LECTURE NOTES

Modern PHP Developer

Luis Ramirez

Section 1: Introduction

In this section, we talk about what PHP is, what you can expect to learn, and how to install PHP on your machines.

What is PHP?

There are no notes for this lecture.

Download the Free E-Book

There are no notes for this lecture.

PHP Environment

In this lecture, we learned about environments and took the time to set up an environment for PHP. First, let's understand the concept of an environment.

In the programming world, an environment is a place where you can write, test, and run computer code. Think of it like a virtual workspace with all the tools you need to make a program work.

Generally, there are two types of environments, local and production.

Local Environment

A local environment refers to the setup you have on your personal computer or local machine, where you can write, test, and run your code. Local environments are primarily used for development and testing purposes.

This gives you, the developer, the opportunity to break things without disrupting your users. You can freely modify and make changes before shipping a program to the real world.

Production Environment

On the other hand, a production environment is a live environment where your code is actually deployed and running for end users.

A production environment is typically hosted on a remote server, and it is designed to handle the demands of real-world usage, including security, scalability, and reliability. The production environment is also monitored, managed, and maintained by a team of professionals to ensure optimal performance.

Using Replit

For this course, we'll be using Replit to help us get started. Watch the video for instructions on how to set up a PHP environment on Replit.

What about Docker?

Professional PHP developers use Docker to manage their environments. Docker is a bit too advanced for us right now, but it will be covered in a future lecture. For now, Replit will suffice.

Resources

- XAMPP
- MAMP
- Laragon
- Replit

Section 2: Working with Data

In this section, we discovered the basic syntax of PHP to help us write dynamic web pages.

The PHP Interpreter

In this lecture, we learned about the PHP interpreter, whose job it is to transform our code into machine code. PHP is an open-source general-purpose scripting language for developing web applications.

- Open-source software (OSS) refers to code that has been freely released to the public.
- A general-purpose language is a programming language that can be utilized to develop different kinds of programs.
- PHP is considered to be a scripting language because it's interpreted.

Compiled vs. Interpreted

So, what does it mean when a programming language is interpreted? Computers are only capable of understanding binary code. Binary code can be difficult to read and write. Herein lies the problem.

For this reason, programming languages were introduced for writing instructions to be executed by a machine. While programming languages are easier to read, computers don't understand the instructions we're giving them. Therefore, our code must be transformed into machine code. This process is known as compilation.

A compiler and an interpreter are both tools used to translate computer code into a form that a computer can execute. The main difference between the two is how they do it:

- A compiler translates the entire source code of a program into machine code (binary) in one go and saves it as an executable file. This file can then be run as many times as needed without the need to recompile it.
- An interpreter, on the other hand, executes the code line by line, translating each line of code into machine code and then executing it immediately. It does not generate an executable file.

Think of it this way: **a compiler is like a translator who translates a book into another language, then gives you the translated book to read. An interpreter is like a translator who translates a sentence, then reads it to you immediately.**

PHP is interpreted. It's safe to say that PHP is a scripting language. However, you can also call it a programming language, and no one would complain otherwise.

Resources

- PHP GitHub

Running a PHP Script

In this lecture, we ran our first PHP script. PHP files can be created with the **.php** extension. In a PHP file, you can freely write plain text like so:

```
Hello World!
```

PHP supports all HTML syntax as well.

```
<h1>Hello World</h1>
```

Static vs. Dynamic Websites

Websites that deliver the same content to visitors regardless of what action the user performs are known as static websites.

1. Browsers can send a request to a server for a specific file.
2. Servers will search for that file.
3. If a file is found, the file is sent back to the browser.

On the other hand, dynamic websites generate unique content to its visitors based on their actions.

1. Browsers can send a request to a server for a specific file.
2. Servers will search for that file.
3. If the file is a PHP file, the PHP file will run through the PHP interpreter, generating an HTML output.
4. The HTML output is sent back to the browser.

PHP Tags

In this lecture, we learned about how to enter PHP mode. By default, PHP does not process raw text or HTML. We must explicitly enter PHP mode to begin giving machines instructions. Entering PHP mode can be done with a pair of PHP tags.

```
<?php ?>
```

Multiple PHP tags can be written inside a PHP file.

```
<?php ?>
<?php ?>
```

They can be written from within HTML like so.

```
<h1><?php ?></h1>
```

However, you cannot write HTML inside PHP like so.

```
<?php <h1></h1> ?>
```

This is because the rules of PHP mode are different from HTML mode.

