text Domain section of the How to Internationalize your Plugin page.
- **Domain Path**: The domain path lets WordPress know where to find the translations. More information can be found in the Domain Path section of the How to Internationalize your Plugin page.
- **Network**: Whether the plugin can only be activated network-wide. It can only be set to true and should be left out when not needed.
- **Update URI**: Allows third-party plugins to avoid accidentally being overwritten with an update of a plugin of a similar name from the WordPress.org Plugin Directory.

Resources

- Header Requirements

Securing the Main Plugin File

In this lecture, we took the time to secure the main plugin file. Users are able to visit files directly without the help of WordPress. However, visiting files directly can cause issues and expose information if not properly accounted for. To combat this issue, you should check if a file is being visited directly.

The most common method of checking if a file is being visited directly is by using a function called `function_exists()`, which is defined by PHP. It'll check if a function has been defined beforehand. Here's how we used it in the **index.php** file.

```
if (!function_exists('add_action')) {
  echo "Seems like you stumbled here by accident. 😜";
  exit;
}
```

Firstly, we're checking if the `add_action` function was defined. This function is defined by WordPress. Since WordPress loads first before our plugins, we can safely assume that the file is not being visited directly if this function is defined. If it isn't available, we exit the script with the `exit` statement.

Inside our condition, we're using the `!` (not) operator. This operator will check if a condition is `false` rather than `true`.

Installing WP Scripts with NPM

In this lecture, we installed the **WordPress Scripts** package in our plugin. By doing so, we'll be provided tools for helping us build blocks. Before doing so, we initialized NPM with our project.

```
npm init -y
```

The `npm` command is available after installing NodeJS. This command allows us to interact with NPM with various subcommands. One of the commands is called `init`, which will create a **package.json** file. This file is required for installing dependencies as it'll keep track of dependencies for your project. We're adding the `-y` flag to skip the question process. NPM will fill your **package.json** file with default values.

Next, we installed the **@wordpress/scripts** package with the following command:

```
npm install @wordpress/scripts --save-dev
```

We're using another subcommand called `install`, which will install a package from the NPM registry. After this command, we must specify the name of the package that we'd like to install. In some cases, the name of a package is the author name followed by the name of the package.

Lastly, the `--save-dev` flag will install the package as a development dependency. During the installation, NPM will create a folder called **node_modules**. You may find dozens of packages when we only installed a single package.

Typically, packages will install additional packages. It's rare for a package to be installed. A comprehensive list of packages can be found in the **package-lock.json** file. You will never need to edit this file or the **node_modules** folder.

Resources

- WordPress Scripts

Upgrading WP Scripts

In this lecture, we took a closer look at the `scripts` object in the **package.json** file. The `scripts` object allows us to add custom commands so that we don't have to type long commands in the command line. You can think of it as a list of shortcuts. By default, one command is given as an example:

```
{
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

The format for the command is the name of the command as the property name and the command itself as the value. We can execute commands by running `npm run <command name here>`. So, for example, we can run the `test` command with the following command: `npm run test`

The WordPress scripts package provides a convenient list of commands that we can add to our **package** file. They're the following:

```
{
  "scripts": {
    "build": "wp-scripts build",
    "check-engines": "wp-scripts check-engines",
    "check-licenses": "wp-scripts check-licenses",
    "format": "wp-scripts format",
    "lint:css": "wp-scripts lint-style",
    "lint:js": "wp-scripts lint-js",
    "lint:md:docs": "wp-scripts lint-md-docs",
    "lint:md:js": "wp-scripts lint-md-js",
    "lint:pkg-json": "wp-scripts lint-pkg-json",
    "packages-update": "wp-scripts packages-update",
    "plugin-zip": "wp-scripts plugin-zip",
    "start": "wp-scripts start",
    "test:e2e": "wp-scripts test-e2e",
    "test:unit": "wp-scripts test-unit-js"
  }
}
```

From this long list of commands, we ran the `packages-update` command. This command will update your packages to the latest version, including the scripts package.

Creating a Bundle

In this lecture, we use the WordPress Scripts package to create a bundle from our project. To get started, we must create a directory called **src**, which is common for most projects. All our block code must originate from this directory. Next, we must create a file called **index.js**. Inside this file, we added the following contents:

```
console.log("Test");
```

Behind the scenes, the scripts package uses Webpack for processing our code. We won't have to configure Webpack since it's already configured for WordPress. We can create a bundle by using the `npm run start` command.

This command should create a directory called **build**. There's another command that produces the same files called `npm run build`. The difference between the commands is that the `start` command will watch our files for changes and update the bundle, whereas the `build` command will produce the bundle for production.

Resources

- Webpack

Creating a Block Metadata File

In this lecture, we created a block metadata file. In the **src** directory, we can create a **block.json** file containing our block's metadata. We added this file to our plugin with the following code:

```
{
  "$schema": "https://raw.githubusercontent.com/WordPress/gutenberg/t
  "apiVersion": 2,
  "name": "udemy-plus/fancy-header",
  "title": "Fancy Header",
  "category": "text",
  "icon": "star-filled",
  "description": "Adds a header with an underline effect",
  "keywords": ["header", "hover", "underline"],
  "version": "1.0.0",
  "textdomain": "udemy-plus"
}
```

We added the following properties:

- `apiVersion` - The current format of the file.
- `name` - A unique ID for the block with the format being a namespace followed by the name of the block.
- `title` - A human-readable name that's displayed on the front end.

- `category` - The category to put the block under.
- `icon` - An icon for the block.
- `description` - A human-readable description of our block that will be displayed on the front end.
- `keywords` - An array of keywords to help clients find your block in the editor.
- `version` - The current version of your block.
- `textdomain` - The textdomain of your block for translations.

Resources

- Metadata
- Dashicons

Registering a Block

In this lecture, we registered a block by using a function called `register_block_type()`. Before doing so, we outsourced the function's logic into a separate file. In the **index.php** file, we added the following code:

```
// Setup
define('UP_PLUGIN_DIR', plugin_dir_path(__FILE__));

// Includes
include(UP_PLUGIN_DIR . 'includes/register-blocks.php');

// Hooks
add_action('init', 'up_register_blocks');
```

Firstly, we're defining a constant called `UP_PLUGIN_DIR` for providing a path relative to our plugin. We're using a function called `plugin_dir_path()` to help us grab the path to our plugin. It has one argument, which is the main plugin file. It needs this information since multiple plugins can be activated, and WordPress doesn't know where to find our plugin.

The `__FILE__` constant is defined by the PHP language. This constant's value is unique to each file that points to the current file its being used.

Afterward, we're including the file with the `include()` function. We're appending two strings with the concatenation operator (`.`).

Lastly, we're running a function called `up_register_blocks` when the `init` hook is triggered. We should register our blocks as early as possible. This hook is triggered once WordPress is initialized.

We created a folder called **includes**. Not required to create this folder, but it can be useful to outsource a plugin's logic into a separate folder for

organization reasons. Inside this folder, we created a file called **register-blocks.php** with the following code:

```
function up_register_blocks() {
  register_block_type(
    UP_PLUGIN_DIR . 'build/block.json'
  );
}
```

We are using the `register_block_type()` function, which has one argument. It's the path to the **block.json** file. From there, WordPress will use the information inside the **block.json** file to register a block.

Resources

- register_block_type() Function

Enqueueing a Block's Script

In this lecture, we enqueued our block's script by using the **block.json** file. In this file, there are three properties that we can add.

- `editorScript` - Loads the script on the editor.
- `viewScript` - Loads the script on the frontend, AND the block appears on the page.
- `script` - Loads the script on both the editor and frontend.

For this example, we used the `editorScript` property since we don't need to load the script on the frontend. In the **block.json** file, we added the following code:

```
{
  "editorScript": "file:./index.js"
}
```

The value must start with the word `file`. This will tell WordPress that the script can be found locally. Following this word, we must provide a path relative to the **block.json** file. In our case, the **block.json** and **index.js** files sit in the same directory.

Adding a Custom Block

In this lecture, we added a UI for our custom block by updating our script file. A UI can be added by using the `registerBlockType()` function from the

`@wordpress/blocks` package. Even though we haven't installed this package, the `@wordpress/scripts` package will allow us to use it.

Check out the resource section for a link to the complete list of packages that are available. All of them can be found in the **packages** directory.

We updated the **index.js** file to the following:

```
import { registerBlockType } from "@wordpress/blocks";
import block from "./block.json";

registerBlockType(block.name, {
  edit() {
    return <p>Fancy Header</p>;
  },
});
```

We're calling the `registerBlockType()` function. It has two arguments. The first argument is the name of the block. In this example, we're importing our **block.json** file and using it as an object in our file. We could hardcode the name, but using the block file as a single source of truth will reduce the likelihood of a typo.

The second argument is an object of configuration settings. You can add the same settings as the **block.json** file. However, its recommended to outsource settings to the block file. WordPress provides this option if you need to use dynamic values since JSON is not a programming language.

Inside this object, we passed in a function called `edit`. This function will be treated as a component, so it should return JSX. In this example, we're returning a `<p></p>` element. If you were to add this block to the Gutenberg editor, a paragraph element should appear.

Resources

- Gutenberg Repo
- registerBlockType() Function

The RichText Component

In this lecture, we got started with the `RichText` component to allow clients to edit a block's content. WordPress defines dozens of components to help us build our block. They're completely optional to use but incredibly beneficial for rapidly building blocks.

This component can be found under the `@wordpress/block-editor` package. We updated our file to the following:

```
import { registerBlockType } from "@wordpress/blocks";
import { RichText } from "@wordpress/block-editor";
import { __ } from "@wordpress/i18n";
import block from "./block.json";

registerBlockType(block.name, {
  edit() {
    return <RichText tagName="h2" placeholder={__("Heading", "udemy-p
  },
});
```

In addition to the `RichText` component, we imported the `__()` function from the `@wordpress/i18n` package. This package will supply us with functions for translating our plugin. The value returned by this function is a translated string.

Inside our `edit()` function, we're returning the `<RichText />` component. We're adding two properties called `tagName` and `placeholder`. The `tagName` property will change the tag surrounding the editable text. By default, this property will be set to `div`. In our example, we're changing it to an `h2` tag since we're developing a header block.

As for the `placeholder` property, it functions the same as the `placeholder` attribute for the `<input>` element. It'll output temporary text if the component is missing a value. For the value, we're using the `__()` function, which will translate a string. The first argument is the text to translate, and the second argument is the text-domain.

Resources

- RichText Component

Storing Data in Attributes

In this lecture, we updated our block by adding attributes. Attributes are data for blocks that Gutenberg will store for you when the block is saved. First, we must updated the **block.json** file to list attributes. In this file, we added the `attributes` property with the following value:

```
{
  "attributes": {
    "content": {
      "type": "string"
    }
  }
}
```

The `attributes` property is a list of data our block will store. The property name will be the name of the attribute, while the value will be settings for the attribute. In this example, we're setting the data type. The following data types are supported: `null`, `boolean`, `object`, `array`, `string`, `integer`, and `number`.

Next, we updated our script to use this attribute. The `edit()` function has been modified to the following:

```
edit({ attributes, setAttributes }) {
  const { content } = attributes;

  return (
    <RichText
      tagName="h2"
      placeholder={ __('Heading', 'udemy-plus') }
      value={ content }
      onChange={ newVal => setAttributes({ content: newVal }) }
    />
  );
}
```

Since we're dealing with a component, WordPress will provide our component with various properties and methods via the props. We're destructuring this parameter to grab the `attributes` and `setAttributes` properties. The `attributes` property will be an object of our data, while the `setAttributes` function will help us update our attribute.

In the `<RichText />` component, we added the `value` prop to set the value of the component. As for updating the attribute, we're listening for the `onChange` event. This event will get triggered when the user types in the component. We are passing in a function to accept the new value and using it to update the `content` attribute.

Resources

- Attributes

Saving the Block

In this lecture, we added the `save()` function to our block. This function will be used for saving the final output of the block. After it's been saved in the database, WordPress will display the block on the frontend with the generated markup. From this function, we can return the final output.

```
save({ attributes }) {
  const { content } = attributes
```

```
  return (
    <RichText.Content
      tagName="h2"
      value={ content }
    />
  );
}
```

From the above example, we're doing the same thing as our `edit()` function with one exception. We're using the `<RichText.Content>` component. This component will render content but without the tools for modifying the content. It's a barebones version of the `<RichText>` component.

Server-side rendering vs Client-side rendering

WordPress allows us to render our blocks on the server or client. Server-side rendering is performed with PHP. Before the content is presented on the page, we can generate the output of a block with PHP, whereas client-side rendering uses JavaScript. The final output of a block is generated directly in the browser. The final output is saved in a database and presented to the user when the page loads.

There are pros and cons to each approach, which we'll talk about after exploring both options.

Aside: JavaScript Spread Operator

In this lecture, we explored the spread operator. It's an operator in JavaScript for merging arrays and object. An array can be merged with the following syntax:

```
const preregistered = ["John", "Jane"];
const newlyRegistered = ["James", "Luis"];

const complete = [...preregistered, ...newlyRegistered];

console.log(complete);
```

The spread operator is written with three dots(`...`). The array's values get spread into the current array. In this example, we're merging the `newlyRegistered` and `preregistered` arrays into the `complete` array.

The order does impact how values get arranged. For example, we can swap the order of values by swapping the variable like so:

```
const complete = [...newlyRegistered, ...preregistered];
```

We can merge objects with the same syntax. Here's an example:

```
const ticket = {
  name: "Luis",
  price: 20,
};

const newTicket = {
  ...ticket,
  name: "John",
};

console.log(newTicket);
```

In this example, the `ticket` object's properties get merged into the `newTicket` object. The order does matter. JavaScript will prioritize the last property that has a duplicate as the value to add to the object.

Resources

- JavaScript Playground

Adding Block Props

In this lecture, we added additional attributes and properties to block that is generated by WordPress. By allowing WordPress to inject properties into our block, it can add helpful UI features for our block, such as formatting options and moving the block around. We can integrate support by importing a function called `useBlockProps()` from the `@wordpress/block-editor` package like so:

```
import { useBlockProps } from "@wordpress/block-editor";
```

Next, we can use this function by spreading the return value into the root element of our block. In this case, we spread the properties on the `<RichText>` component in both the `edit()` and `save()` functions.

```
registerBlockType(block.name, {
  edit({ attributes, setAttributes }) {
    const { content } = attributes;
    const blockProps = useBlockProps();

    return (
      <RichText
        {...blockProps}
        tagName="h2"
        placeholder={__("Heading", "udemy-plus")}
        value={content}
```

```
        onChange={(newVal) => setAttributes({ content: newVal })}
      />
    );
  },
  save({ attributes }) {
    const { content } = attributes;
    const blockProps = useBlockProps.save();

    return <RichText.Content {...blockProps} tagName="h2" value={cont
  },
});
```

For the `save()` function, we're using a variation of the function called `useBlockProps.save()`. This function will add props that are only necessary for the front end. It will not include props that would allow the visitor to modify the block.

Enqueuing Styles for Blocks

In this lecture, we enqueued our styles for the block. Refer to the resource section for the final CSS for our block. Inside the **source** directory, we created a file called **main.css**. This file was enqueued from the **src/index.js** file.

```
import "./main.css";
```

Normally, JavaScript does not support CSS imports. However, by using Webpack, we can import CSS. It'll be extracted into a separate file that is included with our build. Inside the **build** directory, a file called **index.css** will be created. This file can be enqueued with the **block.json** file.

There are two properties for enqueueing files called **style** and **editorStyle**. The **style** property will enqueue a stylesheet on the front end and editor. Whereas the **editorStyle** property will enqueue a file on the editor only. For this example, we used the **style** property.

```
{
  "style": "file:./index.css"
}
```

In addition, we passed in an object to the **useBlockProps** function to add a custom class. In the `edit()` function, we changed it to the following:

```
const blockProps = useBlockProps({
  className: "fancy-header",
});
```

In the `save()` function, we changed it to the following:

```
const blockProps = useBlockProps.save({
  className: "fancy-header",
});
```

Resources

- Fancy Header CSS

Fixing the Fancy Header Styles

In this lecture, we adjusted the styles for the fancy header. In the editor, the text is centered when it should sit on the left side of the editor. This has to do with the styles and elements added to the editor. WordPress adds margins to the block for positioning the blocks in the center of the page.

We fixed our issue by surrounding the `<RichText />` component like so:

```
<div {...blockProps}>
  <RichText
    className="fancy-header"
    tagName="h2"
    placeholder={__("Heading", "udemy-plus")}
    value={content}
    onChange={(newVal) => setAttributes({ content: newVal })}
  />
</div>
```

We moved the `blockProps` object from the `<RichText />` component to the `<div>` tag. Lastly, we added the `className` property to the `<RichText />` component. In addition, the `useBlockProps` function had to be updated since we're not applying the `fancy-header` class to the root element.

```
const blockProps = useBlockProps();
```

This was only updated in the `edit()` function and not the `save()` function.

Formatting the RichText Component

In this lecture, we modified the formatting options that appear in the toolbar by adding the `allowedFormat` property to the `<RichText />` component like so:

```
<RichText
  className="fancy-header"
  tagName="h2"
  placeholder={__("Heading", "udemy-plus")}
  value={content}
  onChange={(newVal) => setAttributes({ content: newVal })}
  allowedFormats={["core/bold", "core/italic"]}
/>
```

The following formats can be added/removed.

- core/bold
- core/code
- core/image
- core/italic
- core/keyboard
- core/link
- core/strikethrough
- core/subscript
- core/text-color
- core/underline

The InspectorControls and PanelBody Components

In this lecture, we got started with adding a color picker to the editor when our component is selected. We can use two components to accomplish this task called `InspectorControls` and `PanelBody`. The `InspectorControls` component can be found under the `@wordpress/block-editor` package, and the `PanelBody` component can be found under the `@wordpress/components` package.

```
import {
  useBlockProps,
  RichText,
  InspectorControls,
} from "@wordpress/block-editor";
import { PanelBody } from "@wordpress/components";
```

Next, we updated our component to the following:

```
edit({ attributes, setAttributes }) {
  const { content } = attributes;
  const blockProps = useBlockProps()

  return (
    <>
```

```
        <InspectorControls>
          <PanelBody title={ __('Colors', 'udemy-plus') }>
            Test
          </PanelBody>
        </InspectorControls>
        <div { ...blockProps }>
          <RichText
            className="fancy-header"
            tagName="h2"
            placeholder={ __('Heading', 'udemy-plus') }
            value={ content }
            onChange={ newVal => setAttributes({ content: newVal }) }
            allowedFormats={ [ 'core/bold', 'core/italic' ] }
          />
        </div>
      </>
    );
}
```

A couple of things worth noting here. Firstly, we're wrapping the `<InspectorControls />` component and `<div>` tag with a fragment since only one root parent element is allowed. Secondly, we're placing the `<InspectorControls />` component outside of our block. This is so that WordPress grabs the correct content.

Inside this component, we are outputting a `<PanelBody />` component, which will output a panel on the sidebar that's toggleable. A title can be added by adding the `title` property. Lastly, we're inserting content inside this component to render raw text. Overall, WordPress should be able to add this component to the sidebar without us doing anything else.

Adding a Color Palette

In this lecture, we added a color palette to our block by using a component called `ColorPalette`. This component can be found in a package called `@wordpress/components`. You can refer to the resource section for more info on this package. WordPress offers various components that we can use in our plugin.

To begin, we updated the **block.json** file to add an attribute for storing the color selected by the client. We called the atrribute `underline_color`.

```
{
  "attributes": {
    "underline_color": {
      "type": "string",
      "default": "#F87171"
    }
  }
}
```

The `type` property is being set to `string`. In addition, we are setting a default value by adding the `default` property. After updating the block file, we updated the `edit()` function for our block to destructure this attribute.

```
const { content, underline_color } = attributes;
```

Next, we imported the `ColorPalette` component from the `@wordpress/components` package.

```
import { PanelBody, ColorPalette } from "@wordpress/components";
```

Lastly, we inserted this component into the `PanelBody` component.

```
<PanelBody title={__("Colors", "udemy-plus")}>
  <ColorPalette
    colors={[
      { name: "Red", color: "#F87171" },
      { name: "Indigo", color: "#818CF8" },
    ]}
    value={underline_color}
    onChange={(newVal) => setAttributes({ underline_color: newVal })}
  />
</PanelBody>
```

On this component, we added three properties.

- `colors` - An array of colors where each color is an object with a name and color.
- `value` - The current value that should be selected.
- `onChange` - An event that gets fired when a new color is selected. The function that gets called will be provided with the new color.

Resources

- Components Package

Dynamic Styles

In this lecture, we applied the color selected by the client. We only updated the `save()` function since the `<RichText />` component does not allow custom styles on the component in the editor. In the `save()` function, we updated the `useBlockProps.save()` function to the following:

```
const { content, underline_color } = attributes;
```

```
const blockProps = useBlockProps.save({
  className: "fancy-header",
  style: {
    "background-image": `
      linear-gradient(transparent, transparent),
      linear-gradient(${underline_color}, ${underline_color});
    `,
  },
});
```

A style can be added by adding the `style` property. This property will be an object of CSS properties to apply to the object where the name of the property is the CSS property, and the value is a valid CSS value. In this example, we are setting the `background-image` property to a linear gradient. We also destructured the `underline_color` attribute.

Attribute Sources

In this lecture, we updated the source for the `content` attribute. By default, attribute values are stored in an HTML comment. If a value can be found from within the HTML of a block, you can tell Gutenberg to search for a value from within our HTML. We updated the `content` attribute to the following:

```
{
  "attributes": {
    "content": {
      "type": "string",
      "source": "html",
      "selector": "h2"
    }
  }
}
```

The `source` property can be added to specify that the value can be found inside the block's HTML by setting this property to `html`. Up next, we added the `selector` property to tell Gutenberg where in our HTML it can find the value. Since blocks can be made up of several elements, we must specify exactly where to find the value. The value can be a valid CSS selector. In this case, we're telling Gutenberg to select the `h2` tag.

Creating Multiple Blocks

In this lecture, we refactored our plugin in preparation for the development of multiple blocks. The `@wordpress/scripts` package supports multiple blocks by creating a **block.json** file for each block. From within this file, it'll use the `script`, `editorScript`, or `viewScript` to find the JS files for a specific block.

For our plugin, we created a **blocks** directory from within the **src** directory. In this directory, we are going to create a folder for each block. We moved the current files into a folder called **fancy-header**.

Afterward, we updated the **register-blocks.php** file to the following:

```
function up_register_blocks() {
  $blocks = [
    ['name' => 'fancy-header']
  ];

  foreach($blocks as $block) {
    register_block_type(
      UP_PLUGIN_DIR . 'build/blocks/' . $block['name']
    );
  }
}
```

We created a multidimensional array called `$blocks`. A multidimensional array is a fancy word for when an array contains more arrays. Each array inside the `$blocks` array will represent a single block. In this example, the first array represents the **Fancy Header** block. PHP arrays can have named indexes. We're not limited to numeric indexes. We can assign a name by adding a string to represent the name followed by a `=>` and the value.

Next, we looped through the array with the `foreach` keyword. This is another solution to looping through arrays. This keyword accepts the array where each item in the array is looped through and assigned to a variable with the `as` keyword. In this example, we're assigning each item to a variable called `$blocks`.

Lastly, we updated the `register_block_type()` function to load the **block.json** file by using the `$block` array. We can reference an item with a named index by passing in a string to the `[]` portion. We're not providing the **block.json** filename since WordPress will automatically search for a file called **block.json** if a directory is given.

After all that, our plugin is ready for multiple blocks. All we have to do is create a folder for that block, add a **block.json** file, and update our array with the name of the folder.

Server-Side Rendering

In this section, we started creating custom blocks that utilize server-side rendering.

What is Server-side rendering?

In this lecture, we got into what server-side rendering is and what makes it different from client-side rendering. The first block we built was rendered on the client. WordPress will call the `save()` function and store the output in the database. Additional processing is not performed.

Whereas server-side rendering takes place on the server. Unlike client-side rendering, an HTML comment of our block is stored in the database as a placeholder. When a page is being requested with a block, the content is created with a PHP function.

In most cases, client-side rendering is faster but does not allow for dynamic content. You will want to use server-side rendering whenever your content changes from time to time. This way, clients don't need to update every post that needs an update.

Creating a Search Form Block

In this lecture, we created a search form block. WordPress already has a block for performing a similar task, but it's not easily customizable. Therefore, we need to resort to a custom block to achieve a custom behavior and appearance. Inside this **src/blocks** directory, we created a folder called **search-form** with the following files:

- **block.json**
- **index.js**
- **main.css**

Next, we updated the **block.json** file with the following code:

```
{
  "$schema": "https://raw.githubusercontent.com/WordPress/gutenberg/t
  "apiVersion": 2,
  "name": "udemy-plus/search-form",
  "title": "Search Form",
  "category": "widgets",
  "description": "Adds a search form",
  "keywords": ["search form"],
  "version": "1.0.0",
```

```
  "textdomain": "udemy-plus",
  "editorScript": "file:./index.js",
  "style": "file:./index.css",
  "attributes": {}
}
```

In addition, we created a code snippet for generating a barebones **block.json** file. Refer to the video of this lecture for the entire process.

After adding the block metadata, we updated the **register-blocks.php** file to include this block. We updated the `$blocks` array like so:

```
$blocks = [
  ['name' => 'fancy-header'],
  ['name' => 'search-form'] // <~ NEW BLOCK
];
```

Lastly, we updated the **index.js** file for the search form block with the following code:

```
import { registerBlockType } from "@wordpress/blocks";
import block from "./block.json";

registerBlockType(block.name, {
  edit() {
    return <p>Search Form</p>;
  },
});
```

Adding a Custom Icon

In this lecture, we added a custom icon. By default, WordPress provides a font icon set called Dashicons. While great, a custom icon can give a premium feel and help your blocks stand out from the crowd.

In the resources section, I provide an SVG image to use as an icon. If you don't know what an SVG image is, that's perfectly fine. It's an image created with code that's similar to HTML. Knowledge of SVG is not required.

We grabbed the code from the SVG image and stored it in a file called **icons.js** like so (The entire image was cut out from the code snippet to prevent the PDF from becoming bloated.):

```
export default {
  primary: <svg></svg>,
};
```

Next, we updated the **index.js** file to use this icon:

```
import { registerBlockType } from "@wordpress/blocks";
import block from "./block.json";
import icons from "../../icons";

registerBlockType(block.name, {
  icon: icons.primary,
  edit() {
    return <p>Search Form</p>;
  },
});
```

Lastly, the `icon` property from the **block.json** file was removed since the icon was being added manually as a property to the object passed into the `registerBlockType()` function.

Resources

- Icon File

Adding the Template

In this lecture, we added the template and imported a CSS file to our main block file like so:

```
import { registerBlockType } from "@wordpress/blocks";
import { useBlockProps } from "@wordpress/block-editor";
import block from "./block.json";
import icons from "../../icons";
import "./main.css";

registerBlockType(block.name, {
  icon: icons.primary,
  edit() {
    const blockProps = useBlockProps();

    return (
      <div {...blockProps}>
        <h1>Search: Your search term here</h1>
        <form>
          <input type="text" placeholder="Search" />
          <div className="btn-wrapper">
            <button type="submit">Search</button>
          </div>
        </form>
      </div>
    );
  },
});
```

The CSS can be found in the resource section. It should be added to the **main.css** file. Not everything worked out of the box. We had to add a selector for the search form to look the same as it did on the front end.