Inside PHP tags, we must write valid code. In the English language, there are rules for how English is read and written. There's a specific structure we must follow for writing English, such as ending a sentence with a period or capitalizing the first word of a sentence. The rules for human languages are referred to as grammar.

In a similar sense, programming languages have rules for how they're read and written. These rules are referred to as a programming language's syntax.

The echo Keyword

In this lecture, we learned about the `echo` keyword. Keywords are a feature in PHP for performing specific actions. If we type a keyword, the PHP interpreter understands that we're trying to perform a specific action based on that keyword.

The `echo` keyword allows us to render content onto the screen.

```
echo "Hello World!";
```

After the `echo` keyword, we can write our message in pair of quotes (`""`). Lastly, we must end our line of code with a semicolon character (`;`).

A statement is another way of describing a single instruction given to a machine or a single line of code. Programming languages give us the ability to communicate with a machine. Instructions can be given to a machine to perform various tasks, from sending an email to processing a transaction.

You can think of a statement as a single sentence in the English language. Multiple sentences can be combined to create a book. In a similar sense, multiple statements can be combined to create an application.

Multiple statements can be written like so.

```
echo "Hello World!";
echo "My name is John.";
```

Resources

- List of Keywords

Comments

In this lecture, we explored comments in PHP, which are similar to comments in HTML and CSS. They're notes that developers can add without PHP trying to process the contents of a comment.

There are three syntax options for writing comments.

```
// Single-line comment

/*
Multiline Comment
*/

# Single-line comment
```

If you were to run the above code in a PHP file, nothing would get outputted since comments don't get processed.

Variables

In this lecture, learned how to create variables. Variables were introduced for storing data, ranging from addresses to the price of a product. Here's an example of a variable.

```
$age;
```

Variables must always start with the `$` character. This character is followed by the name. The following rules must be adhered to for names to be valid:

- May contain alphabetic characters, numbers, and underscores.
- **MAY NOT** contain special characters. (i.e., **#**, **^**, **&**, **@**)
- The first letter may start with a letter or underscore. Numbers are not allowed.

In some cases, you may need to use multiple words in a variable name. Typically, there are three common naming conventions adopted by developers.

- **Camel Casing** - All words are capitalized except for this first word. Example: `$thisIsMyAge`
- **Pascal Casing** - All words are capitalized. Example: `$ThisIsMyAge`
- **Snake Casing** - All words are separated with an `_` character. Example: `$this_is_my_age`

Assignment Operator

In this lecture, we learned how to use the assignment operator to store a value in a variable. But what is an operator? Operators are symbols that accept a value to create/produce a new value.

The assignment operator will instruct PHP to store a value inside a variable. It's written with the `=` character. You can use it like so.

```
$age = 29;
```

We can reference a variable by typing the variable's name like so.

```
echo $age;
```

This outputs:

```
29
```

We have the option of updating a variable after creating it using the same syntax. PHP is smart enough to update a variable if it has already been created.

```
$age = 29;
$age = 30;

echo $age;
```

This outputs:

```
30
```

Keep in mind that variables are case-sensitive. If you make a typo when updating a variable, you may accidentally create a new variable. `$age` is completely different from `$Age`.

Resources

- Operators

Data Types

In this lecture, we learned about data types. Data types are categories for your data. Programming languages require data to be put into categories for easier debugging and optimization of your program. There are two types of systems seen across programming languages.

- **Statically-Typed** – Developers must explicitly set the type of a variable. The type may never change after declaration.
- **Dynamically-Typed** – Developers do not need to set the type. Data types may change at any time.

PHP is considered to be a dynamically-typed language. Dynamically-typed languages are easier for beginners to learn, but there are downsides. They can be slower since the programming language takes care of assigning the type. In addition, variables may not always have the same type throughout the lifetime of a program, which can cause unexpected behavior.

PHP has nine data types which are the following:

- `null`
- `bool`
- `int`
- `float`

- `string`
- `array`
- `object`
- `callable`
- `resource`

The var_dump() Function

In this lecture, we learned about functions, which are a feature for performing a specific set of instructions. An official list of functions can be found in the resource section.