```
.wp-block-udemy-plus-search-form h1,
.editor-styles-wrapper .wp-block-udemy-plus-search-form h1 {
  margin-bottom: 1rem;
  font-size: 1.5rem;
  line-height: 2rem;
  font-weight: 700;
  color: inherit;
}
```

This is normal since our blocks share CSS with the Gutenberg editor and theme.

Resources

- Search Form CSS

The PanelColorSettings Component

In this lecture, we added the `PanelColorSettings` component to render a component for selecting colors. It's very different from the `ColorPalette` component since it'll present colors from a theme to the user. It can also handle multiple colors. To get started, we modified the `attributes` object in the **block.json** file for the **Search Form** block.

```
{
  "attributes": {
    "bgColor": {
      "type": "string",
      "default": "#F87171"
    },
    "textColor": {
      "type": "string",
      "default": "#fff"
    }
  }
}
```

We're storing two colors for the background and the text color of the block. Next, we updated the **index.js** block file for the **Search Form** block by importing the necessary components and functions:

```
import {
  useBlockProps,
  InspectorControls,
```

```
  PanelColorSettings,
} from "@wordpress/block-editor";
import { __ } from "@wordpress/i18n";
```

Next, we destructured the `attributes` object for readability.

```
edit({ attributes, setAttributes }) {
  const { bgColor, textColor } = attributes;

  // Rest of code here...
}
```

Lastly, we updated the JSX returned by the `edit()` function:

```
<>
  <InspectorControls>
    <PanelColorSettings
      title={__("Color Settings")}
      colorSettings={[
        {
          label: __("Background Color", "udemy-plus"),
          value: bgColor,
          onChange: (newVal) => setAttributes({ bgColor: newVal }),
        },
        {
          label: __("Text Color", "udemy-plus"),
          value: textColor,
          onChange: (newVal) => setAttributes({ textColor: newVal }),
        },
      ]}
    />
  </InspectorControls>
  <div {...blockProps}>
    <h1>Search: Your search term here</h1>
    <form>
      <input type="text" placeholder="Search" />
      <div className="btn-wrapper">
        <button type="submit">Search</button>
      </div>
    </form>
  </div>
</>
```

We surrounded the template with fragments since we're using the `InspectorControls` component. Inside this component, we added the `<PanelColorSettings />` component. This component has two properties. The first property is called `title`, which will appear in the panel's header.

The second property is an array of colors called `colorSettings`. Inside this array, we can add an object to represent a color settings. The object must have the following properties:

- `label` - A human-readable label for the color setting.
- `value` - The current color that should be selected.
- `onChange` - An event that will be fired when a new value is selected.

Exercise: Applying Colors

In this lecture, we applied the colors stored in our attributes to the template of our block as an exercise. The solution was to update the `useBlockProps()` function with an object for applying styles to the root element like so:

```
const blockProps = useBlockProps({
  style: {
    "background-color": bgColor,
    color: textColor,
  },
});
```

Next, we had to apply custom styles to the `<button>` element on the page.

```
<button
  type="submit"
  style={{
    "background-color": bgColor,
    color: textColor,
  }}
>
  Search
</button>
```

Creating a Render Callback

In this lecture, we added a render callback for the **Search Form** block. We must tell WordPress what function to run for a block that uses server-side rendering. The `register_block_type()` PHP function has an optional second argument for adding options. You can refer to the link for a list of options.

We updated the **register-blocks.php** function to the following:

```
function up_register_blocks() {
  $blocks = [
    ['name' => 'fancy-header'],
    ['name' => 'search-form', 'options' => [
      'render_callback' => 'up_search_form_render_cb'
    ]]
  ];

  foreach($blocks as $block) {
    register_block_type(
```

```
        UP_PLUGIN_DIR . 'build/blocks/' . $block['name'],
        isset($block['options']) ? $block['options'] : []
      );
    }
}
```

In this example, I added an index called `options` to the **Search Form** block.. In this array, we added an option called `render_callback`, which should contain the name of the function to run.

Next, we updated the loop by using a ternary operator in the `register_block_type()` function. A ternary operator is a shorthand syntax for a conditional statement, except it'll be processed as an expression. To the right of the `?` symbol we must add a condition. In this case, we are using the `isset()` function to check if the `$block['options']` item is defined in the array. If it is, we are passing on this array to the function. Otherwise, we are passing in an empty array.

After defining these functions, we created a new directory called **blocks** inside the **includes** directory. This directory will contain our PHP functions for rendering blocks. Inside this newly created directory, we created a PHP file called **search-form.php** with the following code:

```
function up_search_form_render_cb() {
  return 'Search Form';
}
```

From our function, we must return a string that will serve as the content of the block. WordPress will handle calling our function when rendering the page.

Lastly, we included this file from the **index.php** file.

```
include(UP_PLUGIN_DIR . 'includes/blocks/search-form.php');
```

Resources

- register_block_type()

Rendering a Block with Output Buffers

In this lecture, we used output buffers so that we can have syntax highlighting for our block's HTML template. By default, PHP sends content to the browser as it's outputted by the PHP script. It is not sent all at once. Output buffers change this behavior by storing content in the server's memory. Once it's time to server the page, the content is sent to the browser all at once.

We can enable this behavior by calling the `ob_start()` function like so:

```
ob_start();
```

In our `up_search_form_render_cb()` function, we used output buffers to store the HTML without it being sent to the browser.

```
function up_search_form_render_cb() {
  ob_start();
  ?>
  <div {...blockProps}>
    <h1>Search: Your search term here</h1>
    <form>
      <input type="text" placeholder="Search" />
      <div className="btn-wrapper">
        <button type="submit" style={{
          "background-color": bgColor,
          color: textColor
        }}>Search</button>
      </div>
    </form>
  </div>
  <?php

  $output = ob_get_contents();
  ob_end_clean();

  return $output;
}
```

Most importantly, we retrieved the content from the buffer by using a function called `ob_get_contents()`. This function returns the content as a string, which we store in a variable called `$output`. Afterward, we cleaned and closed the output buffer with the `ob_end_clean()` function. This step is very important. Otherwise, our buffer may affect other plugins, or we may end up cluttering the output buffer.

Lastly, we returned the `$content` variable. By using output buffers, we can avoid using strings for storing HTML that causes us to lose syntax highlighting from the editor.

Using Attributes in a PHP Render Function

In this lecture, we grabbed the attributes for our block in our PHP function. WordPress will handle grabbing attributes from the HTML comment and pass them onto your function as an argument. In the `up_search_form_render_cb()` function, we updated it to the following:

```
function up_search_form_render_cb($atts) {
  $bgColor = esc_attr($atts["bgColor"]);
  $textColor = esc_attr($atts["textColor"]);
  $styleAttr = "background-color:{$bgColor};color:{$textColor};";
}
```

The attributes are stored as items in an array. We are storing them into separate variables for readability. Before doing so, the values are passed into the `esc_attr()` function, which will make sure that our code is safe to use from within an attribute.

Escaping vs. Sanitization

Cleaning data is common in programming. If you clean/filter data before inserting it into the database, this is considered sanitization. If you clean/filter before inserting it into the browser, this is considered escaping.

After loading the attributes, we stored the value for the `style` attribute in a variable. We are using variable interpolation, which is similar to JavaScript string templates. We can inject variables into a string by using double quotes and surrounding variables with `{}` characters.

Next, we updated the template's styles:

```
<div style="<?php echo $styleAttr; ?>"
    class="wp-block-udemy-plus-search-form">
    <h1>Search: Your Search term Here</h1>
    <form>
      <input type="text" placeholder="Search" />
      <div class="btn-wrapper">
        <button type="submit"
          style="<?php echo $styleAttr; ?>">
          Search
        </button>
      </div>
    </form>
  </div>
```

Resources

- Data Sanitization/Escaping

Grabbing the Search Query

In this lecture, we updated the template returned by our function. We applied the following changes:

```
<div style="<?php echo $styleAttr; ?>" class="wp-block-udemy-plus-sea
  <h1>
    <?php esc_html_e('Search', 'udemy-plus'); ?>:
    <?php the_search_query(); ?>
  </h1>
  <form action="<?php echo esc_url(home_url('/')); ?>">
    <input
      type="text"
      placeholder="<?php esc_html_e('Search', 'udemy-plus'); ?>"
      name="s"
      value="<?php the_search_query(); ?>"
    />
    <div class="btn-wrapper">
      <button type="submit" style="<?php echo $styleAttr; ?>">
        <?php esc_html_e('Search', 'udemy-plus'); ?>
      </button>
    </div>
  </form>
</div>
```

First, we're using the `esc_html_e()` function to output translated text. There's another function for outputting translated text called `_e()`. The differences are that the first function will escape the output while the second does not.

Afterward, we used the `the_search_query()` function to grab the current search term. This is known as a template tag, which is a function for outputting content. Most template tags can start with the word **get**. For example, another function for grabbing the search query is called `get_search_query()`. The difference is that **get** variation will return the value instead of outputting it onto the screen.

For the form, we added the `action` attribute to specify where the form data should be submitted to. We are letting WordPress generate the URL with the `home_url()` function, which accepts a path relevant to the home URL. In addition, we are passing on the value to the `esc_url()` function for security.

Lastly, we added the `name` attribute to the `<input />` element since WordPress uses query parameters for storing the search term.

Resources

- Template Tags
- the_search_query()

Adding the Form to the Template

In this lecture, we added the **Search Form** block to the **Search** template for our Udemy theme. There are no lecture notes. This lecture is meant to be watched via video.

Creating a Page Header Block

In this lecture, we got started with the **Page Header** block for the category pages. To save time, I've provided most of the code for getting started. Check out the resource section for the code. Most of the code should be familiar to you. All these files were created in a directory called **src/blocks/page-header**.

After creating those files, we updated the **register-blocks.php** by updating the `$blocks` variable to register the new block:

```
$blocks = [
  ['name' => 'fancy-header'],
  ['name' => 'search-form', 'options' => [
    'render_callback' => 'up_search_form_render_cb'
  ]],
  ['name' => 'page-header', 'options' => [
    'render_callback' => 'up_page_header_render_cb'
  ]]
];
```

Next, we created a file called **search-form.php** inside the **includes/blocks** directory with the following code:

```
function up_page_header_render_cb() {

}
```

Lastly, we updated the **index.php** file to include the new file:

```
include(UP_PLUGIN_DIR . 'includes/blocks/page-header.php');
```

Resources

- Page Header Starter Files

Exercise: Adding the RichText Component

In this lecture, we added the `RichText` component to the **Page Header** block. This was treated as an exercise. The solution for this exercise is the following:

1. Add the `content` attribute to the **block.json** file.

```
{
  "attributes": {
```

```
    "content": {
      "type": "string"
    }
  }
}
```

2. Import the `RichText` component `__` function.

```
import { useBlockProps, RichText } from "@wordpress/block-editor";
import { __ } from "@wordpress/i18n";
```

3. Update the `edit()` function with the arguments and `RichText` component:

```
edit({ attributes, setAttributes }) {
  const { content } = attributes;
  const blockProps = useBlockProps();

  return (
    <>
      <div { ...blockProps }>
        <div className="inner-page-header">
          <RichText
            tagName="h1"
            placeholder={ __('Heading', 'udemy-plus') }
            value={ content }
            onChange={ content => setAttributes({ content }) }
          />
        </div>
      </div>
    </>
  );
}
```

4. Updated the **main.css** file for applying styles to the heading.

```
.wp-block-udemy-plus-page-header h1,
.editor-styles-wrapper .wp-block-udemy-plus-page-header h1 {
  font-size: 1.875rem;
  line-height: 2.25rem;
  font-weight: 500;
  padding: 0;
}
```

The ToggleControl Component

In this lecture, we added the `ToggleControl` component to allow clients to show the category in the header or custom content. To get started, we updated the `attributes` object in the **block.json** file for storing this option.

```
{
  "attributes": {
    "showCategory": {
      "type": "boolean",
      "default": false
    }
  }
}
```

Next, we imported the `InspectorControls`, `PanelBody`, and `ToggleControl` components from their respective packages in the **index.js** file.

```
import {
  useBlockProps,
  RichText,
  InspectorControls,
} from "@wordpress/block-editor";
import { PanelBody, ToggleControl } from "@wordpress/components";
```

Next, we grabbed the `showCategory` attribute from the `edit()` function to apply to the control.

```
const { content, showCategory } = attributes;
```

Lastly, we inserted these components into the fragment element.

```
<InspectorControls>
  <PanelBody title={__("General", "udemy-plus")}>
    <ToggleControl
      label={__("Show Category", "udemy-plus")}
      checked={showCategory}
      onChange={(showCategory) => setAttributes({ showCategory })}
    />
  </PanelBody>
</InspectorControls>
```

Most of the code snippet is familiar to us. The newest thing we did was add the `<ToggleControl />` component. This component has three properties.

- `label` - The text to display next to the control
- `checked` - Whether the option should be enabled.
- `onChange` - An event that gets fired when the option is updated. We're using this option to update the `showCategory` attribute.

Toggling Content

In this lecture, we dynamically toggled the content based on the `showCategory` attribute. Traditional conditional statements are not supported in JSX. We must resort to the ternary operator, which functions exactly the same as a ternary operator in PHP. The syntax is written exactly the same.

In the **index.js** file, we updated the `<RichText />` component with the following expression:

```
{
  showCategory ? (
    <h1>{__("Category: Some Category", "udemy-plus")}</h1>
  ) : (
    <RichText
      tagName="h1"
      placeholder={__("Heading", "udemy-plus")}
      value={content}
      onChange={(newVal) => setAttributes({ content: newVal })}
    />
  );
}
```

It's completely valid to write JSX inside an expression. We are using the `showCategory` attribute to check if it's enabled. If it is, we are displaying a generic category. Otherwise, we're rendering the `<RichText />` component.

We did the same for the `help` property for the `<ToggleControl />` component as an exercise. The `help` property will render text below the control.

```
<ToggleControl
  label={__("Show Category", "udemy-plus")}
  help={
    showCategory
      ? __("Category Shown.", "udemy-plus")
      : __("Custom Content Shown", "udemy-plus")
  }
  checked={showCategory}
  onChange={(showCategory) => setAttributes({ showCategory })}
/>
```

Rendering the Category

In this lecture, we finished the block by updating the render function in the **page-header.php** file with the following code:

```
function up_page_header_render_cb($atts) {
  $heading = esc_html($atts['content']);

  if($atts['showCategory']) {
```

```
    $heading = get_the_archive_title();
  }

  ob_start();
  ?>
  <div class="wp-block-udemy-plus-page-header">
    <div class="inner-page-header">
      <h1><?php echo $heading; ?></h1>
    </div>
  </div>
  <?php
  $output = ob_get_contents();
  ob_end_clean();

  return $output;
}
```

Most of this should be familiar to you. Here's what we did that's new. Firstly, we are using a function called `esc_html()` for escaping the content from the attribute. Secondly, we checked if the `showCategory` attribute is set to `true`. If it is, we are reassigning the `$heading` variable to the current category with the `get_the_archive_title()` function.

After making these changes, we added this block to the **Category** template for the **Udemy** theme. However, we will encounter issues if the client decides not to add a value for the `content` attribute.

During the video, we encountered an error that stated we had an **Undefined Index**, which is related to arrays. This error gets thrown if an array is missing an index that is referenced, whether it's a named or numeric index. This issue can arise if your attributes don't have a value during updates.

We can avoid this issue by always adding a default value to our attributes. In the **block.json** file, we updated the `content` attribute.

```
{
  "attributes": {
    "content": {
      "type": "string",
      "default": ""
    }
  }
}
```

Adding Support for Output Buffers

In this lecture, we took a look at adding support for output buffers if a hosting provider disables this feature. Ultimately, hosting providers have the final say. You can message your hosting provider to enable output buffering if they disable it. In most cases, output buffering is enabled since its part of the core.

Output buffering can be enabled by editing the **php.ini** file, which can be found in the **conf/php** directory of a Local project. You can add the following value to enable output buffering:

```
output_buffering = On
```

Alternatively, to enable output buffering and limit the buffer to a specific size, use a numeric value instead of `on`. For example, to set the maximum size of the output buffer to 4096 bytes, modify the `output_buffering` directive in the **php.ini** file as follows:

```
output_buffering = 4096
```

To disable output buffering, modify the `output_buffering` directive in the php.ini file as follows:

```
output_buffering = off
```

Resources

- PHP Configuration File

Including Files with Globs

In this lecture, we optimized our solution for including files by letting PHP search them with a tool called Glob. Glob can search for files within directories based on patterns. Inside the **index.php** file, we replaced the `include()` functions with the following code:

```
$rootFiles = glob(UP_PLUGIN_DIR . "includes/*.php");
$subdirectoryFiles = glob(UP_PLUGIN_DIR . "includes/**/*.php");
$allFiles = array_merge($rootFiles, $subdirectoryFiles);

foreach ($allFiles as $filename) {
  include_once($filename);
}
```

The Glob tool can be used with the `glob()` function. We can pass in a path and pattern to search for files in. In this example, we are using the `*` character to act as a wildcard. The first variable will store an array of files found in the **includes** directory. The second variable will store an array of files found in subdirectories. Lastly, we're merging both arrays with the `array_merge()` function to create a single loop.

Afterward, we're looping through the `$allFiles` array to begin including files. We're using the `include_once()` function to include a file and prevent files from being included twice.

Include vs Require

There are two types of functions for including files called `include()` and `require()`. Both will accomplish the same task. However, the `include()` function will throw a warning and allow the rest of the script to run. Whereas the `require()` function will stop the script from running if a file can't be found. In most cases, you should use the `include()` function unless you absolutely must stop WordPress from running.

Resources

- Glob

A Different Way to Create a Plugin

In this lecture, we took a look at a different way to create a block with the `@wordpress/create-block` package. This package will handle creating a plugin with the necessary files and tools. You can initialize a new plugin with the following command:

```
npx @wordpress/create-block example
```

The `npx` command will temporarily download and execute a package. It's a great way to use a package without downloading it to your system and having to manually delete it. After running this command, a new project should be created with the following files and folders:

- `.editorconfig` - A file for configuring an editor with specific settings. Helpful for teams working on different teams.
- `.gitignore` - A list of files not to commit
- `example.php` - The main plugin file
- `package.json` - A file with settings for the packages downloaded in a project.
- `readme.txt` - A file that contains a description, installation instructions, and screenshots that can be displayed on a plugin's page if you plan on uploading your plugin to the official WordPress plugin repo.
- `src` - Your block's code.

Inside the `src` directory, WordPress will structure blocks differently by outsourcing the `edit()` and `save()` functions to separate files. Other than that, most of the code is stuff you should be familiar with.

In addition, the project will be set up with Sass. Sass is a language for writing CSS. For this course, we're gonna stick with plain CSS, but you're more than welcome to check out Sass.

Resources

- Create Block Package
- Sass

Gutenberg Block Examples

In this lecture, we explored the repository for blocks developed for Gutenberg. This repo can be helpful for learning block development. Check out the link in the resource section for more info.

There are two versions of each block example. An example with JSX and tooling and another without those tools. The examples with tooling can be found with the word **-esnext** attached to the name. If you're not interested in using JSX, you can check out the examples without. However, in most cases, using tooling is the best way to go.

React.creatElement vs wp.element.createElement()

React is a package maintained by Facebook/Meta. WordPress does not work on this package. Behind the scenes, WordPress will define its own version of React's functions for backward compatibility. If React changes its API, your blocks will continue to work.

Resources

- Gutenberg Examples

Authentication

In this section, we created a series of blocks for adding authentication to our site along with extending the REST API.

Creating the Header Tools Block

In this lecture, we created the **Header Tools** block. To get started, we copied the files from the link in the resource section of this lecture. Afterward, we registered the block inside the **register-blocks.php** file.

```
$blocks = [
  ['name' => 'fancy-header'],
  ['name' => 'search-form', 'options' => [
    'render_callback' => 'up_search_form_render_cb'
  ]],
  ['name' => 'page-header', 'options' => [
    'render_callback' => 'up_page_header_render_cb'
  ]],
  ['name' => 'header-tools', 'options' => [
    'render_callback' => 'up_header_tools_render_cb'
  ]]
];
```

Next, we created a file called **header-tools.php** inside the **includes/blocks** folder. We defined the following function:

```
function up_header_tools_render_cb() {

}
```

Resources

- Header Tools Starter Files

The SelectControl Component

In this lecture, we added a `<SelectControl />` for manipulating our attribute's value. To get started, we updated the **block.json** file for our **Header Tools** block.

```
{
  "attributes": {
    "showAuth": {
```

```
        "type": "boolean",
        "default": true
      }
    }
}
```

Next, we updated the **index.js** file by importing the `SelectControl` component from the `@wordpress/components` package.

```
import { PanelBody, SelectControl } from "@wordpress/components";
```

Next, we extracted the `showAuth` property from the `attributes` object.

```
const { showAuth } = attributes;
```

Lastly, we updated the template by inserting the `SelectControl` component inside the `PanelBody` component:

```
<SelectControl
  label={__("Show Login/Register Link", "udemy-plus")}
  value={showAuth}
  options={[
    { label: __("No", "udemy-plus"), value: false },
    { label: __("Yes", "udemy-plus"), value: true },
  ]}
  onChange={(newVal) => setAttributes({ showAuth: newVal === "true" })}
/>
```

There are four properties that can be added to this component.

- `label` - A human-readable label that will be displayed with the control.
- `value` - The current value for the control.
- `options` - An array of options where each object represents an option. You can add a label and value for each option.
- `onChange` - An event that gets emitted when the value is updated.

During this process, we type cast the `newVal` argument to a boolean because the `SelectControl` component will emit a string, which is not compatible with the data type in our attribute.

The CheckboxControl Component

In this lecture, we added a `<CheckboxControl />` for manipulating our attribute's value. To get started, we updated the **index.js** file by importing the `CheckboxControl` component from the `@wordpress/components` package.

```
import {
  PanelBody,
  SelectControl,
  CheckboxControl,
} from "@wordpress/components";
```

Next, we updated the template by inserting the `CheckboxControl` component inside the `PanelBody` component:

```
<CheckboxControl
  label={__("Show Login/Register Link", "udemy-plus")}
  help={
    showAuth
      ? __("Showing Link", "udemy-plus")
      : __("Hiding Link", "udemy-plus")
  }
  checked={showAuth}
  onChange={(showAuth) => setAttributes({ showAuth })}
/>
```

There are four properties that can be added to this component.

- `label` - A human-readable label that will be displayed with the control.
- `help` - A human-readable text that will be displayed below the control.
- `checked` - Whether the checkbox should be ticked.
- `onChange` - An event that gets emitted when the value is updated.

Exercise: Toggling the Links

In this lecture, I gave you the exercise of toggling the links on the editor. We used the ternary operator to accomplish this task. The code looked like the following:

```
{
  showAuth ?
  // Template here...
  : null
}
```

In this example, there is no alternative template. In these cases, you can set the else condition to `null` to prevent JSX from rendering anything.

However, there's an alternative solution. If you don't plan on rendering an alternative template, you can use the `&&` operator. This operator will allow you to add multiple conditions. For example, take the following:

```
if (true && true) {
  // some code here
}
```

If both conditions are truthy, the block of code will execute. Both conditions must be true. Otherwise, the block of code will never run. We can use this in our JSX as a hack. JavaScript follows a fast-exit strategy. If the first condition evaluates to `false`, the second condition is never checked even if it evaluates to `true`.

We can use this operator to render the template like so:

```
{
  showAuth && // Template here...
}
```

JavaScript will check if the `showAuth` attribute is `true`. If it is, the second condition will run, which is our template. This causes the template to be rendered. If the attribute evaluates to `false`, the template will never be rendered.

Finishing the Header Tools Block

In this lecture, we finished the **Header Tools** block by finishing the PHP render function. The render function contains the typical code for creating output buffers. The unique part of this function was conditionally rendering the links. We used traditional conditional statements to render these blocks.

```
if($atts['showAuth']) {
  ?>
  <!-- Template -->
  <?php
}

if($atts['showCart']) {
  ?>
  <!-- Template -->
  <?php
}
```

We are checking the `$atts` variable in both conditional statements. The attributes are already boolean values. So, there's no need to do anything else besides checking them.

Creating the Authentication Modal Block

In this lecture, we got started with the **Authentication Modal** block. Creating this block was the same process as creating any other block. Check out the resource section for the starter files. These files should be created inside the **src/blocks/auth-modal**. Afterward, we registered a block by updating the `$blocks` array inside the **register-blocks.php** file.

```
$blocks = [
  ['name' => 'fancy-header'],
  ['name' => 'search-form', 'options' => [
    'render_callback' => 'up_search_form_render_cb'
  ]],
  ['name' => 'page-header', 'options' => [
    'render_callback' => 'up_page_header_render_cb'
  ]],
  ['name' => 'header-tools', 'options' => [
    'render_callback' => 'up_header_tools_render_cb'
  ]],
  ['name' => 'auth-modal', 'options' => [
    'render_callback' => 'up_auth_modal_render_cb'
  ]]
]
```

Lastly, we created a file called **auth-modal.php** that contains the function for rendering the block.

Resources

- Auth Modal Starter Files

Toggling the Registration Form

In this lecture, we inserted the `ToggleControl` component for toggling the `showRegister` attribute for the **Authentication Modal** block. This lecture was introduced as an opportunity for creating control components. First, we imported the `ToggleControl` component from the `@wordpress/components` package.

```
import { PanelBody, ToggleControl } from "@wordpress/components";
```

Lastly, we inserted this component into the `PanelBody` component. Here's the finished code:

```
<ToggleControl
  label={__("Show Register", "udemy-plus")}
  help={
    showRegister
      ? __("Showing Registration Form.", "udemy-plus")
      : __("Hiding Registration Form", "udemy-plus")
```

```
  }
  checked={showRegister}
  onChange={(showRegister) => setAttributes({ showRegister })}
/>
```

Rendering the Modal

In this lecture, we finished rendering the modal by updating the PHP function for the **Authentication Modal** block. Before getting started, we created a code snippet to help us create the output buffer. This will boost productivity greatly. We used the snippet extension to generate the following snippet:

```
{
  "Block Output Buffer": {
    "prefix": "block output buffer",
    "body": [
      "\tob_start();",
      "\t?>",
      "",
      "\t<?php",
      "\t\\$output = ob_get_contents();",
      "\tob_end_clean();",
      "",
      "\treturn \\$output;"
    ],
    "description": "Adds an Output Buffer for Blocks"
  }
}
```

Afterward, we inserted the template, which can be found inside the resource section. The modal contains tabs where each tab contains a login form and registration form. Since clients are able to toggle the registration form, we wrapped the elements with conditional statements.

First, we updated the function to include the `$atts` argument. Next, you can find the registration tab by searching for a comment that says **Register Tab**. We wrapped the `<li>` element with a conditional statement. The condition is the `$atts['showRegister']` attribute.

```
if($atts['showRegister']) {
  ?>
  <!-- Register Tab -->
  <?php
}
```

We applied the same condition to the `<form>` element with a comment above it that says **Register Form**.

Resources

- Authentication Modal Template

Loading a View Script

In this lecture, we added a view script, which is a script for a block that gets loaded on the front end. Inside the **src/blocks/auth-modal** folder, we created a file called **frontend.js** with a simple console statement

```
console.log("Auth Modal");
```

This file was enqueued by updating the **block.json** file. We added the `viewScript` property to load this file. You may need to restart the `start` command for the changes to take effect.

```
{
  "viewScript": "file:./frontend.js"
}
```

Waiting for the Document to be Ready

In this lecture, we decided to wait for the document to load with JavaScript since WordPress will enqueue files in the header, which can cause issues since the document hasn't been completely loaded. We used the `document.addEventListener()` function to listen for an event on the document.

JavaScript has an event system. This system allows us to listen for events on elements, such as clicks and keyboard events. In this case, we're listening for an event called `DOMContentLoaded` on the `document` object.

```
document.addEventListener("DOMContentLoaded", () => {
  console.log("test");
});
```

This event gets emitted when the document has finished loading. The first argument to the `addEventListener()` function is the name of the event. The second argument is a callback function that'll get called when the event is emitted.

Opening and Closing the Modal

In this lecture, we started toggling the appearance of the modal. We spent a lot of time writing JavaScript. First, we selected elements that have the `open-modal` class. In addition, we selected our modal for adding/removing classes on it.

```
const openModalBtn = document.querySelectorAll(".open-modal");
const modalEl = document.querySelector(".wp-block-udemy-plus-auth-mod
```

The `open-modal` class can be applied to any element that should open the modal. Afterward, we looped through the elements to add an event listener.

```
openModalBtn.forEach((el) => {
  el.addEventListener("click", (e) => {});
});
```

We must loop through the elements since we can't apply event listeners to an entire array. Looping through elements is as simple as using the `forEach()` function, which can be chained on an array. JavaScript provides methods for looping through arrays. This syntax may seem strange since we're treating an array like an object, but it's completely valid.

On each iteration, we can pass in a function to handle each item in the array. We are provided the element in the function as an argument. From there, we can listen to the `click` event. Afterward, we added the following code:

```
e.preventDefault();
modalEl.classList.add("modal-show");
```

We are calling the `preventDefault()` method since the class can be applied to links, which can cause the page to refresh. Afterward, we updated the classes on the modal element by accessing the `classList` object. This object has a method called `add()` for adding a class.

We did the same thing for closing the modal. This time, we closed the modal by removing the class with the `remove()` method like so:

```
modalEl.classList.remove("modal-show");
```

Switching Tabs

In this lecture, we started working on the tabs for the forms. This process involves adding a class called `active-tab` to the tab links and changing the `display` property of our forms. Here's the code:

```
const tabs = document.querySelectorAll(".tabs a");
const signinForm = document.querySelector("#signin-tab");
const signupForm = document.querySelector("#signup-tab");

tabs.forEach((tab) => {
  tab.addEventListener("click", (event) => {
    event.preventDefault();

    tabs.forEach((currentTab) => currentTab.classList.remove("active-
    event.currentTarget.classList.add("active-tab");

    const activeTab = event.currentTarget.getAttribute("href");

    if (activeTab === "#signin-tab") {
      signinForm.style.display = "block";
      signupForm.style.display = "none";
    } else {
      signinForm.style.display = "none";
      signupForm.style.display = "block";
    }
  });
});
```

In summary, the code does the following:

1. First, we selected the tab links, login form, and registration form
2. Looped through the tabs to add an event listener
3. Removed the `active-tab` class from the tab links
4. Add the `active-tab` class to the tab link by using the `event.currentTarget` property. This property contains the element triggering the event, which means the element should have the `active-tab` class applied to it.
5. Lastly, we toggled the form's display property through the `style` object. This object can contain valid CSS properties. JavaScript will handle adding the properties to the element.

Handling Form Submissions

In this lecture, we began handling the form submission for the registration form. By default, the browser will send the page data to a URL causing the page to refresh. We can prevent this behavior by listening to the `submit` event and calling the `preventDefault()` method from the `event` object like so:

```
signupForm.addEventListener("submit", (event) => {
  event.preventDefault();
});
```

Afterward, we can begin disabling the form and rendering a message in the form with the following code:

```
const signupFieldset = signupForm.querySelector("fieldset");
const signupStatus = signupForm.querySelector("#signup-status");

signupFieldset.setAttribute("disabled", true);
signupStatus.innerHTML = `
<div class="modal-status modal-status-info">
  Please wait! We are creating your account.
</div>
`;
```

We can disable the entire form by using the `<fieldset>` element. This element will affect children form elements. On this element, we added the `disabled` attribute with the `setAttribute()` method. This method has two arguments, which are the attribute name and value.

Lastly, we rendered the message by updating the `innerHTML` property on the `signupStatus` element. I recommend using template strings to write multiline HTML.

What is a REST API?

In this lecture, we talked about REST APIs. A REST API is able to send and receive data through HTTP. We're not limited to sending files only. Using a REST API can be more efficient than refreshing a page for a request. Data is relatively small, which can lead to faster response times. Check out the video for a better breakdown of what a REST API is.

An endpoint refers to the path at the end of a URL. Endpoints point to specific resources.

Resources

- REST API Example
- JSON Formatter
- WordPress REST API Handbook

Practicing with the Rest API

In this lecture, we got some practice with using WordPress's REST API. A REST API is available through the WordPress core. This API exposes most of WordPress's features, from users to posts. For this example, we used the Postman application for testing WordPress's REST API. You can check out the resource section for a link to download the tool.

We looked through various endpoints to understand how they were documented. Generally, WordPress will provide a schema, argument list,

definition, description, and example for an endpoint. A schema will describe the type of values returned by an endpoint. The argument list will provide a list of values we can send to WordPress.

In most examples, the definition will contain the endpoint and HTTP method. An HTTP method describes the type of action you'll be performing. The most common HTTP methods in WordPress are the following:

- `GET` - Retrieve a single or multiple resources
- `POST` - Update or create a resource
- `DELETE` - Delete a resource

The WordPress REST API is accessible under the `/wp-json/` path. After this path, we must append the endpoint. For example, if we were trying to access a list of posts, we would use the following URL: `https://example.com/wp-json/wp/v2/posts`

Resources

- Postman

The Problem with the Core Endpoint

In this lecture, we quickly took a look at a problem with WordPress's endpoint for creating a user. This endpoint is not accessible to anonymous users. It's only accessible to registered users who are logged in. I'm referring to the following endpoint: `wp/v2/users`

If we're interested in registering users, we must have a custom endpoint, which we do in the next set of lectures.

Adding a Custom Endpoint

In this lecture, we created a custom endpoint. A new endpoint must be created after the `rest_api_init` hook. We added the following line to the **index.php** file.

```
add_action('rest_api_init', 'up_rest_api_init');
```

Next, we created a folder for handling our REST API logic inside the **includes** folder called **rest-api**. Inside this directory, we added a new file called **init.php** with the following code:

```
function up_rest_api_init() {
  // example.com/wp-json/up/v1/signup
```

```
  register_rest_route('up/v1', '/signup', [
    'methods' => 'POST',
    'callback' => 'up_rest_api_signup_handler',
    'permission_callback' => '__return_true'
  ]);
}
```

We're running a function called `register_rest_route()` to add a new endpoint. This function has three arguments:

1. **Namespace** - The first URL segment after core prefix. It should be unique to your package/plugin.
2. **Route** - The base URL for the route you are adding.
3. **Arguments** - Either an array of options for the endpoint or an array of arrays for multiple methods.

In this example, the namespace for this and future endpoints will be `up/v1`. The format for a namespace should have a unique ID followed by the version of our API.

As for the third argument, we passed in an array with the following keys:

- `methods` - The HTTP method for accessing this endpoint. We're creating a user, so therefore, we should use the `POST` method.
- `callback` - A function for handling the response of the endpoint.
- `permission_callback` - A function for handling the permissions of the callback. For this key, we're using a function called `__return_true`, which is defined by WordPress. It'll always return `true` since our function should be available to everyone.

Handling the Response

In this lecture, we defined a function responsible for handling the endpoint. In the previous lecture, we set the `callback` option to `up_rest_api_signup_handler`. We defined this function inside the **includes/rest-api** folder with a file called **signup.php**. Inside this function, we created a variable called `response` like so:

```
function up_rest_api_signup_handler() {
  $response = ['status' => 1];

  $response['status'] = 2;
  return $response;
}
```

This variable will be an array with a key called `status`. This key will have a value of `1` if the request fails. Otherwise, the value for this key should be `2`.

We're returning this response. WordPress will handle converting the array into a JSON object and sending it back to the browser.

PHP: Classes

In this lecture, we took a look at PHP classes. Classes are blueprints for creating an object. A class can be defined with the `class` keyword followed by the name of the class:

```
class House {

}
```

We can add properties and methods to a class:

```
class House {
  public $color = 'red';

  function setColor($newColor) {
    $this->color = $newColor;
  }
}
```

In the example above, we added the `public` keyword to a variable to allow the variable to be accessed outside of the class. Inside the function, we are using the `$this` keyword to allow our function to access properties from within the same object. We can access a property or method by using the `->` operator.

We can start using our class by creating a new instance. An instance is a copy of a class. By default, PHP does not allow us to access the properties and methods without a copy of the class.

```
$myHome = new House();

echo $myHome->color;
$myHome->setColor('blue');
echo $myHome->color;
```

Resources

- PHP Playground
- Access Modifiers

Validating the Request Parameters

In this lecture, we validated the request by checking if it had the required data. We can't create a user without a username, email, or password. WordPress will pass on the request as an argument to our function. We can accept the request like so:

```
function up_rest_api_signup_handler($request) {
  // Code here
}
```

Afterward, we accepted the data by using the `get_json_params()` function and stored the data in a variable called `$params`. This function will grab JSON data from the body section of a request. Requests/responses are comprised of a header and body. They're very similar to the `<head>` and `<body>` sections of an HTML document. The header contains info on the request/response, such as the IP. Whereas the body contains data, such as form data.

```
$params = $request->get_json_params();
```

Afterward, we created a conditional statement to check if all data was present:

```
if(
  !isset($params['email'], $params['password'], $params['username'])
  empty($params['email']) ||
  empty($params['password']) ||
  empty($params['username'])
) {
  return $response;
}
```

We're using two functions called `isset()` and `empty()`. The `isset()` function will check if a variable, property, or array key has been added. It'll return a boolean value if the variable exists. The `empty()` function will check if a variable has a value. In the example above, we're checking if the `email`, `password`, and `email` keys are in the response with values. If they aren't, we're returning the `$response` to reject the request.

Resources

- WP_REST_Request

Preventing Duplicate Users

In this lecture, we continued working on the `up_rest_api_signup_handler()` function by preventing duplicate users from signing up with the same

username or email. First, we extracted the values from the JSON object into variables like so:

```
$email = sanitize_email($params['email']);
$username = sanitize_text_field($params['username']);
$password = sanitize_text_field($params['password']);
```

We're using a function for sanitizing the email called `sanitize_email()`. This function will make sure malicious data is not inserted into the database. As for the username and password, we're using the `sanitize_text_field()` function, which is meant for regular text data.

Afterward, we created a conditional statement for checking the username and email:

```
if(
  username_exists($username) ||
  !is_email($email) ||
  email_exists($email)
) {
  return $response;
}
```

The first condition will check if the username exists with the `username_exists()` function. The second condition will check if the email is correctly formatted with the `is_email()` function. Lastly, the final condition will check if the email is taken with the `email_exists()` function. All functions are defined by WordPress.

In addition, we're using the `||` operator, which will allow multiple conditions. It's different from the `&&` operator. Unlike the other operator, it'll check if any of the conditions are `true`. Not all conditions must be true for the block to be executed.

Creating a New User

In this lecture, we inserted the user into WordPress as well as to authenticate the user into the system. We used various functions to accomplish this task. To get started, we used the `wp_insert_user()` function like so:

```
$userID = wp_insert_user([
  'user_login' => $username,
  'user_pass' => $password,
  'user_email' => $email
]);

if (is_wp_error($userID)) {
```

```
    return $response;
}
```

The `wp_insert_user()` function will register a new user. It accepts an array of values. You can refer to the resources section for a link to this function. On the documentation page for this function, you will find a comprehensive list of values. For this example, we are setting the `user_login`, `user_pass`, and `user_email` keys.

The return value from this function will be the ID of the user created or an error. WordPress will wrap errors with an object to allow us to handle errors instead of letting PHP crash the script. Before proceeding further, we are checking if the `$userID` variable is an error by using the `is_wp_error()` function. If it is, we are returning the `$response` variable, thus ending the response.

Once we've created the user, we sent an email to the user to let them know we've created their account with the `wp_new_user_notification()` function.

```
wp_new_user_notification($userID, null, 'user');
```

This function has three arguments, which are the ID of the new user, the second argument is deprecated, and where to send the email. We can set the third argument to `user`, `admin`, or `both`. In most cases, sending an email to the user who registered will suffice. This function is pluggable. If you have a plugin for managing plugins, this plugin should be able to override the default email sent with a custom email.

After sending an email, we logged the user in with the following code:

```
wp_set_current_user($userID);
wp_set_auth_cookie($userID);

$user = get_user_by('id', $userID);
do_action('wp_login', $user->user_login, $user);
```

The `wp_set_current_user()` function will log the user into the system. However, they're not logged in forever. We can persist a user's login by using the `wp_set_auth_cookie()` function. This function will create a cookie that will be stored in the user's browser. Users will be able to revisit the site while staying logged in. WordPress will detect the cookie and use it to login.

Next, we're triggering an event with the `do_action()` function. So far, we've been running functions on events with the `add_action()` function. We can trigger events by passing in the name of the event to the `do_action()` function. In this case, we're emitting the `wp_login` event. Typically, this event

is triggered by WordPress. However, since we're manually logging the user in, we must trigger this event.

In addition, we're passing on the user data that was grabbed with the `get_user_by()` function. This function can grab a user by their email, username, or ID. In this example, we're grabbing a user object by their ID.

Resources

- wp_insert_user()
- wp_login Hook

Inline Scripts

In this lecture, we added an inline script for injecting a variable to expose our REST API URLs to our scripts. First, we must add the appropriate hook for loading an inline script. In the **index.php** file, we added the following hook:

```
add_action('wp_enqueue_scripts', 'up_enqueue_scripts');
```

Next, we created a folder called **front** inside the **includes** directory. Files related to the front end will be added to this folder. Inside this folder, we created the **enqueue.php** file with the `up_enqueue_scripts` function.

Next, we defined a PHP variable called `$authURLs`:

```
$authURLs = json_encode([
  'signup' => esc_url_raw(rest_url('up/v1/signup'))
]);
```

The value for this variable is an array of URLs created with the `rest_url()` function. This function will return a URL to WordPress's REST API. We can append append a path by passing in a string. Next, we escaped the URL with the `esc_url_raw()` function. Lastly, we're wrapping the array with the `json_encode()` function for compatibility with JavaScript.

Up next, we injected this object into the script with the `wp_inline_script()` function.

```
wp_add_inline_script(
  'udemy-plus-auth-modal-script',
  "const up_auth_rest = {$authURLs}",
  'before'
);
```

This function has three arguments. The first argument is the handlename of another script. The inline script does not get automatically added unless another script has been added along with it. We want this script to appear with the **frontend.** file for the **Authentication Modal** block. In your document, the handlename for a script enqueued with the **block.json** file can be found by its ID. At the end of the ID, WordPress will add **-js**, which can be excluded from the handlename.

The second argument is the JavaScript code to inject into the script. WordPress will automatically wrap your code with `<script></script>` tags. The last argument is the position of the script. Should it appear `before` or `after` the handle script? In this example, we want the script to appear before the **frontend.js** file. Otherwise, the file may not have access to our variables.

Submitting the Registration Form

In this lecture, we worked on submitting the registration form. Most of these modifications were performed in the **frontend.js** file for the **Authentication Modal** block. First, we grabbed the values from the input:

```
const formData = {
  username: signupForm.querySelector("#su-name").value,
  email: signupForm.querySelector("#su-email").value,
  password: signupForm.querySelector("#su-password").value,
};
```

If we're selecting form elements, we have access to the value through the `value` property. Afterward, we initiated the request with the `fetch()` function. This function will send a request with JavaScript without refreshing the page. We added the following code:

```
const response = await fetch(up_auth_rest.signup, {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(formData),
});
```

The first argument for this function is the URL. We passed in the variable that was injected from the previous lecture that contains the full HTTP URL to our endpoint. The second argument is an object of configuration settings. We're setting the following:

1. The HTTP method has been set to `POST`.
2. The `Content-Type` header has been set to `application/json` to indicate we're sending JSON.

3. The form data was added to the body of the request as a string with the `JSON.stringify()` function.

The value returned by the `fetch()` function is a promise, which means the function is asynchronous. Asynchronous functions mean that the line of code takes time to execute. JavaScript will not wait for this function to finish. It'll execute the next line. If we want to wait for the function to finish, we can add the `await` keyword. If we do add this keyword, the function must have the `async` keyword prefixed to it.

Lastly, we're storing the value resolved by the promise in a variable called `response`. This variable contains information on the response. We can grab the response body by using the `response.json()` function.

```
const responseJSON = await response.json();
```

Lastly, we verified the status of the response and rendered the appropriate message.

```
if (responseJSON.status === 2) {
  signupStatus.innerHTML = `
  <div class="modal-status modal-status-success">
    Success! Your account has been created.
  </div>
  `;
  location.reload();
} else {
  signupFieldset.removeAttribute("disabled");
  signupStatus.innerHTML = `
  <div class="modal-status modal-status-danger">
    Unable to create account! Please try again later.
  </div>
  `;
}
```

If the response was a success, we reloaded the page. Otherwise, we removed the `disabled` attribute from the `fieldset` element to allow the user to modify their values.

Resources

- Fetch Function

Preparing the Signin Endpoint

In this lecture, we prepared the signin endpoint. Most of this will be a repeat of the registration endpoint. Inside the **init.php** file, we added a new route with

the following info:

```
register_rest_route('up/v1', '/signin', [
  'methods' => 'POST',
  'callback' => 'up_rest_api_signin_handler',
  'permission_callback' => '__return_true'
]);
```

The most noteworthy thing about this registration is the namespace. We're reusing the same namespace since our routes should be grouped into a single route.

For the function, we created a new file called **signup.php** inside the **includes/rest-api** with the following code:

```
function up_rest_api_signin_handler($request) {
  $response = ['status' => 1];
  $params = $request->get_json_params();

  if(
    !isset($params['user_login'], $params['password']) ||
    empty($params['user_login']) ||
    empty($params['password'])
  ) {
    return $response;
  }

  $response['status'] = 2;
  return $response;
}
```

We're doing a couple of things here:

1. Grabbing the data from the request with the `$request->get_json_params()` function.
2. Validating the values from the parameters.
3. Returning the response.

User Authentication

In this lecture, we finished our signin endpoint by authenticating the user. First, we had to extract the data from the request by storing them in variables like so:

```
$email = sanitize_email($params['user_login']);
$password = sanitize_text_field($params['password']);
```

Next, we signed into WordPress by using the `wp_signon()` function. This function accepts an array where we can pass in the email, password, and if we should remember the user.