To get started, we looked at the `var_dump()` function, which outputs the data type and value of a variable. All functions are written by typing their name, followed by a pair of parentheses. Inside the parentheses, we can pass on a value to the function.

```
var_dump(30);
```

This outputs:

```
int(30)
```

Resources

- List of Functions

Null Data Types

In this lecture, we learned about the `null` data type. This data type is used for variables that store nothing. It can be useful for variables that will be needed later, but you don't have a value to assign to them immediately.

You can assign a `null` value to a variable like so:

```
$data = null;
```

`null` is case-insensitive. You can also use `NULL`. Either solution is valid. Most developers prefer to use lowercase letters.

We can view the value and data type of a variable by using the `var_dump()` function like so:

```
var_dump($data);
```

This should output `NULL`.

Boolean Data Type

In this lecture, we learned about the boolean data type. They're useful for storing values that answer yes or no questions. You can set a variable to `true` or `false`. These values assign the `bool` data type to a variable.

```
$isHungry = true;
$hasPermission = false;
```

Alternatively, you can use uppercase letters like so: `TRUE` or `FALSE`. You'll notice that the variable names are questions. It's common practice for variables that store booleans to be named as questions.

Integers and Floats

In this lecture, we learned about integers and floats, which are data types for numbers. PHP introduces two data types for numbers because numbers with decimal values use more memory than whole numbers. To optimize your program, two separate data types were introduced to reduce memory usage.

The **integer** data type can store whole numbers. The **float** data type can store numbers with decimal values.

```
$age = 29; // integer
$price = 123.45 // float
```

Why is it called float? Since the decimal separator can be positioned anywhere in the number, it's considered to be a character that can float anywhere within a number.

Both data types support negative numbers.

```
$age = -29; // integer
$price = -123.45 // float
```

Lastly, both data types can have `_` characters for readability.

```
$distance = 5_000_00_0;
```

PHP will strip these characters away during runtime. They do not affect your program. You can insert them wherever you'd like. They're completely optional.

String Data Type

The string data type is available for storing text. The word "string" can be a weird name for a data type. If you think about it, a group of characters is strung together, hence the word string.

Strings are created with single quotes or double quotes.

```
$firstName = 'John';
$lastName = "Smith";
```

PHP supports string interpolation, which is the process of replacing a placeholder in a string with a variable. For example, we can do the following:

```
$firstName = 'John';
$lastName = "$firstName Smith";
```

This feature is only available for strings created with double quotes. In some cases, you may want to write text immediately after a placeholder. You can wrap the variable with curly brackets to prevent the text from being interpreted as part of the variable name.

```
$firstName = 'John';
$lastName = "{$firstName}™ Smith";
```

Lastly, we have the power to access a specific character of a string by using square brackets. PHP assigns an index to each character in a string. The first character has an index of 0, the second character has an index of 1, and so on and so forth.

To access the third character of a string, we can do the following:

```
var_dump($lastName[2]);
```

We can also update a specific character with this syntax.

```
$lastName[2] = "X";
```

Exercise: Data Type

This exercise is meant to be tackled on Udemy. Check out the resources for a link to the exercise on GitHub.

Resources

- Exercise PHP Data Types

Arrays

In this lecture, we learned about arrays. Arrays are useful for storing a collection of data. Arrays can be created by using a pair of `[]` as the value for a variable like so:

```
$food = ["Salad", "Burger", "Steak"];
```

Existing items can be accessed or updated by their index. Behind the scenes, PHP assigns a number to each item, starting from `0` and incrementing by 1 for each item. For example, we can update the second item in the array like so:

```
$food[1] = "Chicken"; // Change "Burger" to "Chicken"
```

New items can be added to an array by omitting a number from the square brackets. PHP will add the item to the end of the array.

```
$food[] = "Tomato Soup";
```

Associative Arrays

In this lecture, we learned how to assign names to keys in arrays. Instead of allowing PHP to assign numbers, you can override this behavior with custom names to better identify each item in array.

```
$food = [
  "john" => "Salad",
  "jane" => "Burger",
  "sam" => "Steak"
];
```

Associative arrays can be created using the `=>` character to separate the key and value. We can access a specific item from an associative array by the key name.

```
var_dump($food["john"]);
```