```
$user = wp_signon([
  'user_login' => $email,
  'user_password' => $password,
  'remember' => true
]);

if (is_wp_error($user)) {
  return $response;
}
```

This function returns a user object. Otherwise, it'll return an error. We're checking if the variable contains an error. If it does, we are returning the response.

Wiring the Login Form

In this lecture, we worked on the login form on the front end. There are no lecture notes for this lecture. Most of what we did was a repeat of the previous process.

Finishing the Blocks

In this lecture, we finished our blocks by hiding the authentication modal and updating the header tools to greet the user. First, we hide the authentication modal by updating the **auth-modal.php** file with the following code at the top of the function:

```
if(is_user_logged_in()) {
  return '';
}
```

We're using a function defined by WordPress called `is_user_logged_in()` to help us check if the user is logged in. If they are, we will return an empty string to hide the modal.

Next, we updated the **header-tools.php** file with the following code at the top of the function:

```
$user = wp_get_current_user();
$name = $user->exists() ? $user->user_login : 'Sign in';
$openClass = $user->exists() ? '' : 'open-modal';
```

In this example, we're using the `wp_get_current_user()` function to get the currently authenticated user's info. This function returns an object, which has a method called `exists()`. This method will tell us if a user was found. If there is a user logged in, we're storing their name in a variable.

In addition, we're creating a variable for storing the `open-modal` class. This class will trigger the modal. By dynamically adding it to an element, we can prevent errors from appearing in the console when there isn't a modal.

We updated the `<a>` element for the login link with the following code:

```
<a class="signin-link <?php echo $openClass; ?>" href="#">
  <div class="signin-icon">
    <i class="bi bi-person-circle"></i>
  </div>
  <div class="signin-text">
    <small>Hello, <?php echo $name; ?></small>
    My Account
  </div>
</a>
```

Static Properties

In this lecture, we talked about static properties. Static properties allow us to access properties without creating an instance. We can add the `static` keyword to a property like so:

```
class Car {
  public static $brand = "Tesla";
}
```

This keyword should always be applied after the access modifier. Next, we can access the property like so:

```
echo Car::$brand;
```

Instead of using `->` to access properties, static properties must be accessed with the `::` characters. As you can see, we did not have to create an instance. This removes an extra step in the process, but why?

Static properties can be useful when you don't care if a property is not unique to each instance. By using regular properties, each instance will always have a unique copy of a property. Changing a property in one instance will not be reflected in all instances. Whereas static properties are affected globally.

Resources

- PHP Playground

WordPress HTTP Methods

In this lecture, we decided to let WordPress determine the methods of our class. There's a class called `WP_REST_Server` that contains constants. Constants are created with the `const` keyword. By default, these properties are public and accessible without creating an instance.

These properties standardize methods for specific actions. There's `CREATABLE`, `EDITABLE`, `READABLE`, and `DELETEABLE`. Rather than hardcoding the methods, we can use these properties to set the appropriate method. It's not required, but it's always good practice to follow a standard. If the standard changes, you do not need to make further changes. WordPress will be updated to the latest changes.

We updated both routes to use the `CREATABLE` and `EDITABLE` properties:

```
register_rest_route('up/v1', '/signup', [
  'methods' => WP_REST_Server::CREATABLE,
  'callback' => 'up_rest_api_signup_handler',
  'permission_callback' => '__return_true'
]);

register_rest_route('up/v1', '/signin', [
  'methods' => WP_REST_Server::EDITABLE,
  'callback' => 'up_rest_api_signin_handler',
  'permission_callback' => '__return_true'
]);
```

Resources

- WP_REST_Server Class

SQL Fundamentals

In this section, we took a break from WordPress to start exploring SQL. Learn how to interact with a database through raw queries.

Introduction to SQL

In this lecture, we talked about SQL databases. WordPress primarily supports two databases called MySQL and MariaDB. MariaDB is a fork of MySQL, which is a copy of a program that was taken in a different direction. Most of your code will be compatible with both databases.

Local will provide us with a client for working with our database called Adminer. We'll be using it for this course. You can look at alternative options in the resource section of this lecture.

Resources

- MySQL
- MariaDB
- MySQL Workbench
- HeidiSQL
- Sequel Pro

Databases and Tables

In this lecture, we talked about databases and tables. First, we connected to a database by providing the following information:

- **Server** - The URL to your database.
- **Username** - The username to your database.
- **Password** - The password to your database.
- **Database** - The name of your database since multiple databases can be hosted on a single server.

You can refer to Local for the values of each field in your client. Next, we moved our discussion to tables. Tables allow you to store groups of data. You can have a post for users, posts, comments, etc. Each table will have an engine and collation.

An engine is responsible for processing queries. Each engine will perform certain actions better than others. The most common engine is `InnoDB`. This

is the default engine provided by the database, which is an all-around general-purpose engine that will cover most use cases.

The collation is the character set supported by the database. If you're interested in storing various characters, you should use the `utf8_mb4_unicode_ci`. This collation supports a wide range of characters from various languages.

Creating Tables

In this lecture, we created a table with a custom query. We typed the following query:

```
CREATE TABLE products (
  ID bigint(20) NOT NULL,
  name varchar(255)
);
```

A couple of things worth noting about this query. Firstly, reserved keywords are not case-sensitive. We can write `CREATE TABLE` as `Create Table` or `create table`. Either formatting works. Most developers prefer to use all uppercase letters for reserved keywords. Lowercase letters for custom names.

The `CREATE TABLE` keyword will create a table. This keyword is followed by the name of the table and the columns. Inside the parentheses, we're creating two columns called `ID` and `name`. They're separated with a comma.

After specifying a column name, we must add the data type. SQL languages offer various data types for the same type of data to allow us to constrain the size. It's considered good practice to only specify a size that doesn't exceed your needs. You can refer to the resource section of this lecture for a complete list of data types.

Lastly, we're using the `NOT NULL` keywords to prevent the `ID` column from accepting an empty value. Behind the scenes, MariaDB/MySQL performs validation on your data. If a value does not match the data type of a column or is empty, the value will be rejected.

Resources

- Data Types for SQL

Inserting Data

In this lecture, we talked about inserting data into a database. There are common operations you'll perform when interacting with a database called **CRUD**, which is short for Create, Retrieve, Update, and Delete. We explored each operation in this course.

We started with creating data. Data can be created with the `INSERT INTO` keywords. This is followed by the name of the table to insert data into. Next, we can specify values for each column in the table by using the `VALUES` keyword like so:

```
INSERT INTO products
VALUES (1, "Hats");
```

We must provide a list of values for each column in the table. Alternatively, we can provide a list of columns to add values for by adding a pair of `()` after the name of the table like so:

```
INSERT INTO products(ID)
VALUES (1);
```

However, this will not work if you forget to add a column that does not allow null values.

Reading Data

In this lecture, we talked about reading data from the database by using the `SELECT` keyword. After this keyword, we can provide a list of columns to retrieve values from. Next, we can specify the table to grab data from using the `FROM` keyword like so:

```
SELECT ID
FROM products
```

We can select all columns by replacing the names of columns with a `*` character like so:

```
SELECT *
FROM products
```

In some cases, you may want to filter the records from a table using the `WHERE` keyword like so

```
SELECT *
FROM products
```

```
WHERE name="Hats"
```

You can find a complete list of comparison operators in the resource section of this lecture.

Resources

- SQL Operators

Updating Data

In this lecture, we got started with updating data by using the `UPDATE` keyword. We can specify the table to update by adding the name of the table afterward. Lastly, we can add the `SET` keyword to start updating values from specific columns.

```
UPDATE products
SET name="Shirts"
WHERE ID=2
```

It's very important to add the `WHERE` keyword to update specific records. Otherwise, all records will be updated.

Deleting Data

In this lecture, we deleted data from a table by using the `DELETE FROM` keywords. We can specify the name of the table after these keywords. By itself, this query will delete all records. It's very important to add a condition to prevent a disaster from occurring by using the `WHERE` keyword.

```
DELETE FROM products
WHERE ID=2
```

Lastly, we deleted the table by using the `DROP TABLE` keyword like so:

```
DROP TABLE products
```

Custom Post Types

In this section, we got back into WordPress by exploring custom post types in WordPress.

What's Next

In this lecture, we talked about what's coming next in this section. There are no lecture notes.

Registering a Custom Post Type

In this lecture, we registered a custom post type with the `register_post_type()` function. According to the documentation, it's recommended to use the `init` hook when calling this function. We already have a function for this hook for registering blocks. Regardless, we created a new function. It's perfectly fine to have multiple functions for the same hook. This keeps our code organized.

```
add_action('init', 'up_recipe_post_type');
```

Inside the **includes** directory, we created a file called **recipe-post-type.php**. This file will contain the `up_recipe_post_type` function definition. Inside this function, we pasted in the code example that can be found in the example in the documentation. This should register a post type called `book`.

Resources

- Registering Custom Post Types

Customizing the Post Type

In this lecture, we customized the post type for recipes. From the example, we have an array called `$labels`. This contains a list of translatable text that will appear in various areas of the custom post type. This array is using two functions called `__()` and `_x()`. Both functions return translatable text.

However, the `_x()` function has an argument for providing a note to the translator. This can be helpful for providing additional info to help translators understand the message. In the end, WordPress does not use this argument. It's purely for translators.

The translations were updated to use the `udemy-plus` text domain, and the word `book` was replaced with `recipe`. After modifying the labels, we updated the options:

```
$args = array(
  'labels'              => $labels,
  'public'              => true,
  'publicly_queryable'  => true,
  'show_ui'             => true,
  'show_in_menu'        => true,
  'query_var'           => true, // ?recipe=pizza
  'rewrite'             => array( 'slug' => 'recipe' ), // /recipe/piz
  'capability_type'     => 'post',
  'has_archive'         => true,
  'hierarchical'        => false,
  'menu_position'       => 20,
  'supports'            => array( 'title', 'editor', 'author', 'thumbn
);

register_post_type( 'recipe', $args );
```

The following options are available:

- `labels` - A list of translatable text that will appear in various locations.
- `public` - Should the post type be available on the front end.
- `publicly_queryable` - Should the frontend be able to query for posts.
- `show_ui` - Should WordPress render a UI in the admin dashboard.
- `show_in_menu` - Show the post type in the menu of the WordPress dashboard.
- `query_var` - An option to rename the query parameter in the URL. Set to `true` to use the name of the post type.
- `rewrite` - An option to set the path in the URL for viewing a post.
- `capability_type` - Permissions for editing/managing this post type.
- `has_archive` - Should this post type have an archive for viewing older posts.
- `hierarchical` - Can posts have parent posts
- `menu_position` - The position of the menu in the WordPress admin dashboard. A lower number means a higher position. Set to `null` to position at the bottom of the menu.
- `supports` - A list of WordPress features that should be supported.

Lastly, we called `register_post_type()` function that has two arguments. The name of the post type and the array of arguments.

Resources

- register_post_type()

Adding Support for Gutenberg

In this lecture, we added support for Gutenberg. By default, WordPress will allow custom post types to be edited with the classic editor. This is because the REST API is not supported by custom post types unless we explicitly set the `show_in_rest` option to `true` like so:

```
$args = array(
  'description' => __('A custom post type for recipes', 'udemy-plus')
  'show_in_rest' => true,
);
```

In addition, we added the `description` option to provide a human-readable description of our post type. After adding this option, the Gutenberg editor should appear for our posts.

Taxonomies

In this lecture, we added support for the Category and Tags taxonomies in our post type. A taxonomy is the science of classification. We classify everything in our world, from biology to music genres. Posts can be classified too. By default, WordPress has two taxonomies called categories and tags.

We can add support for these taxonomies by updating the arguments to include the `taxonomies` option. This option will be an array of taxonomies. We updated this option to the following:

```
$args = array(
  'taxonomies' => ['category', 'post_tag']
);
```

After adding this option, categories and tags should appear for our post type.

Plugin Activation

In this lecture, we ran into a problem with our post type not being properly registered. WordPress will not recognize the new path for our recipes. We must register this process manually. The best point to do this is during plugin activation. From the **index.php** file, we added a new hook by using the `register_activation_hook()` function.

```
register_activation_hook(__FILE__, 'up_activate_plugin');
```

This function will register a hook for plugin activation. The first argument is the path to the main plugin file. We are using the `__FILE__` constant to grab the path to the current plugin file. Next, we must provide the name of a function to run during activation. We defined this function from within a file called **activate.php**.

```
function up_activate_plugin(){
  // 6.0 < 5.8
  if(version_compare(get_bloginfo('version'), '5.8', '<')) {
    wp_die(
      __('You must update WordPress to use this plugin.', 'udemy-plus
    );
  }
}
```

In this function, we added a conditional statement to compare the current version of WordPress with the minimal required version for WordPress. While WordPress can check the compatibility of our plugin, this feature is still new. On very old versions of WordPress, it'll be possible to activate our plugin. For extra security, we are going to perform this process manually.

In a conditional statement, we are using the `version_compare()` function, which is defined by PHP. It's capable of comparing versions with various formats. We are passing in the current version with the `get_bloginfo()` function. Next, we are passing in the minimal supported version. Lastly, we are passing in a comparison operator.

If this function returns `true`, the plugin is installed on an unsupported version of WordPress. In this case, we are killing the script with the `wp_die()` function, which will also output a neatly formatted message.

Resources

- get_bloginfo()

Flushing Rewrite Rules

In this lecture, we flushed the rewrite rules. In the database, WordPress will store a copy of all URLs. This helps WordPress identify what URL can render what type of content. This info is stored in the `wp_options` table under the name `rewrite_rules`. This table has four columns, which are the ID, name, value, and autoload.

By default, WordPress will register various settings. If the option can be found from the admin dashboard, you will probably find its value here. For the `rewrite_rules` option, there are various rules for URLs. However, this option does not include a rule for our post type.

We can add our rule by calling the `flush_rewrite_rule()` function like so from the activation file:

```
up_recipe_post_type();
flush_rewrite_rules();
```

First, we are calling the `up_recipe_post_type()` function since it contains the custom post type registration. Our post type may not be registered during this hook. Just to make sure, we're manually registering it. Next, we called the `flush_rewrite_rules()` function to update the entry in the database to include our custom path.

You can deactivate and reactivate the plugin to test the rules. If you refresh the database, the `rewrite_rules` entry should contain our new path.

Resources

- flush_rewrite_rules()

Custom Taxonomies

In this lecture, we registered a custom taxonomy with the `register_taxonomy()` function. According to the documentation, this function must be called during the `init`. For this example, we're going to reuse the `up_recipe_post_type` function for registering the taxonomy. At the bottom of the function, we added the following:

```
register_taxonomy('cuisine', 'recipe', [
  'label' =>  __( 'Cuisine', 'udemy-plus' ),
  'rewrite' =>  ['slug' => 'cuisine'],
  'show_in_rest' => true
]);
```

This function has three arguments, which are the name of the cuisine, post type, and an array of settings. In this example, we are adding a label, the unique URL for viewing a term and enabling this taxonomy for the REST API. The `show_in_rest` option is very important if you want users to be able to add taxonomy terms to the post type from the Gutenberg editor.

Resources

- register_taxonomy()

Custom Taxonomy Fields

In this lecture, we added custom taxonomy fields to the cuisine taxonomy. WordPress has a hook for inserting fields by using a hook called `{$taxonomy}_add_form_fields`. This hook has a placeholder that can be replaced with the name of the taxonomy. Anytime you see a pair of curly brackets, you can assume this is a placeholder. We used this hook like so:

```
add_action('cuisine_add_form_fields', 'up_cuisine_add_form_fields');
```

Next, we defined this function from a directory called **includes/admin**. The name of the file is called **cuisine-fields.php**.

```
function up_cuisine_add_form_fields() {
  ?>
  <div class="form-field">
    <label><?php _e('More Info URL', 'udemy-plus'); ?></label>
    <input type="text" name="up_more_info_url">
    <p>
      <?php
        _e(
          'A URL a user can click to learn more information about thi
          'udemy-plus'
        );
      ?>
    </p>
  </div>
  <?php
}
```

There's not much to say about this code aside from the fact that we're copying WordPress's classes for styling the field. This step is optional, but the UI should be consistent with WordPress's UI. In addition, the `<input />` element has a name. The `name` attribute is important as WordPress will use this name for sending the value to the backend.

Resources

- _add_form_fields Hook

Understanding Metadata

In this lecture, we talked about metadata. There are no lecture notes. This lecture is meant to be consumed via video.

Saving Term Metadata

In this lecture, we saved the term metadata by using a hook called `create_{$taxonomy}`. This hook will run when a new term is added. It has a

placeholder for the name of the taxonomy.

```
add_action('create_cuisine', 'up_save_cuisine_meta');
```

Next, we defined this function inside a file called **save-cuisine.php** from the **includes/admin** folder.

```
function up_save_cuisine_meta($term_id) {
  if(!isset($_POST['up_more_info_url'])) {
    return;
  }

  add_term_meta(
    $term_id,
    'more_info_url',
    sanitize_url($_POST['up_more_info_url'])
  );
}
```

This function has one argument, which is the ID of the term created. Before saving the metadata, we validated the post data. By default, PHP will store post data in a variable called `$_POST`. This is an array of value submitted by a form. A value can be referenced by the name given in the `name` attribute. In this example, we are checking if this variable contains the `up_more_info_url` value.

If it does, we are adding the metadata with the `add_term_meta()` function. This function has three arguments, which are the ID, the name of the metadata, and the value. Before inserting this value into the database, we are sanitizing it with the `sanitize_url()` function.

Resources

- create_taxonomy

Editing Taxonomy Fields

In this lecture, we finalized the taxonomy by creating a field in the edit form of the term along with updating the function for saving the metadata. First, we had to create the field for the edit form. WordPress has a hook called `{$taxonomy}_edit_form_fields`, which we used like so:

```
add_action('cuisine_edit_form_fields', 'up_cuisine_edit_form_fields')
```

We defined the `up_cuisine_edit_form_fields` function from within the **cuisine-fields.php** file like so:

```
function up_cuisine_edit_form_fields($term) {
  $url = get_term_meta($term->term_id, "more_info_url", true);

  ?>
  <tr class="form-field">
    <th>
      <label><?php _e('More Info URL', 'udemy-plus'); ?></label>
    </th>
    <td>
      <input type="text" name="up_more_info_url"
        value="<?php echo esc_attr($url); ?>">
      <p class="description">
        <?php
          _e(
            'A URL a user can click to learn more information about t
            'udemy-plus'
          );
        ?>
      </p>
    </td>
  </tr>
  <?php
}
```

In this function, we're using a different markup, which we created by studying the markup for the other fields. In this function, we are accepting the `WP_Term` object as an argument called `$term`. This object contains information on the current term being edited.

We used this object to retrieve the metadata. To get the metadata, we used the `get_term_meta()` function. This function has three arguments which is the ID, the name of the metadata, and whether to return an array or a single value. The ID was retrieved through the `$term` argument. This object has a property called `term_id`. It's the ID from the database.

Once we grabbed the metadata, we used it as the value for the `<input />` element.

The last step was to update the `up_save_cuisine_meta()` function. We replaced the `add_term_meta()` function with the `update_term_meta()` function. These functions have the same arguments. The main difference is that the first function will add metadata while the second function will add/update the metadata. We do not need to check if the metadata already exists. The `update_term_meta()` function performs this check on our behalf.

Resources

- edit_form_fields Hook

- WP_Term

Creating the Recipe Summary Block

In this lecture, we got started with developing a recipe summary block for the recipe post type. There are no lecture notes. You are meant to watch via video.

Resources

- Starter Files

Retrieving the Post ID with Block Context

In this lecture, we grabbed the ID of the post with block context. The block context is a feature for communicating data between the editor and block. We can accept the current post ID by adding an array called `usesContext` to the block configuration.

```
{
  "usesContext": ["postId"]
}
```

This array will contain a list of values to accept from the editor. In this case, we're accepting the `postId` value. Next, we can update the `edit()` function to destructure the `props` object for the `context` object like so:

```
edit({ context }) {
  const { postId } = context
}
```

From this object, we're destructuring the `postId` property into a variable. For more info, you can refer to the resource section of this lecture.

Resources

- Block Context

Grabbing Term IDs with useEntityProp

In this lecture, we grabbed the term IDs associated with the current post by using a function called `useEntityProp()`. WordPress provides us with this

function for retrieving and updating metadata for a post. We can grab this function from the `@wordpress/core-data` package:

```
import { useEntityProp } from "@wordpress/core-data";
```

Next, we used this function to grab the term IDs. This function has 4 arguments, which are the type of data to return, the post type, the taxonomy, and lastly, the ID of the post. This function returns the values retrieved by WordPress and a function for updating these values.

```
const [termIDs] = useEntityProp("postType", "recipe", "cuisine", post
```

In our case, we're not going to grab the function for updating the values. We'll allow WordPress to handle this process.

In WordPress, an entity is an object that represents data. This data can be anything from post data to metadata.

The useSelect Function

In this lecture, we got started with grabbing the terms for a post with their IDs. At the moment, we only have the IDs of the terms but not the names. We'll need to query the database for this info. WordPress has a function for querying the database called `useSelect()`. We can import it from the `@wordpress/data` package.

```
import { useSelect } from "@wordpress/data";
```

Next, we used this function to begin initiating a query. The function accepts a function that will be executed whenever the block changes.

```
useSelect(() => {
  console.log("useSelect() called");
}, []);
```

However, this is not ideal. If we were to leave the function as-is, the query would be constantly initiated, which can impact performance. The query should only run when the cuisine taxonomy is updated on the post. The `useSelect()` function has a second argument, which is an array of dependencies. We passed in the `termIDs` variable to limit the number of times this function runs.

```
useSelect(() => {
  console.log("useSelect() called");
}, [termIDs]);
```

Retrieving Cuisine Terms

In this lecture, we initiated the request for grabbing the cuisine terms with the IDs. WordPress has a Data API, which exposes functions for interacting with the data from the Gutenberg editor. We can also interact with custom data. First things first, we need to access a store.

Gutenberg manages massive amounts of data. To keep things organized, data is organized into stores, which is short for storage. If we're interested in accessing post data, we must access the `core` store. A list of stores can be found from the link in the resource section of this lecture. After accessing a store, we have access to functions for interacting with that specific store.

```
const { cuisines } = useSelect(
  (select) => {
    const { getEntityRecords } = select("core");

    return {
      cuisines: getEntityRecords("taxonomy", "cuisine", {
        include: termIDs,
      }),
    };
  },
  [termIDs]
);
```

In this example, we're accepting the `select()` function as an argument. This function will allow us to select a store, which is passed in as a string. Next, we are destructuring the store to grab a specific function called `getEntityRecords()`. This function allows us to grab specific records from a table in a database using the REST API.

We are returning an object, which will be returned by the `useSelect()` function so that our data is accessible outside of the function. We are using the `getEntityRecords()` function to grab terms from the database. We're grabbing the cuisine taxonomy.

Lastly, we're filtering the results by passing in an object as a third argument. The object can be a list of arguments supported by the REST API. In this case, we're limiting the results to terms with specific IDs.

Resources

- Data Module Reference
- Category Endpoint

Registering Term Meta

In this lecture, we registered the term metadata with the REST API. WordPress requires metadata to be registered for it to be exposed via the REST API. This is so that developers are aware of the type of data that is available through the REST API. We can use the `register_term_meta()` function to perform this action. We called this function at the bottom of the `up_recipe_post_type()` function:

```
register_term_meta('cuisine', 'more_info_url', [
  'type' => 'string',
  'description' => 'A URL for more information on a cuisine',
  'single' => true,
  'default' => '#',
  'show_in_rest' => true,
]);
```

This function has three arguments. The first argument is the name of the taxonomy. The second argument is the name of the metadata key. Lastly, we can provide an array of arguments to configure the metadata. We added the following values:

- `type` - The data type of the metadata.
- `description` - A short description of the metadata.
- `single` - Whether the value is a single value or an array of values.
- `default` - The default value of the metadata if it's not available for a post.
- `show_in_rest` Whether to expose the metadata in the REST API.

Resources

- register_term_meta()

Mapping the Cuisines

In this lecture, we began displaying the cuisines into the template. We inserted the cuisines into the `<div>` element with a class called `recipe-cuisine`. The following code was added:

```
{
  cuisines &&
    cuisines.map((term, index) => {
      const comma = cuisines[index + 1] ? ", " : "";
```

```
      return (
        <>
          <a href={term.meta.more_info_url}>{term.name}</a>
          {comma}
        </>
      );
    });
}
```

First, we checked if there were cuisines before looping through the array. After checking the `cuisines` variable, we're looping through this array with the `map()` function. Unlike the `forEach()` function, this function will return an array with the modified items. In this case, we're going to create elements.

We're accepting the current item and the index in the callback function. The index will help us check if there's another item in the array so that we can add a comma. In the template, we're rendering the term URL, name, and a comma.

Waiting for Data to be Resolved

In this lecture, we decided to wait for the data to be resolved. At the moment, the user will view a blank space. A better user experience would be displaying the spinner. WordPress has such a component that we can use to indicate that we're loading data.

```
const { cuisines, isLoading } = useSelect(
  (select) => {
    const { getEntityRecords, isResolving } = select("core");
    const taxonomyArgs = ["taxonomy", "cuisine", { include: termIDs }

    return {
      cuisines: getEntityRecords(...taxonomyArgs),
      isLoading: isResolving("getEntityRecords", taxonomyArgs),
    };
  },
  [termIDs]
);
```

First, we updated the `useSelect()` function. In this function, we're grabbing the `isResolving()` function. This function will watch a request to let us know if it's complete. Next, we outsourced the arguments for the request into a variable since the `isResolving()` function needs a copy of these arguments. We spread this array into the `getEntityRecords()` function.

Lastly, we created a new property called `isLoading` with the `isResolving()` function as the value. We must provide this function with the name of the function to watch along with the arguments since requests can come from different plugins/blocks.

After grabbing this information, the template was updated:

```
{
  isLoading && <Spinner />;
}
{
  !isLoading && cuisines && cuisines.map((term, index) => {});
}
```

In this example, we're checking the `isLoading` variable, which will be a boolean value. If `true`, the request is pending. If that's the case, we displayed the `<Spinner />` component, which was imported from the `@wordpress/components` package.

```
import { Spinner } from "@wordpress/components";
```

Resources

- Spinner

Saving Post Metadata

In this lecture, we saved metadata whenever a recipe post was published/updated. WordPress has a hook for this event called `save_post_{$post->post_type}`. The placeholder can be replaced with the name of the post type like so:

```
add_action('save_post_recipe', 'up_save_post_recipe');
```

Next, we defined this function inside the **includes/admin** folder called **save-recipe.php**.

```
function up_save_post_recipe($post_id) {
  if (defined( 'DOING_AUTOSAVE' ) && DOING_AUTOSAVE) {
    return;
  }

  $rating = get_post_meta($post_id, 'recipe_rating', true);
  $rating = empty($rating) ? 0 : floatval($rating);

  update_post_meta($post_id, 'recipe_rating', $rating);
}
```

This function checks if the `DOING_AUTOSAVE` constant is defined with the `defined()` function. WordPress will add this constant if the post is being

autosaved. An autosaved post may not have a post ID, which means saving metadata will not work. If the request is an autosave, we're returning the function to prevent it from running.

After checking this constant, we're grabbing the `$post_id` argument. Inside the function, we're grabbing the metadata from the post with the `get_post_meta()` function. This function has three arguments, which are the ID of the post, metadata key, and if the value is a single value.

Next, we're checking if the rating is empty; if it is, we'll convert it into a number. Otherwise, we'll save the original value with the `floatval()` function for type casting the value as a number.

Lastly, we're updating the post metadata with the `update_post_meta()` function. This function has three arguments which are the ID of the post, the name of the metadata, and the value itself.

Resources

- save*post* Hook

Registering Post Metadata

In this lecture, we updated the `recipe-post-type.php` file to expose our post metadata via the REST API by calling the `register_post_meta()` function.

```
register_post_meta('recipe', 'recipe_rating', [
  'type'               => 'number',
  'description'        => 'The rating for a recipe',
  'single'             => true,
  'default'            => 0,
  'show_in_rest'       => true,
]);
```

This function has similar arguments to the `register_term_meta()` function. We must provide the name of the post type, the name of the metakey, and a list of arguments. We're configuring the following options with our metadata:

- `type` - The data type of the metadata.
- `description` - A short description of the metadata.
- `single` - Whether the value is a single value or an array of values.
- `default` - The default value of the metadata if it's not available for a post.
- `show_in_rest` Whether to expose the metadata in the REST API.

Lastly, we updated the arguments for the custom post type by including the `custom-fields` option in the `supports` array. Without this support, the

metadata may not be retrievable with the REST API.

```
$args = array(
  'supports'           => array(
    'title', 'editor', 'author', 'thumbnail', 'excerpt', 'custom-fiel
  )
);
```

Grabbing Post Metadata

In this lecture, we used the `useSelect()` function to grab the metadata from the post in our block. This time, we're using the `core/editor` store for retrieving the `getCurrentPostAttribute()` function. This function will allow us to grab metadata for a post like so:

```
const { rating } = useSelect((select) => {
  const { getCurrentPostAttribute } = select("core/editor");

  return {
    rating: getCurrentPostAttribute("meta").recipe_rating,
  };
});
```

In the object, we're grabbing the metadata by passing in `meta` to the function. Keep in mind, other plugins may add metadata. This function will return an object of metadata associated with the post. We're only interested in our metadata, so we accessed the `recipe_rating` property.

Adding a Rating Component

In this lecture, we are going to add a component from an external library called MUI. React is a popular library with a large ecosystem of packages. We can use React components with our blocks. We downloaded this package by using the command from the home page. Refer to the resource section for the link.

After installing the package, we imported it into our **Recipe Summary** block like so:

```
import Rating from "@mui/material/Rating/index.js";
```

Next, we added this component to our block with the following template:

```
<Rating value={rating} readOnly />
```

We're using the `<Rating />` component to render a rating UI. Two properties are being added. The `value` property will set the initial value. The `readOnly` property will prevent users from being clicked on to change the rating. Rating should only be possible on the front end.

Resources

- MUI

Creating a Database Table with Adminer

In this lecture, we decided to create a custom database table for storing the rating since we need to connect the rating with the user and post. We used adminer to help us generate the table. Watch the video for the keypoints from this lecture.

Creating a Database Table during Plugin Activation

In this lecture, we update the plugin activation function to handle creating the table. We borrowed the query from Adminer to help us create this table. We added the following code to the `up_activate_plugin()` function.

```
global $wpdb;
$tableName = "{$wpdb->prefix}recipe_ratings";
$charsetCollate = $wpdb->get_charset_collate();

$sql = "CREATE TABLE {$tableName} (
  ID bigint(20) unsigned NOT NULL AUTO_INCREMENT PRIMARY KEY,
  post_id bigint(20) unsigned NOT NULL,
  user_id bigint(20) unsigned NOT NULL,
  rating decimal(3,2) unsigned NOT NULL
) ENGINE='InnoDB' {$charsetCollate}";

require(ABSPATH . "/wp-admin/includes/upgrade.php");
dbDelta($sql);
```

First, we're grabbing the `$wpdb` object as a global variable. Global variables are variables available to all functions in a script. We can use this variable for various properties and methods related to the database. In this case, we're using it to get the prefix of the database and collation to be consistent with WordPress's database design.

Next, we stored the query in a variable called `$sql`. Inside our query, we're injecting the table name and collation. The backticks have also been removed since they're not needed.

Lastly, we included a file called **upgrade.php**. This file contains a function called `dbDelta()`. This function will execute the query and allow plugins to modify the query if necessary.

Rendering the Block

In this lecture, we talked about rendering the block. There are no lecture notes for this lecture.

Resources

- Recipe Summary Template

Displaying the Cuisine Terms with PHP

In this lecture, we displayed the cuisine terms for a recipe. First, we had to grab the ID of the post. Luckily, by using the block context feature, the ID is exposed as an argument to our function as the third argument like so:

```
function up_recipe_summary_render_cb($atts, $content, $block) {
  $postID = $block->context['postId'];
}
```

We stored the ID in a variable called `$postID`. Afterward, we grabbed the post's terms with the `get_the_terms()` function. This function has two arguments, which are the ID of the post and the name of the taxonomy.

```
$postTerms = get_the_terms($postID, 'cuisine');
```

This function returns an array of terms. We looped through the array to generate a string that is stored in a variable called `$cuisines`.

```
$cuisines = '';
$lastKey = array_key_last($post_terms);

foreach ($postTerms as $key => $term) {
  $url = get_term_meta($term->term_id, 'more_info_url', true);
  $comma = $lastKey === $key ? "" : ", ";
  $cuisines .= "
    <a href='{$url}' target='_blank'>{$term->name}</a>{$comma}
  ";
}
```

In our loop, we're using grabbing the term meta for the URL of the term. In addition, we're checking if the current iteration is the last item in the array.

First, we grabbed the last key by using the `array_key_last()` function. Next, we checked if the `$lastKey` variable matches the `$key` variable from the loop. If it does, we can assume it's the last item. Lastly, we outputted the comma after the link.

The last step was to render the `$cuisines` variable in our template like so:

```
<div class="recipe-data recipe-cuisine">
  <?php echo $cuisines; ?>
</div>
```

Preparing the Frontend

In this lecture, we prepared the frontend by passing on data to our block from the backend. There are different approaches we can use to accomplish this. In this case, we used data attributes. First, we grabbed the post's rating with the `get_post_meta()` function.

```
$rating = get_post_meta($postID, 'recipe_rating', true);
```

We updated our template by adding a `<div>` element with a class called `recipe-data`.

```
<div class="recipe-data" id="recipe-rating"
  data-post-id="<?php echo $postID; ?>"
  data-avg-rating="<?php echo $rating; ?>"
  data-logged-in="<?php echo is_user_logged_in(); ?>">
</div>
```

In the above example, we're adding data attributes for the post ID, current rating, and if the user is logged in. After adding this information, we created a script called `frontend.js` with the following code:

```
document.addEventListener("DOMContentLoaded", () => {
  const block = document.querySelector("#recipe-rating");
  const postID = parseInt(block.dataset.postId);
  const avgRating = parseFloat(block.dataset.avgRating);
  const loggedIn = !!block.dataset.loggedIn;

  console.log(postID, avgRating, loggedIn);
});
```

In the above example, we're selecting the element with the data attributes with its ID. Next, we're extracting each data attribute's value into a variable

through an object called `dataset`. This object is available for all data attributes. A value can be referenced without including the `data` attribute.

In addition, we're typecasting the variables with the `parseInt()`, `parseFloat()` and `!!` operator. The `parseInt()` function will convert a value into a whole number. The `parseFloat()` function will convert a value into a number with decimal values. Lastly, the `!!` operator will convert a value into a boolean value.

Loading React on the Frontend

In this lecture, we began rendering the rating UI with React. First, we had to import a few set of functions.

```
import { render, useState } from "@wordpress/element";
import Rating from "@mui/material/Rating/index.js";
```

We're importing the `render` and `useState` functions from the `@wordpress/element` package. Behind the scenes, this package reexports React's `render` and `useState` functions. They're a copy of these functions to help us with backward compatibility. If React changes its API, we can continue using the old functions while benefitting from the latest version of React. WordPress will handle the differences.

Next, we created a component called `RecipeRatings`.

```
function RecipeRating(props) {
  return <Rating />;
}
```

This component is rendering the `<Rating />` component. Next, we updated our event handler by rendering this function with the `render()` function.

```
render(
  <RecipeRating postID={postID} avgRating={avgRating} loggedIn={logge
  block
);
```

In the above example, we're passing on the data to this component as props. In addition, we're rendering this component in our block. We updated our component to use this data by creating state. One state property for handling the current rating and another for permissions.

```
const [avgRating, setAvgRating] = useState(props.avgRating);
const [permission, setPermission] = useState(props.loggedIn);
```

Lastly, we updated the `<Rating />` component to use these values like so:

```
<Rating
  value={avgRating}
  precision={0.5}
  onChange={async () => {
    if (!permission) {
      return alert(
        "You have already rated this recipe, or you may need to login
      );
    }

    setPermission(false);
  }}
/>
```

In the above example, we added the `value` property to set the initial rating. The `precision` property allows us to configure the precision of the rating by increments. Lastly, the `onChange` event will run when a new rating is added.

The function passed into the `onChange` event is asynchronous since we'll be sending data to the WordPress server. Before doing so, we're checking the user's permission. If they have rated before or are not logged in, they will not be able to rate a recipe. Afterward, we're changing their permission to prevent users from rating the recipe multiple times.

Exercise: Preparing the Rating Endpoint

In this lecture, we created an endpoint for handling new ratings. Most of what we did was standard except for one thing. During the registration of the endpoint, we set the `permission_callback` option to the `is_user_logged_in` function. This function will check if the user is logged in.

```
register_rest_route('up/v1', '/rate', [
  'methods' => WP_REST_Server::CREATABLE,
  'callback' => 'up_rest_api_add_rating_handler',
  'permission_callback' => 'is_user_logged_in'
]);
```

We don't want anyone to rate a recipe. It should be limited to authenticated users. Next, we created the `up_rest_api_add_rating_handler()` function:

```
function up_rest_api_add_rating_handler($request) {
  $response = ['status' => 1];
  $params = $request->get_json_params();

  if (
```

```
    !isset($params['rating'], $params['postID']) ||
    empty($params['rating']) ||
    empty($params['postID'])
  ) {
    return $response;
  }

  $response['status'] = 2;
  return $response;
}
```

Lastly, we updated the `frontend.js` file to handle the response if the rating was a successful:

```
if (response.status === 2) {
  setAvgRating(response.rating);
}
```

If the rating was a success, we'll update the `<Rating />` component to render the new average rating. However, this data isn't available from the response. In a future lecture, we'll ad this to the response.

Prepared SQL Statements

In this lecture, we spent time creating a query to check if the user has already rated the recipe by using the `$wpdb` global variable. First things first, we prepared the data necessary for the query.

```
$rating = round(floatval($params['rating']), 1);
$postID = absint($params['postID']);
$userID = get_current_user_id();
```

We're storing the new rating, ID of the post, and ID of the user. The rating is surrounded with the `floatval()` and `round()` function. This is to make sure the rating will be suitable for inserting into the database. After setting up the data, we wrote a query to select records from the `recipe_ratings` table.

```
global $wpdb;
$wpdb->get_results($wpdb->prepare(
  "SELECT * FROM {$wpdb->prefix}recipe_ratings
  WHERE post_id=%d AND user_id=%d",
  $postID, $userID
));
```

We're using the `$wpdb->get_results()` function to perform a query. Before doing so, we're using the `$wpdb->prepare()` function to make sure that the query is secure. WordPress will strip malicious characters from the values to

prevent an SQL injection. This function accepts the query and the data that will be inserted into the query.

The query can be written with placeholders for the data. Three placeholders are supported:

- `%d` - Number
- `%s` - String
- `%f` - Float

After creating the query, we're checking the `num_row` property to check if the query found any records. If it did, that means the user has rated a recipe. If there's a record, we returned the response.

```
if($wpdb->num_rows > 0) {
  return $response;
}
```

Resources

- wpdb

Inserting the Recipe

In this lecture, we inserted the recipe into the database. To accomplish this task, we used the `$wpdb->insert()` method. This method will generate and prepare a query. It's a convenient method since we do not need to write the query ourselves. It has three arguments, the table name, the values to insert, and the data type for each value.

```
$wpdb->insert(
  "{$wpdb->prefix}recipe_ratings",
  [
    'post_id' => $postID,
    'rating' => $rating,
    'user_id' => $userID
  ],
  ['%d', '%f', '%d']
);
```

Next, we recalculated the average rating for a recipe. This time, we're using the `$wpdb->get_var()` function to retrieve a single value from a query instead of columns. In this example, we're using the `AVG()` SQL function to help us calculate the average of a column value.

After grabbing the average, we updated the post metadata with the `update_post_meta()` function.

```
$avgRating = round($wpdb->get_var($wpdb->prepare(
  "SELECT AVG(`rating`)
  FROM {$wpdb->prefix}recipe_ratings
  WHERE post_id=%d",
  $postID
)), 1);

update_post_meta($postID, 'recipe_rating', $avgRating);
```

Lastly, we updated the response by adding the `$avgRating` variable to the `$response` array.

```
$response['rating'] = $avgRating;
```

Updating the Permissions

In this lecture, we updated the permissions by preventing users from rating a recipe if they've already done so when the block loads. We looked at a different way to count the records from a table by using an SQL function. In the **recipe-summary.php** file, we added the following code:

```
global $wpdb;
$userID = get_current_user_id();
$ratingCount = $wpdb->get_var($wpdb->prepare(
  "SELECT COUNT(*)
  FROM {$wpdb->prefix}recipe_ratings
  WHERE post_id=%d AND user_id=%d",
  $postID, $userID
));
```

We're performing a standard query. However, the selection will return the value from the `COUNT()` function. This function will count the results from the selection. This function accepts the name of the column to count. In this case, we're counting all columns.

After grabbing this info, we updated our template by adding a data attribute with the count.

```
<div class="recipe-data" id="recipe-rating"
  data-post-id="<?php echo $postID; ?>"
  data-avg-rating="<?php echo $rating; ?>"
  data-logged-in="<?php echo is_user_logged_in(); ?>"
  data-rating-count="<?php echo $ratingCount; ?>">
</div>
```

Lastly, we updated the **frontend.js** file by grabbing this information and passing it onto our component.

```
const ratingCount = !!parseInt(block.dataset.ratingCount);

render(<RecipeRating ratingCount={ratingCount} />, block);
```

The component is using the `useEffect()` function to check this information to disable the permissions like so:

```
useEffect(() => {
  if (props.ratingCount) {
    setPermission(false);
  }
}, []);
```

Custom Hooks

In this lecture, we took a moment to make our plugin extendable by adding a custom hook. Adding custom hooks can increase the appeal of your plugin. We can create a custom hook by using the `do_action()` function. The first argument is the name of the hook. The second argument is optional data to provide to hook functions. In our endpoint for rating a recipe, we added the following hook after a recipe has been inserted:

```
do_action('recipe_rated', [
  'postID' => $postID,
  'rating' => $rating,
  'userID' => $userID
]);
```

Next, we tested our hook by creating a custom plugin that sends out an email. In a file called **example.php** that we created in the **wp-content/plugins** directory, we added the following code:

```
/*
 * Plugin Name: Udemy Plus Emailer
 */

add_action('recipe_rated', function($data) {

});
```

We can listen to our custom hook like any other hook. In this example, we're outsourcing the function into an anonymous function to save time. Inside the function, we prepared the data:

```
$post = get_post($data['postID']);
$authorEmail = get_the_author_meta('user_email', $post->post_author);
$subject = 'Your recipe has received a new rating';
$message = "Your recipe {$post->post_title} has received a
  rating of {$data['rating']}.";
```

We're grabbing the post info since we need the ID of the user by using the `get_post()` function. This function accepts the ID of the post, which was provided with the hook. To get the author's email, we used the `get_the_author_email()` function, which accepts the field to grab and the ID of the author.

Lastly, we prepared the subject and message of the email. We can send an email with the `wp_mail()` function, which accepts the email, subject, and message.

```
wp_mail($authorEmail, $subject, $message);
```

Nested Blocks

In this section, we started working on creating nested blocks for the Gutenberg editor.

Creating a Nested Block

In this lecture, we talked about our next block, which is a block for creating team members. I recommend watching the video to review the idea of designing blocks.

Otherwise, we prepared two blocks. The starter files for the blocks can be found in the resource section. After creating the blocks, we updated the **register-blocks.php** file for registering both blocks like so:

```
$blocks = [
  ['name' => 'fancy-header'],
  ['name' => 'search-form', 'options' => [
    'render_callback' => 'up_search_form_render_cb'
  ]],
  ['name' => 'page-header', 'options' => [
    'render_callback' => 'up_page_header_render_cb'
  ]],
  ['name' => 'header-tools', 'options' => [
    'render_callback' => 'up_header_tools_render_cb'
  ]],
  ['name' => 'auth-modal', 'options' => [
    'render_callback' => 'up_auth_modal_render_cb'
  ]],
  ['name' => 'recipe-summary', 'options' => [
    'render_callback' => 'up_recipe_summary_render_cb'
  ]],
  ['name' =>  'team-members-group'],
  ['name' => 'team-member'],
];
```

Resources

- Team Members Block Starter Files
- Single Team Member Block Starter Files

Adding Support for Inner Blocks

In this lecture, we added support for inserting blocks into the **Team Members Group** block. WordPress has a component called `InnerBlocks` for inserting blocks. We can import this component from the `@wordpress/block-editor` package like so:

```
import { InnerBlocks } from "@wordpress/block-editor";
```

Next, we can add this component to our block like so:

```
<InnerBlocks
  orientation="horizontal"
  allowedBlocks={["udemy-plus/team-member"]}
/>
```

In the example, we are adding two properties. The `orientation` property will configure the direction blocks can be stacked. Blocks can be stacked vertically or horizontally. The next property is called `allowedBlocks`, which is an array of blocks that can be inserted into the current block.

In the `save()` function of a block, the inner content can be rendered with the `InnerBlock.Content` component. This component doesn't require configuration since it doesn't render a UI for inserting blocks. It simply adds the blocks inserted into the current block.

```
<InnerBlocks.Content />
```

Resources

- Inner Blocks Component

Inner Blocks Template Options

In this lecture, we configured the `InnerBlocks` component further by adding a template. Most blocks with support for inner blocks will provide an example of the types of blocks that can be added. We can configure the template by adding the `template` property.

```
<InnerBlocks
  orientation="horizontal"
  allowedBlocks={["udemy-plus/team-member"]}
  template={[
    [
      "udemy-plus/team-member",
      {
        name: "John Doe",
        title: "CEO of Udemy",
        bio: "This is an example of a bio.",
      },
    ],
    ["udemy-plus/team-member"],
    ["udemy-plus/team-member"],
  ]}
  // templateLock="all" // Cant add or rearrange blocks
  // templateLock="insert" // Cant add but can rearrange blocks
/>
```

The `template` property is an array of blocks to insert into the component. A block is represented with an array where the first item is the name of the block and the second item is an object of attributes.

In addition to modifying the template, we can lock the template with the `templateLock` property. By default, the existing blocks can be removed, and new ones can be added. Setting this property to `all` will lock the template completely. Whereas the `insert` value will prevent new blocks from being added, but the current set of blocks can be rearranged.

Adding Columns

In this lecture, we updated our block to render the inner blocks in columns. The CSS classes are already ready from the **main.css** file. We can add columns by adding the `cols-` class followed by the number of columns. Currently, our block supports 2 - 4 blocks. We updated the `RangeControl` component to restrict the number of columns by adding the `min` and `max` properties.

```
<RangeControl
  label={__("Columns", "udemy-plus")}
  onChange={(columns) => setAttributes({ columns })}
  value={columns}
  min="2"
  max="4"
/>
```

This component will be responsible for modifying the `columns` attribute for the **Team Members Group** block. After modifying this component, we applied the class to the root element of our block through the `useBlockProps()` function:

```
const blockProps = useBlockProps({
  className: `cols-${columns}`,
});
```

We did the same in the `save()` function:

```
const { columns } = attributes;
const blockProps = useBlockProps.save({
  className: `cols-${columns}`,
});
```

Adding the MediaPlaceholder Component

In this lecture, we began working on the image for the team member. First, we need a UI for uploading images. Luckily, WordPress has a component for completely managing this process. We can import a component called `MediaPlaceholder` from the `@wordpress/block-editor` package.

```
import { MediaPlaceholder } from "@wordpress/block-editor";
```

Inside our block, we inserted this component below the `<img>` tag.

```
<MediaPlaceholder
  allowedTypes={["image"]}
  accept="image/*"
  icon="admin-users"
  onSelect={(image) => {
    console.log(image);
  }}
  onError={(err) => console.error(err)}
/>
```

We added five properties:

- `allowedTypes` - An array of mime types. We can also just provide the category of the mime type. This will limit the files that can be seen in the UI.
- `accept` - The mime type of the file that will be uploaded to WordPress if the user decides to upload a file.
- `icon` - An icon from the Dashicons icon set.
- `onSelect` - An event that will run when an image has been selected or uploaded. The function will provide the image as an argument.
- `onError` - An event that will run when an error occurs, like an upload failure. The error is passed on as an argument.

A mime type is another way to describe a file type. It's common for applications to check the mime type instead of an extension. The format for a mime type is the category followed by the file type. The category and file type are separated with a `/` character.

Resources

- MediaPlaceholder
- MIME Types

Custom Image Sizes

In this lecture, we registered a custom image size for our blocks. You may want to create custom image sizes for pages to load faster. New image sizes should be registered with a hook called `after_setup_theme`. This hook runs after the theme has been initialized.

```
add_action('after_setup_theme', 'up_setup_theme');
```

Inside the **includes** directory, we created a file called **setup.php** with the `up_setup_theme` function

```
function up_setup_theme() {
  add_image_size('teamMember', 56, 56, true);
}
```

In this function, we are running the `add_image_size()` function to register a new image size. This function has four arguments:

- Name - A unique name to identify the size.
- Width - Width of the image.
- Height- Height of the image.
- Crop - Whether to crop the image or not if the image cannot be resized properly.

You can check out the resource section of this lecture for more info on how the cropping mechanism works. Before testing this code, you should regenerate the thumbnails with a plugin called **Regenerate Thumbnails**. The size will only be applied to new images but not existing images. It's recommended to regenerate the thumbnails for existing images to have the new image size.

However, our solution will not work just yet. Let's talk about why in the next lecture.

Resources

- Image Cropping

Adding Filter Hooks

In this lecture, we talked about filter hooks for allowing WordPress to expose the custom image size via the REST API. By registering a new image size, the REST API will not add the image to the response unless we explicitly tell it to. We

can add our size to the response by using a filter hook.

A filter hook is a hook that accepts and returns a value. We can return a new value, the original value, or a modified version of the original value. We can use a filter hook by using the `add_filter()` function like so:

```
add_filter('image_size_names_choose', 'up_custom_image_sizes');
```

We're using a filter hook called `image_size_names_choose` which will provide a list of valid sizes that can be made available in a REST response. Next, inside the **includes** directory, create a file called **image-sizes.php** with the function defined.