Multidimensional Arrays

In this lecture, we learned about multidimensional arrays. A multidimensional array is an array that contains more arrays. This allows you to create a nested structure of complex data. Here's an example:

```
$food = [
  "john" => ["Salad", "Curry"],
  "jane" => "Burger",
  "sam" => "Steak"
];
```

As you can see, the `john` key in the array is storing an additional array. We can access items from within a nested array by using square bracket syntax

```
var_dump($food["john"][1]);
```

In some cases, you may accidentally attempt to access an array from within an array that doesn't exist like so.

```
var_dump($food["bob"][1]);
```

In these cases, you may get the following error: **Trying to access array offset on value of type null.**

The error states that the inner array doesn't exist, so it can't access the item in the array. If you ever encounter this error, you should always double-check that you're accessing an existing array correctly.

Type Casting

In this lecture, we learned how to typecast a value, which is the process of changing a value from one datatype to another. The value itself may change to accommodate the datatype.

Converting a value into a boolean will always produce a `true` value unless the value is the following:

- the integer value `0`
- the float values `0.0` and `-0.0`
- an empty string
- the string `"0"`
- an array with zero elements
- the type `NULL`

Converting a value into an integer follows the current rules:

- A boolean `false` becomes `0`, and a boolean `true` becomes `1`.
- Floating point numbers will always be rounded toward zero.
- A string that is either fully numeric or leading numeric is converted to the corresponding numeric values. All other strings evaluate to zero.

- The value null is always converted to zero.

Converting a value into a float follows the same rules as integers with the addition of the following rules:

- A numeric or leading numeric string will resolve to the corresponding float value. All other strings will be evaluated to zero.
- All other types of values are converted first to integers and then to a float.

Converting a value into a string follows the current rules:

- A `false` boolean value becomes an empty string, and a `true` value becomes the string `"1"`.
- Integers and floats are converted to a textual representation of those numbers.
- All arrays are converted to the string `"Array"`.
- The value NULL is always converted to an empty string.

An array is used to store a bunch of elements to be accessed later. Arrays can contain zero, one, or more elements. Therefore, converting values of type integer, float, string, bool, and resource creates an array with a single element. The element can be accessed at the index zero within the new array. Casting a `NULL` into an array will give you an empty array.

Resources

- Typecasting Gist

Type Juggling

In this lecture, we learned that type juggling is a behavior in PHP where values are automatically type cast without our explicit permission. If we pass on the wrong data type, PHP may attempt to typecast the value automatically to the correct data type.

For example, the `echo` keyword expects a string. If we pass in an integer, the value is typecasted into a string.

```
echo 50;
```

PHP automatically performs this task, so we don't have to do the following:

```
echo (string) 50;
```

Resources

- Echo Keyword

Arithmetic Operators

In this lecture, we learned about arithmetic operators. These operators allow us to perform math. Here's an example of using the operators for addition, subtraction, multiplication, and division.

```
$data = 1 + 2 - 3 * 4 / 5;
```

PHP adheres to the order of operations. We can use parentheses to change the outcome of an equation like so:

```
$data = 1 + (2 - 3 * 4) / 5;
```

The modulo operator (`%`) performs division and returns the remainder.

```
$data = 11 % 2; // -> 1
```

The exponentiation operator (`**`) performs exponentiation where the value on the left of the operator is the base and the value on the right is the exponent.

```
$data = 10 ** 2; // -> 100
```

The Problem with Floats

Whenever you're working with floats, you come across situations where the float value is unexpected. This is a problem with most programming languages since dealing with decimal values is challenging.

Before PHP can execute our application, it must be compiled into machine code. Machine code is the only language machines are able to understand. Programming languages were introduced to be easier to read.

During this process, float values can become broken. There are ways to avoid this, which will be explored in future lectures.

Resources

- Arithmetic Operators
- Floating Guide

Assignment Operators

In this lecture, we explored different ways of assigning values to variables. PHP offers variations of arithmetic operators with the assignment operator. These operators allow you to apply mathematical operations to an existing value.

```
// Assignment Operators (= += -= *= /= %= **=)
$a = 10;
var_dump($a); // -> 10

$a += 2;      // $a = $a + 2;
var_dump($a); // -> 12

$a -= 2;      // $a = $a - 2;
var_dump($a); // -> 10

$a *= 10;     // $a = $a * 10;
var_dump($a); // -> 100

$a /= 10;     // $a = $a / 10;
var_dump($a); // -> 10

$a %= 6;      // $a = $a % 6;
var_dump($a); // -> 4

$a **= 4;    // $a = $a ** 4;
var_dump($a); // -> 256
```

Resources

- Assignment Operators

Comparison Operators

In this lecture, we learned about comparison operators. The comparison operators allow you to compare two values. PHP can compare values with different types.