```
function up_custom_image_sizes($sizes) {
  return array_merge($sizes, [
    'teamMember' => __('Team Member', 'udemy-plus')
  ]);
}
```

Inside this function, we're given an array of sizes. We're adding onto this array by adding our custom size where the key is the name of the size, and the value is a human-readable name. We're using the `array_merge()` php function to merge the two arrays into one. Lastly, the merged array is returned.

After adding our size, the custom size will appear in the HTTP response for an image. We can use this information to render the image in the block.

Handling Selections

In this lecture, we began handling the images selected by the user. The `onSelect` event gets emitted when a new image is selected in the media library. We updated the function to set the attributes in the block.

```
onSelect={image => {
  setAttributes({
    imgID: image.id,
    imgAlt: image.alt,
    imgURL: image.sizes.teamMember.url
  })
}}
```

We are updating the attributes with the values from the `image` object. Next, we updated the `<img>` tag above the `<MediaPlaceholder />` component:

``jsx { imgURL && {imgAlt} }

```
If an image has been selected, the image will be rendered on the page. We should also toggle the `<MediaPlac

We're gonna set this property to the `imgURL` attribute.

```javascript
disableMediaButtons={imgURL}
```

Even though this attribute is a string, strings are converted into booleans where an empty string is `false`, and a non-empty string is `true`.

Handling URL Images

In this lecture, we added support for image URLs. The `MediaPlaceholder` component supports image URLs by adding an event called `onSelectURL`. The function handler will be provided the URL to the image.

```
onSelectURL={url => {
  setAttributes({
    imgID: null,
    imgAlt: null,
    imgURL: url
  })
}}
```

We're using this argument to update the attributes. We're resetting the other attributes just in case since the image is not stored locally on the server.

Understanding Blob Images
```

In this lecture, we looked at a problem with our current solution. At the moment, we're able to select images from the media library, but uploading does not work. The reason is that the image is still being uploaded to the server, so the image's info is not available.

WordPress leverages a browser feature called a blob. A blob is an object that represents a file. These files are temporary and have URL that we can use to display an image in the browser. A blob is given to our event handler, which doesn't contain the `sizes` or `alt` properties. This causes our function to fail.

We need to update our function to handle blobs and wait for the image upload to be completed. We will handle this task in the next lecture.

Handling Blob Images

In this lecture, we updated our function to handle a blob image. First, we should detect if we have a blob image or not. WordPress has a function called `isBlobURL` exported from a package called `@wordpress/blob`. We imported this function into our block's code.

```
import { isBlobURL } from "@wordpress/blob";
```

Next, we updated the function for the `onSelect` event with the following code:

```
onSelect={image => {
  let newImgURL = null;

  if(isBlobURL(image.url)) {
    newImgURL = image.url
  } else {
    newImgURL = image.sizes ?
      image.sizes.teamMember.url :
      image.media_details.sizes.teamMember.source_url
  }

  setAttributes({
    imgID: image.id,
    imgAlt: image.alt,
    imgURL: newImgURL
  })
}}
```

In the above example, we're checking if the `image.url` property contains a blob URL. If it does, we are storing the URL in a variable. Otherwise, we're grabbing this information from the `images.sizes.teamMember.url` or `images.media_details.sizes.teamMember.source_url` function.

WordPress stores the URL in different locations based on if the image is being uploaded or being selected from the media library. You should always check where the URL is before storing it.

Exercise: Adding a Spinner

In this lecture, we added a spinner while a file was being uploaded. Most of what we did here has been covered before. However, I want to mention the CSS for the spinner. The spinner is created with an SVG element like so:

```
<svg class="components-spinner"></svg>
```

The spinner does not have absolute positioning. To move the spinner above the image, I used absolute positioning in my CSS like so:

```
.wp-block-udemy-plus-team-member .components-spinner {
  position: absolute;
  top: 15px;
  left: 9px;
}
```

Using State to Prevent Blobs

In this lecture, we used state to prevent our block from saving a blob URL. By storing the URL in the state, we don't have to save the image URL in an attribute until the upload is complete. Blob URLs are not reliable. They expire the moment the user exits their browser.

In our file, we imported the `useState` function from the `@wordpress/element` package.

```
import { useState } from "@wordpress/element";
```

Next, we created state for storing the image URL in the `edit` function:

```
const [imgPreview, setImgPreview] = useState(imgURL);
```

Afterward, we updated our template to use our state:

```
{
  imgPreview && <img src={imgPreview} alt={imgAlt} />;
}
{
  isBlobURL(imgPreview) && <Spinner />;
}
<MediaPlaceholder
  onSelect={(image) => {
    let newImgURL = null;

    if (isBlobURL(image.url)) {
      newImgURL = image.url;
    } else {
      newImgURL = image.sizes
        ? image.sizes.teamMember.url
        : image.media_details.sizes.teamMember.source_url;

      setAttributes({
        imgID: image.id,
        imgAlt: image.alt,
        imgURL: newImgURL,
      });
    }

    setImgPreview(newImgURL);
  }}
  disableMediaButtons={imgPreview}
/>;
```

The `setAttributes()` function was moved into the `else` block since we don't want to update the attributes unless we have a valid HTTP URL. At the end of the function, we're updating the state using providing the `newImgURL` variable.

Fixing Blob Memory Leaks

In this lecture, we had to clear the blob from memory since blobs aren't automatically released until the browser has closed. If the user continues to edit a page while a blob is active, the performance of the editor may become sluggish. This is considered a memory leak. A memory leak is when memory hasn't been released for other data.

WordPress provides a function for releasing memory for a blob called `revokeBlobURL` from the `@wordpress/blob` package.

```
import { isBlobURL, revokeBlobURL } from "@wordpress/blob";
```

Next, we called this function at the end of the `else` block in the `onSelect` event. This function has one argument, which is a valid blob URL. If an invalid URL is provided, an error will not be thrown. We are providing this function with the URL from the state.

```
revokeBlobURL(imgPreview);
```

Quickly Refactoring the Team Member Block

In this lecture, we quickly refactored the **Team Member** block for readability. There are no notes for this lecture.

Adding a Toolbar Option

In this lecture, we added an option in the toolbar for replacing the image. Just like other areas of Gutenberg, the toolbar can be extended with custom buttons four our block. Buttons can be added by using the `BlockControls` component. This component can be imported from the `@wordpress/block-editor` package.

```
import { BlockControls } from "@wordpress/block-editor";
```

Next, we imported a component called `MediaReplaceFlow` that will generate a button for opening the media library from the same package.

```
import { MediaReplaceFlow } from "@wordpress/block-editor";
```

After grabbing these components, we added them to our block.

```
<BlockControls>
  <MediaReplaceFlow
    name={__("Replace Image", "udemy-plus")}
    mediaId={imgID}
    mediaURL={imgURL}
    accept="image/*"
    allowedTypes={["image"]}
    onError={(err) => console.error(err)}
    onSelect={selectImage}
    onSelectURL={selectImageURL}
  />
</BlockControls>
```

The following properties have been added:

- `name` - The text to display on the button.
- `mediaId` - The ID of the media item to select in the media library.
- `mediaURL` - The URL of the media item to select in the media library.
- `accept` - The mime type of the file.
- `allowedTypes` - The mime type of the files to display in the media library.
- `onError` - An event that gets emitted on errors.
- `onSelect` - An event that gets emitted when an image has been selected.
- `onSelectURL` - An event that gets emitted when a URL is submitted.

The functions for the events were outsourced to variables since they're the same as the functions for the `MediaPlaceholder` component's events. This component's events were updated to use these events too.

Custom Toolbar Buttons

In this lecture, we added a custom toolbar button for deleting an image. Custom buttons can be added by using a component called `ToolbarButton` from the `@wordpress/components` package.

```
import { ToolbarButton } from "@wordpress/components";
```

Next, we added this component inside the `BlockControls` component with the following properties.

```
<ToolbarButton
  onClick={() => {
    setAttributes({
      imgID: 0,
      imgURL: "",
      imgAlt: "",
    });

    setImgPreview("");
  }}
>
  {__("Remove Image", "udemy-plus")}
</ToolbarButton>
```

There's not much going on here besides resetting the attributes and image preview state to their original values. The `onClick` event is emitted when the button is clicked.

In addition to adding the toolbar, we added the `group` property to the `BlockControls` component to add separators to our buttons. This helps make them stand out from the other buttons on the toolbar.

``jsx ...

```
## Lecture 243: Exercise: Toggling the Image Alt Field

In this lecture, I gave you an exercise to toggle the Image Alt field when an image has been added. If an im

```jsx
{ imgPreview && !isBlobURL(imgPreview) && <TextareaControl /> }
```
```

Custom Block Context

In this lecture, we provided the **Team Members Group** block's attribute value to children blocks by using the block context. WordPress enables us to communicate data between components. In the parent block, we must add the `providesContext` property with a unique ID for the value and the attribute to expose to children blocks like so:

```
{
  "providesContext": {
    "udemy-plus/image-shape": "imageShape"
  }
}
```

It's recommended to give your values unique IDs. The most common format is the name of the plugin followed by the name of the attribute.

Next, we update children blocks to accept this context in the **block.json** file like so:

```
{
  "usesContext": ["udemy-plus/image-shape"]
}
```

Lastly, we could update `edit()` function's argument list to accept the `context` object to grab the data from the parent block like so:

```
export default function({ context })

}
```

We defined a variable for the image's classes. It's recommended to add a class that starts with `wp-image-` followed by the ID. This is for other plugins to select specific media items. Not required, but recommended. Lastly, we added another class for the image shape.

```
const imgClass = `wp-image-${imgID} img-${context["udemy-plus/image-shape"]}`;
```

You can refer to the CSS file for the list of classes for applying shapes to the image. We're using CSS clip paths to generate different shapes. In the resource section of this lecture, you will find a link for generating your own custom paths.

Lastly, we added this class to the `<img>` element.

```
<img src={imgPreview} alt={imgAlt} className={imgClass} />
```

Resources

- Clip Paths

Preparing the Social Media Links

In this lecture, we began preparing the social media links by updating the attribute and looping through the attribute. Firstly, we updated the `socialHandles` attribute by adding a default value:

```
{
  "socialHandles": {
    "type": "array",
    "default": [
      { "url": "https://facebook.com/udemy", "icon": "facebook" },
      { "url": "https://instagram.com/udemy", "icon": "instagram" }
    ]
  }
}
```

In the array, we are adding an object to represent each link. Each object will have a URL and icon. These icons are taken from Bootstrap's icon set. Check out the resource section for a link to this icon set.

In `<div>` tag with the class `social-links`, we looped through the attribute's value with the `map()` method.

```
{
  socialHandles.map((handle, index) => {
    return (
      <a href={handle.url} key={index}>
        <i class={`bi bi-${handle.icon}`}></i>
      </a>
    );
  });
}
```

We're accepting the `index` argument to set the `key` attribute on the `<a>` element. React can have a difficult time keeping track of multiple looped elements. To help it identify specific elements, we can associate a specific item from the array by using the `key` attribute. The value for this attribute should be a unique value from the object. In this case, we're using the index.

Resources

- Bootstrap Icons

Adding a Tooltip

In this lecture, we added a button for adding a social media link for a team member. This button will have a tooltip, which WordPress can help us with. From the `@wordpress/components` package, we imported two components called `Tooltip` and `Icon`.

```
import { Tooltip, Icon } from "@wordpress/components";
```

The `Tooltip` component will handle rendering the tooltip whenever a mouse hovers over an element. The `Icon` component will render an icon from WordPress's Dashicon font set.

Next, we added the button below the other links like so:

```
<Tooltip text={__("Add social media handle", "udemy-plus")}>
  <a
    href="#"
    onClick={(e) => {
      e.preventDefault();

      setAttributes({
        socialHandles: [
          ...socialHandles,
          {
            icon: "question",
            url: "",
          },
        ],
      });
    }}
  >
    <Icon icon="plus" />
  </a>
</Tooltip>
```

The `Tooltip` component must surround the element it should be applied to. We can set the message by configuring the `text` property. Inside this component, we added an `<a>` element with a click event to add a new item to the `socialHandles` array.

Lastly, we're inserting the icon with the `Icon` component. An icon can be configured with the `icon` property. The value for this property must be a valid icon from the Dashicon set.

Detecting Block Selection

In this lecture, we grabbed a property from WordPress for detecting when the block was selected. WordPress provides this information through a component's props called `isSelected`. We updated our function to accept this information.

```
export default function({ isSelected }) {
  ...
}
```

Next, we used this information to toggle the button for adding a new link like so:

```
{
  isSelected && <Tooltip></Tooltip>;
}
```

Tracking the Active Selection

In this lecture, we started keeping track of the current link being edited by the client. We're gonna render a form for editing the social media links. But first, we need to know which link they want to edit. We decided to use state to keep track of this information. Inside our component's function, we defined the state like so:

```
const [activeSocialLink, setActiveSocialLink] = useState(null);
```

By default, we're setting the state to `null` to not display the form initially. Afterward, we updated the template by adding the `onClick` event to change the state when a link is clicked.

```
<a
  href={handle.url}
  key={index}
  onClick={(e) => {
    e.preventDefault();

    setActiveSocialLink(activeSocialLink === index ? null : index);
  }}
  className={activeSocialLink === index && isSelected ? "is-active" : ""}
>
  <i class={`bi bi-${handle.icon}`}></i>
</a>
```

On the link, we're setting the state to the current index. However, if the same link is clicked twice, we'll toggle the state back to `null` to hide the form. In addition, we're applying the `isActive` class to the link if the current link and active link match to highlight the link. This lets the client know what link they're editing.

Next, we updated the link for adding a new social media link by changing the state to the last item in the array like so:

```
<a
  href="#"
  onClick={(e) => {
    e.preventDefault();

    setAttributes({
      socialHandles: [
        ...socialHandles,
        {
          icon: "question",
          url: "",
        },
      ],
    });

    setActiveSocialLink(socialHandles.length);
  }}
>
  <Icon icon="plus" />
</a>
```

Lastly, we created a container for the form that will be conditionally rendered:

```
{
  isSelected && activeSocialLink !== null && (
    <div className="team-member-social-edit-ctr"></div>
  );
}
```

Editing a Social Media Link

In this lecture, we updated our block to edit and delete a link from a team member. First, we imported two components called `TextControl` and `Button`.

```
import { TextControl, Button } from "@wordpress/components";
```

Two `TextControl` components were added for modifying the icon and URL, respectively. In the `onChange` event of either component, we're updating the attribute by storing a copy of the array and current item. WordPress does not recommend directly updating an attribute. Therefore, we should create copies.

After doing so, we can save the copy as the attribute value with the `setAttributes()` function.

```
onChange={url => {
  const tempLink = {...socialHandles[activeSocialLink]}
  const tempSocial = [...socialHandles]

  tempLink.url = url
  tempSocial[activeSocialLink] = tempLink

  setAttributes({ socialHandles: tempSocial })
}}
```

Next, we added the `Button` component. On this component, we added the `isDestructive` property to change the color of the button to red. Inside the `onClick` event, we're removing an item from the array by using the `splice()` function, which accepts the index of the item to remove and the number of items to remove. Lastly, we're resetting the `activeSocialLink` state to `null` since the item will be deleted from the array.

```
<Button
  isDestructive
  onClick={() => {
    const tempSocial = [...socialHandles];
    tempSocial.splice(activeSocialLink, 1);

    setAttributes({
      socialHandles: tempSocial,
    });
    setActiveSocialLink(null);
  }}
>
  {__("Remove", "udemy-plus")}
</Button>
```

Saving the Team Member Block

In this lecture, we began working on the `save()` function for our block. First, we had to grab the image's shape, which was not accessible from the block context. Therefore, we had to save a copy of the context value as an attribute. In the **block.json** file, we added the `imageShape` attribute like so:

```
{
  "imageShape": {
    "type": "string",
    "default": "hexagon"
  }
}
```

Next, we updated the attribute with the context.

```
setAttributes({
  imageShape: context["udemy-plus/image-shape"],
});
```

Lastly, we're able to use this attribute from within the `save()` function:

```
const { imageShape } = attributes;
const imgClass = `wp-image-${imgID} img-${imageShape}`;
```

After grabbing everything we needed, the template was updated to render the image and social media links.

Optimizing Attributes

In this lecture, we began optimizing the attributes of the **Team Member** block by querying the template for the values. Inside the **block.json** file, we updated the `name`, `title`, and `bio` attributes:

```
{
  "name": {
    "type": "string",
    "source": "html",
```

```
    "selector": ".author-meta strong"
  },
  "title": {
    "type": "string",
    "source": "html",
    "selector": ".author-meta span"
  },
  "bio": {
    "type": "string",
    "source": "html",
    "selector": ".member-bio p"
  }
}
```

The `source` property can be set to `html` to select the inner contents of an HTML tag. After adding this property, we can add the `selector` property to select a specific element. This property accepts any valid CSS selector.

Afterward, we updated the image attributes except for the ID since we're not storing it in the template:

```
{
  "imgAlt": {
    "type": "string",
    "default": "",
    "source": "attribute",
    "selector": "img",
    "attribute": "alt"
  },
  "imgURL": {
    "type": "string",
    "source": "attribute",
    "selector": "img",
    "attribute": "src"
  }
}
```

Instead of using `html` as the source, we're using `attribute`, which allows us to grab an attribute's value from an HTML attribute. If the source is an attribute, we must add the `attribute` property to specify an attribute from the element selected by the `selector` property.

Lastly, we updated the `socialHandles` attribute to the following:

```
{
  "socialHandles": {
    "type": "array",
    "default": [
      { "url": "https://facebook.com/udemy", "icon": "facebook" },
      { "url": "https://instagram.com/udemy", "icon": "instagram" }
    ],
    "source": "query",
    "selector": ".social-links a",
    "query": {
      "url": {
        "source": "attribute",
        "attribute": "href"
      },
      "icon": {
        "source": "attribute",
        "attribute": "data-icon"
      }
    }
  }
}
```

We can set the `source` property to `query` to create an array of objects where each property in the object can have a custom query for grabbing data from a template. The queries can be created within the `query` property like we're doing above. Unlike before, we do not need to add the `selector` to each inner query as WordPress will use the parent query.

Block Previews

In this lecture, we are going to add a custom preview for our block. By default, WordPress will render a preview for a block by using the default settings of the block. You may want to give a better preview by adding the `example` property to the block configuration object like so:

```
{
  "example": {
    "attributes": {
      "columns": 2
```

```
    },
    "innerBlocks": [
      {
        "name": "udemy-plus/team-member",
        "attributes": {
          "name": "John Doe",
          "title": "CEO of Udemy",
          "bio": "A short description of John Doe.",
          "imgURL": "https://i.picsum.photos/id/237/100/100.jpg?hmac=Pna_vL4vYBRMXxFMY-lYXcZAL34T7PZWdNDlI
        }
      },
      {
        "name": "udemy-plus/team-member",
        "attributes": {
          "name": "Jane Doe",
          "title": "CEO of Udemy",
          "bio": "A short description of John Doe.",
          "imgURL": "https://i.picsum.photos/id/237/100/100.jpg?hmac=Pna_vL4vYBRMXxFMY-lYXcZAL34T7PZWdNDlI
        }
      }
    ]
  }
}
```

Inside this setting, we can configure the attributes with the `attributes` option and add inner blocks by adding the `innerBlocks` option. In the array of the `innerBlocks` option, you can add an object to represent each block. In the object, we can specify the block with the `name` option and change the attributes through the `attributes` object.

After adding this example, Gutenberg will render a real-time preview of a block after hovering over it. This can be a great way to give users a better idea of what a block will look like after adding it to a page.

Querying Posts

In this section, we started creating blocks that will create custom queries for posts.

Creating a Popular Recipes Block

In this lecture, we began creating a block for popular recipes. There are no notes for this lecture. Check out the resource section for the starter files.

Resources

- Popular Recipes Starter Files

The QueryControls Component

In this lecture, we got started with developing the **Popular Recipes** block by leveraging WordPress's components to help us add a UI for filtering posts. We are going to give users the option of filtering the number of posts and cuisines. We can import a component called `QueryControls` from the `@wordpress/components` package.

```
import { QueryControls } from "@wordpress/components";
```

We can add this component to the sidebar like so:

```
<QueryControls
  numberOfItems={count}
  minItems={1}
  maxItems={10}
  onNumberOfItemsChange={(count) => setAttributes({ count })}
/>
```

In this example, we are adding four properties for adding a slider for modifying the post count. They're the following:

- `numberOfItems` - The initial number of posts.
- `minItems` - The minimum possible value.
- `maxItems` - The maximum possible value.
- `onNumberOfItemsChange` - An event that gets emitted when the count changes.

Keep in mind, this component creates inputs for modifying a query but does not actually create the query itself.

Resources

- QueryControls Block

Category Suggestions

In this lecture, we queried the database for a list of suggestions to display to the user when filtering posts by cuisines. First, we need to import the `useSelect` function to initiate a query.

```
import { useSelect } from "@wordpress/data";
```

Next, we initiated a query with this function. In this function, we grabbed the terms for the cuisine taxonomy by using the `getEntityRecords()` function. Most of this is not new to us except the `per_page` parameter. By setting this parameter to `-1`, we are not going to limit the number of results returned by the query.

```
const terms = useSelect((select) => {
  return select("core").getEntityRecords("taxonomy", "cuisine", {
    per_page: -1,
  });
});
```

After grabbing the terms, we formatted the data. The `QueryControls` component requires that the data be formatted as an object instead of an array, which is what's returned by the `useSelect()` function. We began the conversion with the following code:

```
const suggestions = {};
terms?.forEach((term) => {
  suggestions[term.name] = term;
});
```

In the above example, there are two note-worthy things going on. Firstly, we're using optional chaining. The `terms` variable will initially be empty until the request is complete. If the value is empty, the `forEach()` method will not be available, thus causing an error. We can use the `?` after the variable to instruct the browser not to run the function if the value is empty.

Secondly, we're adding a new property to the `suggestion` object with the square bracket notation. A dynamic property name can be created with this

syntax, where the name can be passed into the square brackets.

On the `QueryControls` component, we applied these suggestions with the `categorySuggestions` property.

```
<QueryControls
  numberOfItems={count}
  minItems={1}
  maxItems={10}
  onNumberOfItemsChange={(count) => setAttributes({ count })}
  categorySuggestions={suggestions}
  onCategoryChange={(newTerms) => {
    console.log(newTerms);
  }}
/>
```

We also have to add the `onCategoryChange` event to handle updates. Otherwise, the input will not appear.

Updating the Attribute

In this lecture, we began updating the `onCategoryChange` event on the `QueryControls` component to handle updates. In addition, we added the `selectedCategories` property for binding the `cuisines` attribute. This will set the initial value of the input.

```
<QueryControls
  onCategoryChange={(newTerms) => {
    const newCuisines = [];

    newTerms.forEach((cuisine) => {
      if (typeof cuisine === "object") {
        return newCuisines.push(cuisine);
      }

      const cuisineTerm = terms?.find((term) => term.name === cuisine

      if (cuisineTerm) newCuisines.push(cuisineTerm);
    });

    setAttributes({ cuisines: newCuisines });
  }}
  selectedCategories={cuisines}
/>
```

In the above example, we began looping through the `newTerms` variable to push the terms into the `newCuisines` array. The `newTerms` variable will contain existing and new terms. We handled both scenarios by checking if the item in the current iteration is an object. If it is, it's an existing term, which we just pushed into the array.

If it's a new term, we searched for the term object from the `terms` variable to grab the whole object. New terms are added as strings instead of whole objects. We need the object if we want to query the database for posts.

In the above example, we're using the `find()` function to search for the term object. This function accepts a function for searching for a value within an array. If `true` is returned, the value is returned. Otherwise, JavaScript will continue looping through the array until a value is found. In this example, we're searching for the value by its name.

Lastly, we pushed the term into the object if a term was found. After the new array was ready, we updated the `cuisines` attribute.

Testing the Endpoint

In this lecture, we tested the endpoint for the recipe post type. By default, WordPress will create a unique endpoint for our post types with the following format: `wp/v2/<post-type-name>`. In our case, the endpoint is the following: `wp/v2/recipe`.

With Postman, we added the following parameters and values:

- `per_page:10` - The number of posts to grab
- `cuisine:<num>` - A list of term IDs to filter the posts by in the **cuisine** taxonomy.
- `_embed` - Includes the author's data and featured image data with each post.

Unfortunately, the endpoint will not order the posts by metadata. We fixed this issue in the next lecture.

Resources

- Posts Endpoint

Ordering Posts by Metadata

In this lecture, we ordered the posts by metadata by using a filter called `rest_{posttype}_query`. The placeholder can be replaced with the name of our post type. In the main plugin file, we added this filter hook with the following code:

```
add_filter('rest_recipe_query', 'up_rest_recipe_query', 10, 2 );
```

The third and fourth arguments will set the priority and number of arguments accepted by the function. Multiple functions can listen to the same hook. The priority can determine the order of the functions. A lower number means a higher priority. By default, the priority is `10`.

The fourth argument is the number of arguments that our function can accept. By default, WordPress assumes that our function will accept one argument, but this hook will send two arguments. Therefore, we must support multiple arguments by specifying the number of arguments that'll be accepted by the function.

After adding the hook, we defined the `up_rest_recipe_query` function in a file called **includes/rest-api/recipe-query-mod.php**.

```
function up_rest_recipe_query($args, $request){
  $orderby = $request->get_param('orderByRating');

  if (isset($orderby)) {
    $args["orderby"] = "meta_value_num";
    $args["meta_key"] = "recipe_rating";
  }

  return $args;
}
```

In the above example, we're grabbing the `orderByRating` parameter. By default, we should not assume that users want to order posts by a metadata key. Instead, we're going to check if this parameter is sent with the request. If it is, we'll change the order of posts by adding the `orderby` and `meta_key` fields.

The `orderby` field will tell WordPress to order the fields by a numeric metadata field. The `meta_key` field will provide the name of the metadata field to order by. These fields will be applied to a class called `WP_Query`, which WordPress uses to query posts from the database. All fields documented can be applied to the `$args` variable.

Lastly, we returned this `$args` variable since this is a filter hook. In our Postman test, we added the following parameters:

- `orderByRating` - Our custom parameter for ordering posts by a metadata key.
- `order` - The direction of the ordering: ascending or descending

Resources

- WP Query

Querying Posts in a Block

In this lecture, we began querying the posts by using the `useSelect()` function. First, we had to update our array to be an array of term IDs since that's what the REST API is expecting. We used this `map()` function to accomplish this:

```
const cuisineIDs = cuisines.map((term) => term.id);
```

Next, we created the query with the `useSelect()` function:

```
const posts = useSelect(
  (select) => {
    return select("core").getEntityRecords("postType", "recipe", {
      per_page: count,
      _embed: true, // allows us to get the featured image
      cuisine: cuisineIDs,
      order: "desc",
      orderByRating: 1,
    });
  },
  [count, cuisines]
);
```

In this function, we're using the `getEntityRecords()` function to initiate the query. We're not limited to querying taxonomy terms. We can grab posts by setting the first argument to `postType` and the second argument to the name of the post type.

Afterward, we began modifying the parameters of the query by limiting the results, embedding the image/author, filtering posts by the cuisine IDs, and, lastly, ordering the results by the metadata.

The `useSelect()` function has a third argument, which will rerun the query if specific variables are updated. We can provide an array of variables to watch. In this example, we're watching the `count` and `cuisines` attributes. If either variable updates, the request is sent again.

Rendering Raw HTML

In this lecture, we began rendering the results of our query for the **Popular Recipes** block. First, we looped through the results by using the `map()` function:

```
{
  posts?.map((post) => {});
```

```
}
```

Next, we prepared the image by storing it in a variable like so:

```
const featuredImage =
  post._embedded &&
  post._embedded["wp:featuredmedia"] &&
  post._embedded["wp:featuredmedia"].length > 0 &&
  post._embedded["wp:featuredmedia"][0];
```

Before grabbing the image, we're checking if the image exists since not all posts are required to have an image. The image can be found within the `post._embedded['wp:featuredmedia']` array.

Lastly, we began rendering the template from within the loop like so:

```
<div class="single-post">
  {featuredImage && (
    <a class="single-post-image" href={post.link}>
      <img
        src={featuredImage.media_details.sizes.thumbnail.source_url}
        alt={featuredImage.alt_text}
      />
    </a>
  )}
  <div class="single-post-detail">
    <a href={post.link}>
      <RawHTML>{post.title.rendered}</RawHTML>
    </a>
    <span>
      by <a href={post.link}>{post._embedded.author[0].name}</a>
    </span>
  </div>
</div>
```

In the above example, we're rendering the image, title, and author. For the title, the title may store HTML. If that's the case, the tags will be escaped. To prevent this behavior, we're using a React component called `RawHTML`, which will allow users to render HTML. This component can be rendered from the `@wordpress/element` package.

```
import { RawHTML } from "@wordpress/element";
```

The WP_Query Class

In this lecture, we began working on rendering the block with our server-side function. Inside the `up_popular_recipes_cb` function, we updated the

argument list to accept the attributes and extracted them into variables like so:

```
function up_popular_recipes_cb($atts) {
  $title = esc_html($atts['title']);
  $cuisineIDs = array_map(function($term) {
    return $term['id'];
  }, $atts['cuisines']);
}
```

For the cuisine IDs, we're looping through the `$atts['cuisine']` attribute with the `array_map()` function since the attributes store objects. We don't need objects. We need an array of numeric IDs. So, we used this function to help us generate an array of IDs. The first argument of this function is the function that will handle looping through each item and grabbing a specific item on each iteration. The second argument is the array itself.

After preparing the data, we began building the query. Unlike before, we must manually create the query, whereas previously, the query was built by WordPress. Custom queries are created with the `WP_Query` class. We prepared the parameters in a variable called `$args`.

```
$args = [
  'post_type' => 'recipe',
  'posts_per_page' => $atts['count'],
  'orderby' => 'meta_value_num',
  'meta_key' => 'recipe_rating',
  'order' => 'DESC'
];
```

In the example above, we're filtering the results by the post type, limiting the results, and lastly, ordering the results by their rating. Up next, we checked if the `$cuisineIDs` variable is empty.

```
if (!empty($cuisineIDs)) {
  $args['tax_query'] = [
    [
      'taxonomy' => 'cuisine',
      'field' => 'term_id',
      'terms' => $cuisineIDs,
    ]
  ];
}
```

If it isn't, we are modifying the query further by filtering the results by the IDs of the taxonomy terms. We can perform this task by adding the `tax_query` parameter. The value for this parameter is an array of arrays where each inner array can have configuration settings for a specific taxonomy.

In this example, we're filtering the posts by the `cuisine` taxonomy. If a post contains a specific taxonomy, the post will be included in the results. The taxonomy terms are added to the query by setting the `field` option to `term_id`, which tells WordPress to use the IDs of the terms and setting the `terms` parameter to a list of term IDs.

Lastly, we initiated a query by creating a new instance of the `WP_Query` class and passing on the arguments.

```
$query = new WP_Query($args);
```

The WordPress Loop

In this lecture, we began looping through the data from query by creating something called **The Loop**. The loop is a concept in WordPress on how posts should be looped through in a template or block. Refer to the resource section for more info.

In our block, we initiated the loop with the following code:

```
if ($query->have_posts()) {
  while ( $query->have_posts() ) {
    $query->the_post();

    ?>

    <?php
  }
}
```

In this example, we're using the `have_posts()` function to check if the query has results. If it does, we began the loop. Once again, we're using the `have_posts()` function. This function performs another job, which is to check if all the posts have been looped through. If they have, the loop will stop running.

Inside the loop, we're running the `the_post()` function to check if we're on the first iteration. If so, WordPress will grab the first post from the query. Otherwise, it'll move on to the next post until there are no more posts that have been looped through. This function should always appear at the beginning of the loop. You may run into strange behavior if you don't.

After running the loop, we used a series of functions to render info on the post.

- `the_permalink()` - The URL to the post.

- `the_post_thumbnail()` - The image of the post. This function has a parameter, which is the size of the image.
- `the_title` - The title of the post.
- `the_author()` - The author of the post.

We don't need to provide information on the post to these functions since they'll check if they're running in the loop. WordPress does most of the heavy lifting for us.

After every post had been looped through, we ran the following function:

```
wp_reset_postdata();
```

WordPress will always query the database for posts. This is considered the main query. Custom queries are considered secondary queries. If we create a secondary query, WordPress will lose focus of the main query. We must run this function after we're finished with our secondary query. Otherwise, other developers may be grabbing post data from query instead of the main query, which can lead to inconsistent behavior.

Resources

- The Loop
- Template Tags

Creating a Daily Recipe Block

In this lecture, we got started with creating a **Daily Recipe** block. There are no notes for this lecture.

Resources

- Daily Recipe Starter Files

Transients

In this lecture, we created a function for temporarily storing a random recipe in the database. WordPress has a feature called transients, which allows us to store in the database with an expiration. In a file called **generate-daily-recipe.php**, we defined a function called `up_generate_daily_recipe()`.

```
function up_generate_daily_recipe() {
  global $wpdb;

  $id = $wpdb->get_var(
```

```
    "SELECT ID FROM {$wpdb->posts}
    WHERE post_status='publish' AND post_type='recipe'
    ORDER BY rand() LIMIT 1"
  );

  set_transient('up_daily_recipe_id', $id, DAY_IN_SECONDS);

  return $id;
}
```

In this function, we're querying the database with the `$wpdb->get_var()` function. The query will select the ID from the posts table. WordPress has a property called `$wpdb->posts` for storing the name of the posts table. Next, we're filtering posts that are published and are the `recipe` post type. Lastly, we're ordering the results with the `ORDER BY` keywords with the `rand()` function, which will randomize the order. The results can be limited with the `LIMIT` keyword.

After querying the database for a recipe, we're using the `set_transient()` function to store a value in the database. This function has three arguments, which are the name of the transient, the value, and the expiration time.

WordPress has a few constants for creating the expiration time, which we used in this example. Check out the resource section of this lecture for more info on transients.

Resources

- Transients

Custom Endpoint

In this lecture, we registered a new route for grabbing a random recipe. In the **init.php** file, we added the following code:

```
register_rest_route('up/v1', '/daily-recipe', [
  'methods' => WP_REST_Server::READABLE,
  'callback' => 'up_rest_api_daily_recipe_handler',
  'permission_callback' => '__return_true'
]);
```

This code snippet will register a route with the following endpoint: `up/v1/daily-recipe`. Next, we created a file called **daily-recipe.php** with the function for handling the request.

```
function up_rest_api_daily_recipe_handler() {
  $response = [
    'url' => '',
```

```
    'img' => '',
    'title' => ''
  ];

  $id = get_transient('up_daily_recipe_id');

  if(!$id){
    $id = up_generate_daily_recipe();
  }

  $response['url'] = get_permalink($id);
  $response['img'] = get_the_post_thumbnail_url($id, 'full');
  $response['title'] = get_the_title($id);

  return $response;
}
```

In the function, we're using the `get_transient()` function to retrieve the transient from the database. This function accepts the name of the transient. If the transient doesn't exist, that means it either expired or a daily recipe hasn't been retrieved yet. If there is no recipe, we generate a new one with the `up_generate_daily_recipe()` function.

Afterward, we updated the response with the URL, image, and title. We're using template tags while passing on the ID of the post since we're not in a loop.

Displaying the Daily Recipe in the Editor

In this lecture, we began displaying the recipe in the Gutenberg editor. There are no notes for this lecture.

Finishing the Daily Recipe Block

In this lecture, we rendered the **Daily Recipe** block with server-side rendering. Nothing new was covered in this lecture besides discussing the inconsistencies with some of WordPress's functions. WordPress has a function called `the_post_post_thumbnail_url()` and `get_the_post_thumbnail_url()`.

Both functions produce a URL to an image but have different parameters. The `the_post_post_thumbnail_url()` does not accept the ID. This means you must resort to the other function for outputting the URL. Not all functions are like this. Some pairs will accept IDs. It varies from function to function. You should always refer to the documentation to catch these gotchas.

Resources

- the_post_thumbnail_url()
- get_the_post_thumbnail_url()

Measuring Queries

In this lecture, I showed you how to measure lines of code by using the `microtime()` function. This function is defined by PHP, which will return the current time in microseconds. We can call this function before and after a line of code to measure how long it took to run.

I showed you an example by measuring the time it took to run a raw SQL query and a query created by the `WP_Query` class. Generating a random recipe is faster with raw SQL than with WordPress's query system. In most cases, the WordPress query system is good enough, but if it's too slow, you may want to consider writing a raw SQL query.

Exploring More Block Features

In this section, we created an entirely new plugin for a single block and submitted it to the WordPress repository.

Creating an Alert Box Block

In this lecture, we began working on a new plugin to follow WordPress's approach of one block per plugin. To save time, a new plugin can be created with the following command:

```
npx @wordpress/create-block alert-box --namespace u-plus
```

We're using a new option called `namespace`, which will allow us to configure the namespace of the plugin. In this example, we're changing the namespace to `u-plus` to match the current namespace of our project.

After creating the plugin, we overrode the files with the files in the resource section of this lecture. Check it out for the complete list of modified files.

Resources

- Alert Box Starter Files

Understanding SASS

In this lecture, we learned about SASS, which is popular among the WordPress community. SASS is a language that extends CSS with features like nesting, functions, mixins, and so much more. Since SASS is not supported in browsers, SASS code must be converted into CSS. Luckily, by using the `@wordpress/create-block` package, this process has been set up for us.

In the project, we are given a file called **style.css** for containing the block's styles. Inside this file, we have the following code:

```
div.wp-block-u-plus-alert-box {
  /* div.wp-block-u-plus-alert-box.is-style-accented */
  &.is-style-accented {
  }

  /* div.wp-block-u-plus-alert-box .dashicon */
  .dashicon {
```

```
    }
}
```

In this example, we're not limited to adding properties. We can use additional selectors that will be converted into valid CSS selectors. Children selectors will have the parent selector prefixed to them. We can reference the parent selector with the `&` character.

This is just one feature, but nested selectors provide clarity and act as a shortcut for writing CSS. There are more features worth checking. Check out the resource section for a link to the official site for SASS.

Resources

- SASS

Editor Styles

In this lecture, we talked about an additional file called **editor.scss**. WordPress provides this file for applying styles to our block for the Gutenberg editor only. During the build process, the filename is changed to **index.css**. Since we're not going to need special styles for the editor, we removed this file, including the `editorStyle` property from the **block.json** file.

Block Supports

In this lecture, we decided to leverage WordPress's features in our blocks. Some features can be enabled with a flick of a switch. For a complete list of features that can be added to our block, check out the resource section of this lecture for a link.

In the **block.json** file, we added the following:

```
{
  "supports": {
    "html": false,
    "align": true
  }
}
```

In this example, we are adding the `supports` option to the block file to begin configuring the features to enable/disable. First, we're disabling the `html` option to prevent users from modifying our block with HTML. Next, we're setting the `align` option to `true` to enable alignment options on our block.

There are 5 alignment options, which are left, center, right, wide, and full. You may not want to support all options. In this case, you can set the `align` option to an array of alignment options like so:

```
{
  "supports": {
    "align": ["full"]
  }
}
```

WordPress will apply classes to the parent element of the block with the name of the alignment option. Here's an example of our block using these styles:

```
.alignfull {
  border-radius: 0;
}
```

Resources

- Supports

Block Variations

In this lecture, we added a block variation to the **Alert Box** block. A block variation is a block that has different default attribute values, thus causing the appearance to look different. Variations will inherit the `edit` and `save` functions of the original block. You only need to change the name, icon, and attributes.

Inside the **index.js** file of our block, we added the following:

```
{
  variations: [
    {
      name: "u-plus/alert-box-accented",
      title: "Alert Box Accented",
      icon: Icons.accented,
      attributes: {
        textColor: "#111827",
        className: "is-style-accented",
      },
    },
  ];
}
```

In this example, we're changing the properties of the original block. In the `attributes` object, we can modify the class name by adding the `className` property. Technically not an attribute, WordPress treats it as so.

In addition, we imported some icons. You can find these icons in the resource section of this lecture. In the same file, we imported it with the following code:

```
import Icons from "./icons";
```

Make sure to update the **block.json** file to remove the icon since we're setting the icon to an SVG image.

Resources

- Icons

Block Styles

In this lecture, we explored a feature called block styles for switching between different styles of a block without manually adding a class to the block or forcing the user to re-add a block to the page. We can automatically apply different classes to a block by adding the `styles` option to the block configuration like so:

```
{
  "styles": [
    {
      "name": "regular",
      "label": "Regular",
      "isDefault": true
    },
    {
      "name": "accented",
      "label": "Accented"
    }
  ]
}
```

In this option, we can add an array of styles that are represented as object. In each object, we can add a name and label. The `name` property will be the name of the class. All classes are prefixed with `is-style-`. The `label` property will appear on the editor. In addition, we can add a property called `isDefault` to set the default style of a block.

Resources

- Block Styles

Understanding Deprecations

In this lecture, we talked about deprecations. In some cases, you may need to update a block after creating it. Before WordPress renders a block, it'll perform validation on a block to help developers catch upgrade issues. Validation is performed with the following steps:

1. Extract attributes from the block saved in the database.
2. Pass on these attributes to the current `save()` function.
3. Compare the output of the `save()` function with the template saved in the database.

If the templates do not match, WordPress will fail the validation. If that's the case, we can help WordPress upgrade a block, which is what we do in the next lecture.

Handling Template Deprecations

In this lecture, we began working on upgrading a block from an older version to a newer version. There are a couple of steps you should take for this process. Firstly, you should create a new file for storing the older configuration of a block. In this case, we created a file called **v1.js**. This file was imported into the **index.js** file and added to the configuration through an array called `deprecations`.

```
registerBlockType("u-plus/alert-box", {
  deprecated: [v1],
});
```

Inside this file, we exported an object with the following attributes: `supports`, `attributes`, `save`. We do not need to provide every attribute. The values in each of these properties must contain the original values from the old block.

Resources

- Deprecation

Publishing a Plugin

In this lecture, I walked through the process of submitting a publish to the official WordPress plugin repository. I recommend checking out the resource section for useful links. There are no lecture notes.

Resources

- Plugin Readme
- Plugin Submission

Custom Admin Pages

In this section, we created custom admin pages for editing our custom plugin settings.

WordPress APIs

In this lecture, we briefly talked about the various WordPress APIs. Check out the resource section of this lecture for more info. They're the following:

- **Dashboard Widgets API** - An API for adding widgets to the WordPress admin dashboard.
- **Database API** - An API for interacting with the database of WordPress
- **File Header API** - An API for processing files with file headers.
- **File System API** - An API for securely connecting with a server for transferring files.
- **HTTP API** - An API for sending HTTP requests with PHP.
- **Metadata API** - An API for adding metadata to posts, users, comments, and terms.
- **Options API** - An API for adding settings or data to the database.
- **Plugin API** - An API for working with hooks in WordPress.
- **Quick tags API** - An API for adding buttons to the classic editor.
- **Rewrite API** - An API for modifying the URL structure of a WordPress site.
- **Settings API** - An API for generating forms for modifying options.
- **Shortcodes API** - An API for adding custom shortcodes to a site.
- **Theme Modification API** - An API for adding theme options.
- **Theme Customization API** - An API for adding fields and panels to the WordPress customizer.
- **Transients API** - An API for storing data temporarily in the database.
- **Widgets API** - An API for adding widgets to the sidebar of a theme.
- **XML-RPC WordPress API** - An API for exposing our site to mobile and desktop apps.

Resources

- Core APIs

Understanding the Options API

In this lecture, we talked about the options API. This API standardized a way for plugins to add settings to a WordPress site. Rather than creating a custom table for settings, we can leverage the **wp_options** table for storing plugin

settings. WordPress offers a few functions for adding, removing, and modifying data from this table.

Resources

- Options API

Adding Plugin Settings

In this lecture, we began adding our plugin settings during the activation of the plugin. The first step was to check if the settings existed. It's possible that users may have activated our plugin earlier. We do not want to override our original settings. In the activation file, we added the following:

```
$options = get_option('up_options');

if(!$options) {

}
```

We're retrieving an option from the database by using the `get_option()` function. This function has one argument, which is the name of the option. If an option doesn't exist, a `false` value is returned. We are using a conditional statement to check the existence of this value. If it doesn't, we can proceed to add our options to the database.

```
add_option('up_options', [
  'og_title' => get_bloginfo('name'),
  'og_image' => '',
  'og_description' => get_bloginfo('description'),
  'enable_og' => 1
]);
```

We can insert new data by using the `add_option()` function. This function has two arguments, which are the name of the option and the value. In this example, we are going to create options for modifying the open graph settings of a site.

Open graph is a protocol for helping social media platforms generate a preview of site. We are going to allow users to modify the title, image, and description. In addition, we'll allow users to disable this option if they have another plugin for adding open graph tags.

Resources

- Open Graph

- Open Graph Preview

Adding a Custom Admin Menu & Page

In this lecture, we added a custom admin page to the admin dashboard. WordPress has a hook for registering a new menu item called `admin_menu`.

```
add_action('admin_menu', 'up_admin_menus');
```

In the **includes/admin** folder, we created a file called **menus.php** with the function for handling this hook.

```
function up_admin_menus() {
  add_menu_page(
    __('Udemy Plus', 'udemy-plus'),
    __('Udemy Plus', 'udemy-plus'),
    'edit_theme_options',
    'up_plugin_options',
    'up_plugin_options_page',
    plugins_url( 'letter-u.svg', UP_PLUGIN_FILE )
  );
}
```

In this function, we're running a function called `add_menu_page()` to add a new menu. It has six arguments.

1. The text to display in the title of the page.
2. The text to display in the menu link.
3. The capability for viewing this page.
4. A menu slug/ID for identifying the page.
5. A function for rendering the content of the page.
6. An icon

Capabilities can be thought of as the permissions for a certain action. You can refer to the resource section for a complete list of capabilities. As for the icon, we're using the `plugins_url()` function for generating a HTTP to the icon. This function has two arguments, which are the name of the file and the plugin file.

We defined the constant in the second argument in the main plugin file as so:

```
define('UP_PLUGIN_FILE', __FILE__);
```

Lastly, we defined the function for rendering the content:

```
function up_plugin_options_page() {
  ?>
  <div class="wrap">
    Test
  </div>
  <?php
}
```

We're using a class called `wrap`, which will position the content perfectly on the page next to the menu. You should always wrap your content with an element that has this class.

Resources

- add_menu_page() Function
- Roles and Capabilities
- SVG Icon

Preparing the Admin Page

In this lecture, we began preparing the admin page by adding a form. In the resource section of this lecture, I provided a link to classes for helping you design the form. Ahead of time, I've prepared the template for our form. In the template, I'm using the `esc_html_e()` function for translating messages while escaping the output.

After adding the template, we updated the `<form>` element with a few attributes:

```
<form novalidate="novalidate" method="POST" action="admin-post.php"><
```

On this element, we added the `method` attribute to set the HTTP method. The `action` attribute will tell the browser where to send the form data. In this case, we're sending the data to a file called **admin-post.php**. This file can be found in the **wp-admin** directory. It's the recommended file for sending form submissions.

Up next, we added a hidden input:

```
<input type="hidden" name="action" value="up_save_options" />
```

Multiple forms can be sent to the same URL. To help WordPress identify our form, we can set the `action` to a unique value. Without this input, we will not be able to intercept the request and handle the form submission.

Afterward, we invoked a function called `wp_nonce_field()`.

```
wp_nonce_field('up_options_verify');
```

Nonce stands for **number used once**. They're a feature for generating a unique value in the form that can be verified on the backend. They're very difficult to spoof, so it prevents hackers from submitting our form without permission.

The last step we took was updating the inputs with their respective values from our custom option. For the final input, we had a checkbox where we had to toggle the `checked` attribute. WordPress has a special function for adding this attribute called `checked()`. We added it to the input like so:

```
<input name="up_enable_og" type="checkbox" id="up_enable_og"
  value="1" <?php checked('1', $options['enable_og']); ?> />
```

This function accepts the two values to compare. If they match, the `checked` attribute is outputted onto the input.

Resources

- WordPress Admin Styles
- Admin Form
- Nonces

Handling Admin Form Submissions

In this lecture, we intercepted the form submission by using a hook called `admin_post_{action}`. This hook runs when a form has been submitted to the **admin-post.php** file. The function will run if the form has the specified action.

```
add_action('admin_post_up_save_options', 'up_save_options');
```

In addition, we logged the form data by using the following code:

```
print_r($_POST);
```

The `$_POST` variable will contain the form data submitted by a form. PHP handles this process for you. It'll be an array of values, which we can output by using the `print_r()` function since the `echo` keyword cannot output arrays or objects.

Resources

- Admin*post* Hook

Updating Options

In this lecture, we began updating the options with the values submitted by the user in the form. Before doing so, we decided to validate the request to prevent users with insufficient permissions from submitting the form.

```
if(!current_user_can('edit_theme_options')){
  wp_die( __( 'You are not allowed to be on this page.', 'udemy-plus'
}
```

In the above example, we're using the `current_user_can()` function to check a user's capabilities. We can pass in a capability that the current user should have. If the function returns `false`, we will kill the script with the `wp_die()` function. This function kills the script while outputting a message to the screen.

Afterward, we checked the nonce to verify that it was sent with the `check_admin_referer()` function.

```
check_admin_referer('up_options_verify');
```

This function accepts the nonce created by our form. If the nonce is invalid, the script will be killed.

After updating the options, we redirected the user:

```
wp_redirect( admin_url('admin.php?page=up_plugin_options&status=1') )
```

The `wp_redirect()` function handles redirecting the user. This function accepts a valid URL. In this example, we're using the `admin_url()` function to help us generate a URL relative to admin dashboard. This function accepts a path. In this instance, we're redirecting them to our custom admin page. This page exists under the **admin.php** file, where the `page` query parameter should hold the slug of our page.

Lastly, we updated our page to output a message if the form was submitted successfully by checking if the `status` query parameter exists in the URL. By default, PHP will store query parameters as an array in the `$_GET` variable.

```
if(isset($_GET['status']) && $_GET['status'] == 1){

}
```

Enqueuing the Media Library

In this lecture, we enqueued the media library. The media library is not limited to the Gutenberg editor. We can add it to custom admin pages. Before enqueueing the media library's asset files, we used a hook called `admin_enqueue_scripts`, which will run when the user visits the admin dashboard.

```
add_action('admin_enqueue_scripts', 'up_admin_enqueue');
```

Lastly, we updated our function to accept an argument called `$hook_suffix`. This argument contains the name of the current page.

```
function up_admin_enqueue($hook_suffix) {
  if ($hook_suffix === "toplevel_page_up_plugin_options") {
    wp_enqueue_media();
  }
}
```

We used this argument to check if the user is viewing our custom admin page. If they are, we'll enqueue the files with the `wp_enqueue_media()` function. This function will handle enqueueing all the files required for the media library.

Adding New Entry Files to Webpack

In this lecture, we created a new script file that's not related to blocks. Since that's the case, it will not be processed to Webpack. However, we may want to run it through Webpack for optimization. In this case, we need to update the Webpack configuration.

In the root directory of our plugin, we created a file called **webpack.config.js**. If this file exists, Webpack will prioritize our configuration over WordPress's configuration file. However, we're not interested in overriding the configuration by WordPress. We're simply interested in extending it.

We can extend the configuration by importing the configuration object by WordPress and spreading it into our object.

```
import defaultConfig from "@wordpress/scripts/config/webpack.config.j

export default {
  ...defaultConfig,
  entry: {
    ...defaultConfig.entry(),
    "admin/index": "./src/admin",
  },
};
```

In the above configuration, we're running the `defaultCOnfig.entry()` function to add WordPress's entry point to our config object. Entry points refer to the files that Webpack should process. After adding WordPress's entry point, we added our file to the list of files to process.

The property name represents where to store the file in the build. The extension can be excluded since Webpack will add it for us. The property value is the path to the file.

Registering the Admin Assets

In this lecture, we began registering the admin assets. Most of this process was familiar to us. We used a new function called `wp_register_script()` to register a script. This function has five arguments.

1. Handle name
2. URL to the file
3. Dependencies
4. Version of the file
5. Whether to load the file in the header or footer

For the third and fourth arguments, we can use the file generated by Webpack. In the build of our files, Webpack will create a PHP file. This file is an array of dependencies in our script and the version of the file. This array is returned by the file.

We can include this file like so:

```
$adminAsset = include(UP_PLUGIN_DIR . 'build/admin/index.asset.php');
```

Returning values are limited to functions. We can return values from files that will be returned by the `include()` function.

Lastly, we called the `wp_register_script()` function like so:

```
wp_register_script(
  'up_admin',
```

```
    plugins_url('/build/admin/index.js', UP_PLUGIN_FILE),
    $adminAsset['dependencies'],
    $adminAsset['version'],
    true
);
```

Resources

- wp_register_script()

Exercise: Adding an Open Graph Image Size

In this lecture, we added a custom image size as an exercise. There are no notes for this lecture.

Selecting Images

In this lecture, we began initializing and opening the media uploader. With the media files enqueued, we are given access to a function called `wp.media()` for creating a new instance of the media library.

```
let mediaFrame = wp.media({
  title: "Select or Upload Media",
  button: {
    text: "Use this media",
  },
  multiple: false,
});
```

This function accepts an object of configuration settings. We can configure the title of the post, the text in the button, and whether multiple images can be selected. The instance is returned by the function. We can interact with the media library through the `mediaFrame` variable.

First, we opened the library whenever the user clicked on the button.

```
ogImgBtn.addEventListener("click", (e) => {
  e.preventDefault();

  mediaFrame.open();
});
```

Next, we added a custom event listener called `select`. This event is emitted when image is selected.

```
mediaFrame.on("select", () => {
  const attachment = mediaFrame.state().get("selection").first().toJS
  ogImgCtr.src = attachment.sizes.opengraph.url;
  ogImgInput.value = attachment.sizes.opengraph.url;
});
```

In the event handler, we called the `state()` function to grab the data from the media library. Next, we chained the `get()` function to select a specific piece of data from the collection of data called `selection`. This will return an array of images selected by the user. Afterward, we chained the `first()` function to get the first item in the array. Lastly, we called the `toJSON()` function to convert the data into an object.

After grabbing this information, we updated our elements with the URL from the image.

Resources

- wp.media() Method

Adding a Submenu

In this lecture, we added a submenu to our custom menu. We shouldn't always add top-level menus to the dashboard. We can add a submenu to any existing or custom menu on the dashboard by using the `add_submenu_page()` function.

```
add_submenu_page(
  'up_plugin_options',
  __('Alt Udemy Plus', 'udemy-plus'),
  __('Alt Udemy Plus', 'udemy-plus'),
  'edit_theme_options',
  'up_plugin_options_alt',
  'up_plugin_options_alt_page'
);
```

This function has five arguments:

1. The name of the parent menu slug.
2. The text to appear in the title of the page.
3. The text to appear in the menu of the page.
4. The capability for viewing this page.
5. The menu slug.
6. The function to run when viewing the page.

Resources

- Settings API

Registering an Options Group

In this lecture, we registered our option with a group. The settings API is capable of updating multiple options. To render multiple options on a page, we must register our options with a group. We can do so by using the `admin_init` hook.

```
add_action('admin_init', 'up_settings_api');
```

This hook will run during the initialization of the admin dashboard. From this hook, we can register our option with a group by using the `register_setting()` function. This function has two arguments, the name of our group and the option to associate with the group.

```
register_setting('up_options_group', 'up_options');
```

Lastly, we prepared the form like so:

```
<form method="post" action="options.php"></form>
```

The `action` attribute must be set to the **options.php** file. Unlike custom forms, forms created with the settings API must send their data to this file. Lastly, we added our options group to the form with the `settings_fields()` function.

```
settings_fields('up_opts_group');
```

This function has one argument, which is the name of the options group that the form will update. Any options associated with this group will be updated.

Adding Sections

In this lecture, we added a section to our options page by using a function called `add_settings_section`. Sections are a great way to organize fields within page. At least one section must be present on the page before you can add fields.

```
add_settings_section(
  'up_options_section',
  esc_html__('Udemy Plus Settings', 'udemy-plus'),
```

```
  'up_options_section_cb',
  'up_options_page'
);
```

This function has four arguments:

1. The ID of the section.
2. The title of the section.
3. The function for rendering additional content.
4. The name of the page.

Afterward, we rendered this section of the page by using the `do_settings_sections()` function. This function accepts the name of the section.

```
do_settings_sections('up_options_page');
```

Adding Fields

In this lecture, we began adding fields to the section we added to the page. Fields must be assigned to a section. We can create a field by using the `add_setting_field()` function.

```
add_settings_field(
  'og_title_input',
  esc_html__('Open Graph Title', 'udemy-plus'),
  'og_title_input_cb',
  'up_options_page',
  'up_options_section'
);
```

This function has five arguments:

1. The ID of the field.
2. The text for the label of the field.
3. The function for rendering the input.
4. The name of the page that this field should appear on.
5. The name of the section that this field should appear on.

Next, we defined the function for rendering the input:

```
function og_title_input_cb() {
  $options = get_option('up_options');
  ?>
  <input class="regular-text" name="up_options[og_title]"
    value="<?php echo esc_attr($options['og_title']); ?>" />
```

```
  <?php
}
```

As usual, we're grabbing the `up_options` option from the database. We're using this option to set the value of the input. The most important thing to note down about this input is the `name` attribute. This attribute's value must be the name of the option that will be updated in the database. If you're storing an array, you can add square brackets to update a specific item from the array.

Resources

- Settings API Input Fields

Registering Post Metadata

In this lecture, we registered more post metadata with the `register_post_meta()` function. Unlike before, we're going to apply the metadata to all post types. We can add metadata for all post types by passing an empty string into the first argument.

```
register_post_meta('', 'og_title', [
  'single' => true,
  'type' => 'string',
  'show_in_rest' => true,
  'sanitize_callback' => 'sanitize_text_field',
  'auth_callback' => function() {
    return current_user_can('edit_posts');
  }
]);
```

Most of the options are familiar to us. I'm introducing two new options called `sanitize_callback` and `auth_callback`. The `sanitize_callback` option allows us to sanitize the value during updates to the metadata through the REST API. In this example, we're running the value through the `sanitize_text_field()` function. As for the `auth_callback` option, this option allows us to limit who can update, add, or delete this metadata. In this example, we're limiting permission to users who have the `edit_posts` capability.

In another metadata registration, we also looked at the `default` option for setting a default value:

```
register_post_meta('', 'og_override_image', [
  'default' => false
]);
```

Enqueueing Block Editor Assets

In this lecture, we began preparing the script and enqueueing the files for the Gutenberg editor. Files can be added to the editor with the `enqueue_block_editor_assets` hook.

```
add_action('enqueue_block_editor_assets', 'up_enqueue_block_editor_as
```

In our function, we checked the current page since the Gutenberg editor will render for the post editing page or the full-site editor. Our script should not be loaded for the full-site editor since we're trying to modify metadata for a specific post.

```
$current_screen = get_current_screen();
if ($current_screen->base == 'appearance_page_gutenberg-edit-site') {
  return;
}
```

We can use the `get_current_screen()` function to grab information related to the current page. This function returns an object with a property called `base` with the name of the page.

Registering a Sidebar

In this lecture, we registered a sidebar for modifying the metadata of a post. Custom sidebars can be created by importing the `PluginSidebar` component from the `@wordpress/edit-post` package. However, we shouldn't add this component prematurely, we should wait until the editor is ready. WordPress has a special function for running code in the editor called `registerPlugin()` from the `@wordpress/plugins` package.

```
import { registerPlugin } from "@wordpress/plugins";
import { PluginSidebar } from "@wordpress/edit-post";
```

We called the `registePlugin()` function to begin rendering a sidebar. This function has two arguments, the name of the plugin and an object with a `render()` function for rendering content.

```
registerPlugin("up-sidebar", {
  render() {
    return (
      <PluginSidebar
        name="up-sidebar"
        icon="share"
        title={__("Udemy Plus Sidebar", "udemy-plus")}
```

```
      >
        Test
      </PluginSidebar>
    );
  },
});
```

Within the function, we're returning the `PluginSidebar` component to render the sidebar. This component has three properties for the name, icon, and title. This component will handle adding a button to the editor and toggling the sidebar.

Selecting Metadata

In this lecture, we began selecting the metadata and rendering fields for modifying the data. There are no notes for this lecture.

Resources

- Sidebar Fields

Dispatching Updates

In this lecture, we began performing updates to our metadata during modifications to the fields. WordPress has a set of functions dedicated to updating data, which can be retrieved with the `useDispatch()` function:

```
import { useDispatch } from "@wordpress/data";
```

This function works similarly to the `useSelect()` function. First, we must select a storage for accessing methods to interact with the data from that store. In our case, functions for modifying metadata can be accessed from the `core/editor` storage.

```
const { editPost } = useDispatch("core/editor");
```

From this store, we're grabbing the `editPost` function for modifying the metadata. You can modify the metadata like so:

```
editPost({
  meta: {
    meta: { og_title },
  },
});
```

Resources

- Data Reference

Custom Media Upload Interface

In this lecture, we explored a different option for opening the media library. WordPress has two components for allowing developers to create a custom interface for opening the media libary called `MediaUpload` and `MediaUploadCheck`. We imported them from the `@wordpress/block-editor` package.

```
import { MediaUpload, MediaUploadCheck } from "@wordpress/block-edito
```

The `MediaUploadCheck` component will verify that the user has the proper capabilities for interacting with the media library. We did not have to include this component before since the `MediaPlaceholder` component performs that step for us. We added this component to the sidebar along with the `MediaUpload` component.

```
<MediaUploadCheck>
  <MediaUpload
    allowedTypes={["image"]}
    render={({ open }) => {
      return (
        <Button isPrimary onClick={open}>
          {__("Change Image", "udemy-plus")}
        </Button>
      );
    }}
    onSelect={(image) => {
      editPost({
        meta: {
          og_image: image.sizes.opengraph.url,
        },
      });
    }}
  />
</MediaUploadCheck>
```

On this component, we have three properties.

- `allowedTypes` - An array of valid file types.
- `render` - A function for rendering the UI for opening the media library. The function is provided a function called `open`, which can be called for opening the media library.
- `onSelect` - An event that gets emitted when an image is selected.

Rendering Tags in the Head

In this lecture, we began rendering the tags for generating a preview with the Open Graph Protocol. Here's what we learned. First, we learned how to generate a URL to our site with the `site_url()` function. This function accepts a path relative to the base URL.

```
$url = site_url('/');
```

Next, before updating the open graph data with information for a single post, we had to check if we were on a single post with the `is_single()` function.

```
if(is_single()) {

}
```

If we're viewing a single post, the URL to a post can be grabbed with the `get_permalink()` function with the ID of the post passed in.

```
$url = get_permalink($postID);
```

Lastly, the open graph preview can be configured with `<meta>` tags. These tags must have two attributes, the `property` attribute for configuring the type of value and the `content` attribute for setting the value.

```
<meta property="og:title"
  content="<?php echo esc_attr($title); ?>" />
<meta property="og:description"
  content="<?php echo esc_attr($description); ?>" />
<meta property="og:image"
  content="<?php echo esc_attr($image); ?>" />
<meta property="og:type" content="website" />
<meta property="og:url" content="<?php echo esc_attr($url); ?>" />
```

The following open graph values were configured:

- `og:title` - The title of the preview.
- `og:description` - The description for the preview.
- `og:image` - A URL to the image.
- `og:type` - The type of content that is on the page.
- `og:url` - The URL to the site.

Finishing Touches

In this section, we added some finishing touches by exploring topics that don't deserve a section of their own but are still worth talking about.

Preparing a New Format Type

In this lecture, we began creating a format type, which is a feature for applying an effect to a selection of text. There are no notes for this lecture.

Resources

- Neon CSS

Registering a Format Type

In this lecture, we registered a plugin for registering a new format type. First, we need to import `registerFormatType` and `RichTextToolbarButton` from their respective packages.

```
import { registerFormatType, toggleFormat } from "@wordpress/rich-tex
import { RichTextToolbarButton } from "@wordpress/block-editor";
```

Next, we called the `registerFormatType()` function, which will allow us to add a format to the toolbar.

```
registerFormatType("udemy-plus/neon", {
  title: __("Neon", "udemy-plus"),
  tagName: "span",
  className: "neon",
  edit() {},
});
```

This function has two arguments. The first argument is the name of the format. The second argument is a configuration object. In this object, we can define the following properties:

- `title` - The readable name of the format.
- `tagName` - The tag that'll surround the text selection.
- `className` - The name of the class that'll be applied to the tag.
- `edit` - The function that'll render the button.

Within the function, we returned the following:

```
return (
  <RichTextToolbarButton
    title={__("Neon", "udemy-plus")}
    icon="superhero"
    isActive={isActive}
    onClick={() => {
      onChange(
        toggleFormat(value, {
          type: "udemy-plus/neon",
        })
      );
    }}
  />
);
```

We're using the `RichTextToolbarButton` component to help us render the button. WordPress will handle rendering this button in the correct location. This component has four properties.

- `title` - The name of the button.
- `icon` - A dashicon that'll be displayed next to the name.
- `isActive` - Whether the format should be active based on the current selection and format.
- `onClick` - An event that'll be emitted when the user clicks on the button.

For the `onClick` event, we're using a function called `onChange` to verify that the format should be updated. If that's the case, we passed in the `toggleFormat()` to handle toggling the selection. This function has two arguments, which are the selection and the type of format.

Some of this information is available as props to the `edit()` function, which you can destructure like so:

```
edit({ isActive, value, onChange }) {

}
```

Conditionally Rendering Toolbar Buttons

In this lecture, we conditionally rendered our button for specific blocks by using state. We grabbed the currently selected block by calling the `getSelectedBlock()` function from the `core/block-editor` store.

```
const selectedBlock = useSelect((select) =>
  select("core/block-editor").getSelectedBlock()
);
```

Next, we applied it like so:

```
{
  selectedBlock?.name === 'core/paragraph' &&
}
```

Translation Functions

In this lecture, we explored the various translation functions available within WordPress. WordPress offers dozens of functions for translating text. They're the following:

- `__()` - Returns the translated text.
- `_e()` - Echoed the translated text
- `_n()` - Returns the singular/plural form of text.
- `_x()` - Returns the translation with an additional note for providing context.
- `_ex()` - Echos the translation with support for context. A combination of the `_e()` and `_x()` functions.
- `_nx()` - A combination of the `_n()` and `_x()` functions.

In addition to these functions, WordPress has additional functions for escaping translation messages.

```
esc_attr__();
esc_attr_e();
esc_attr_x();
esc_html__();
esc_html_e();
esc_html_x();
```

Most of these functions are defined with PHP. These functions are also available in JavaScript from the `@wordpress/i18n` package. The following functions are available.

- `__()`
- `_n()`
- `_x()`
- `_nx()`

Creating a Translation Template with the WP CLI

In this lecture, we created a template for our translations with the WP CLI. The WP CLI is a tool for managing WordPress sites with the command line. This

tool can be installed manually. If you're using Local, this tool is available via the site shell option. You can test the installation of the WP CLI by running the following command:

```
wp --version
```

We can create a template with the CLI by running the following command in our plugin's directory:

```
wp i18n make-pot . languages/udemy-plus.pot
```

This command accepts the location to find the translation messages and the name of the file. After running this command, WordPress will scan the directory for all uses of WordPress's official translation functions. It'll extract the strings into a file called **udemy-plus.pot**.

WordPress recommends storing the files in a directory called **languages**. The name of the file should be named after the name of your plugin. Lastly, the file extension should be **pot** to indicate that this file should serve as a starting point for all translators.

Resources

- WP CLI Handbook

Loading Translations

In this lecture, we began loading the translations from the **languages** directory of our plugin. WordPress recommends using the `init` hook to begin this process.

```
add_action('init', 'up_load_php_translations');
```

From our function, we called the `load_plugin_textdomain()` function. This function has three arguments, which are the text domain of our plugin, a deprecated argument, and the directory where the translations are stored relative to the plugins directory.

```
load_plugin_textdomain(
  'udemy-plus',
  false,
  'udemy-plus/languages'
);
```

Adding a New Language

In this lecture, we used **Loco Translate** to add a new language. There are no notes for this lecture.

Loading Block Translations

In this lecture, we loaded our translations for our blocks. First, we need to use the `wp_enqueue_scripts` hook. However, we must wait until WordPress has registered and enqueued our block's scripts. We can do so by setting the priority to a high value. By default, WordPress uses a priority of `10`.

```
add_action('wp_enqueue_scripts', 'up_load_block_translations', 100);
```

Afterward, from the function, we created an array of all the block handles and looped through all of them. The handle name of a block is the name with the word `editor-script` appended to it.

```
$blocks = [
  'udemy-plus-fancy-header-editor-script',
  'udemy-plus-advanced-search-editor-script',
  'udemy-plus-page-header-editor-script',
  'udemy-plus-featured-video-editor-script',
  'udemy-plus-header-tools-editor-script',
  'udemy-plus-auth-modal-script',
  'udemy-plus-auth-modal-editor-script',
  'udemy-plus-recipe-summary-script',
  'udemy-plus-recipe-summary-editor-script',
  'udemy-plus-team-members-group-editor-script',
  'udemy-plus-team-member-editor-script',
  'udemy-plus-popular-recipes-editor-script',
  'udemy-plus-daily-recipe-editor-script'
];

foreach($blocks as $block) {
  wp_set_script_translations(
    $block,
    'udemy-plus',
    UP_PLUGIN_DIR . 'languages'
  );
}
```

On each loop, we're using the `wp_set_script_translations()` function. This function has three arguments, which are the name of the block, the text domain, and the directory where the translations are stored.

Resources

- List of Block Handles

Uninstalling a Plugin

In this lecture, we prepared our plugin for uninstallation by creating a file called **uninstall.php**. WordPress will run the code in this file when our plugin is deleted from a site. Before running the code, you should always check if the user has authorized this action by checking for a constant called `WP_UNINSTALL_PLUGIN`. If this constant is not defined, the script should be exited immediately.

```
if (!defined('WP_UNINSTALL_PLUGIN')) {
  exit;
}
```

Afterward, we deleted the plugin's options by using the `delete_option()` function, which accepts the name of the option.

```
delete_option('up_options');
```

Lastly, we used the `$wpdb` variable to run a function called `query()` for executing a custom query. This query uses the `DROP TABLE` keywords to delete a table. The `IF EXISTS` keywords will check if the table exists before deleting it. If it doesn't, the query will not throw an error. Errors can disrupt the uninstallation process, which we shouldn't since our plugin may become unremovable.

```
global $wpdb;
$wpdb->query(
  "DROP TABLE IF EXISTS {$wpdb->prefix}recipe_ratings"
);
```

Debugging Hooks and Queries

In this lecture, I talked about a plugin called **Query Monitor** for helping us debug a plugin or theme by inspecting the queries and hooks generated by WordPress/third-party plugins. There are no notes for this lecture.

Resources

- Debugging in WordPress
- Query Monitor
- Debug Bar

Quick Adjustments to the Theme

In this lecture, we updated our theme's template parts by replacing the static versions of the template with their respective blocks. There are no notes for this lecture.

Resources

- Updated Template Parts

Bundling Plugins with TGMPA

In this lecture, we talked about how to bundle plugins with a theme by using a class called **TGM Plugin Activation**. This class is completely free. It'll check a site's plugins for a list of plugins, and generate a UI for installing/activating missing plugins. Check out the resource section for a downloadable link to the class.

You can include this class from within the **functions.php** file. Next, you can use a hook for registering a set of plugins called `tgmpa_register`.

```
add_action('tgmpa_register', 'u_register_plugins');
```

We defined a function for handling this process. In this function, we had an array for a set of plugins that should be activated within a theme.

```
$plugins = array(
  array(
    'name'      => 'Regenerate Thumbnails',
    'slug'      => 'regenerate-thumbnails',
    'required'  => false,
  ),
  array(
    'name' => 'Udemy Plus',
    'slug' => 'udemy-plus',
    'source' => get_template_directory() . '/plugins/udemy-plus.zip',
    'required' => true,
  ),
);
```

There are four options we explored.

- `name` - The human-readable name of the plugin.
- `slug` - The slug assigned by WordPress from the official repo. If the plugin is from a local source, this would be the name of the plugin folder.

- `source` - A path to the zip file for downloading the plugin locally.
- `required` - Whether the plugin should be required or recommended.

Next, we created an array of settings.

```
$config = array(
  'id' => 'udemy',
  'menu' => 'tgmpa-install-plugins',
  'parent_slug' => 'themes.php',
  'capability' => 'edit_theme_options',
  'has_notices' => true,
  'dismissable' => true,
);
```

In this array, we added the following settings:

- `id` - A unique ID to identify our list of plugins since other developers can also require plugins.
- `menu` - A unique slug since the class will add a menu item to the dashboard.
- `parent_slug` - The name of the parent menu that the new page should be assigned under.
- `capability` - The capability required for installing/activating plugins.
- `has_notices` - Whether the class should display notices if the site does not have the required plugins.
- `dismissable` - Whether the notices can be dismissed.

After creating these arrays, we passed them onto a function called `tgmpa()`, which accepts the array of plugins and configuration settings.

```
tgmpa( $plugins, $config );
```

Once the plugins have been registered, the class will handle generating a UI for installing/activating plugins. If you have a locally sourced plugin, I highly recommend storing the zip file of a plugin in a separate directory called **plugins**.

Resources

- TGM Plugin Activation
- Regenerate Thumbnails

The End

In this section, I give one final farewell!

Conclusion

Thanks for watching! Please remember to rate the course. I hope to see you in another one of my courses.