If you were to use `==` or `!=`, the datatypes of a value are typecasted if they're not the same data type. Whereas the `===` and `!==` will not, which can cause a comparison to fail. In most cases, you should be strict with your comparisons to avoid headaches.

PHP also has two operators comparing values that aren't equal to each other, which are `!=` and `<>`. There are no differences between these operators.

```
// Comparison Operators (== === != <> !== < > <= =>)
var_dump(1 == "1"); // -> true
var_dump(1 === 1);  // -> true
var_dump(1 != 2);   // -> true
var_dump(1 <> 2);   // -> true
var_dump(1 !== "1"); // -> true
var_dump(1 < 2);    // -> true
var_dump(2 > 1);    // -> true
var_dump(1 <= 2);   // -> true
var_dump(2 >= 1);   // -> true
```

Resources

- Comparison Operators

Error Control Operators

In this lecture, we learned how to suppress errors. You can suppress errors with the error control operator (`@`). This operator should be used sparingly. PHP offers various features to help you avoid errors, so you shouldn't need this operator often.

```
// Error Control Operators
@var_dump((string) [1]);
```

Incrementing and Decrementing Numbers

In this lecture, we learned how to update numbers by one using the increment/decrement operators. The increment/decrement operators allow you to add/subtract a value by 1. You can position these operators before or after a value. If placed after a value, PHP will return a value before updating the value. Whereas placing it before the value will cause the value to be updated before returning it.

```
// Increment/Decrement Operators(++ --)
$a = 10;

var_dump($a++);
var_dump($a--);
```

Logical Operators

In this lecture, we learned how to support multiple conditions with logical operators. Logical operators allow us to chain multiple conditions. The `&&` or `and` operators allow us to verify that multiple conditions are truthy. The `||` or `or` operators allow only one condition to be true for the entire expression to evaluate to `true`. Lastly, the `!` operator allows us to check for a `false` value instead of a `true` value.

```
// Logical Operators (&& || ! and or)
var_dump(true && true); // -> true
var_dump(true and true); // -> true
var_dump(true || false); // -> true
var_dump(true or false); // -> true
var_dump(!true);        // -> false
```

Operator Precedence

In this lecture, we learned about operator precedence. If multiple operators are used within a single expression, PHP will prioritize specific operators over others. You can check out the link in the resource section of this lecture for the order.

For example, multiplication has higher precedence than addition.

```
$a = 5 + 2 * 10;
```

But what if operators have the same precedence, like multiplication and division.

```
$a = 5 / 2 * 10;
```

In this case, PHP uses associativity. It varies from operator to operator. However, it'll either start with the operator on the left or on the right to determine which operators have higher precedence.

Some operators have non-associativity. These operators don't allow you to use them within the same expression multiple times like the `<` and `>` operators.

```
$a = 5 < 2 > 8;
```

Lastly, you might encounter problems when using operators that have super lower precedence, like the `and` operator. PHP offers two operators for performing the same thing, which is the `&&` and `and` operators.

However, they have different precedences, which can lead to different results.

```
$a = true && false; // -> false
$a = true and false; // -> true
```

In the above example, the second expression evaluates to `true` even though the second condition fails, which should technically lead to a `false` value. However, the `=` operator has higher precedence, so PHP will assign `true` to the variable before checking the second condition.

This behavior can lead to unexpected results. For this reason, you should avoid the `and` and `or` operators. Instead, use their original variations, which are `&&` and `||`, respectively.

Resources

- Operator Precedence

Constants

In this lecture, we learned about another type of variable called constants. Constants are variables that can never change their value after being created. They can be created with the `const` keyword.

In PHP, we have flexibility with naming things. However, there is a set of reserved keywords we should avoid to prevent conflicts.

Here's how a constant is created.

```
const FULL_NAME = "John Smith";
```

The naming rules for variables apply to constants with the exception of adding the `$` character at the beginning of the name. It's common practice to use all uppercase letters so that it's easier to identify constants. Not required but recommended.

Resources

- Predefined Constants

String Concatenation

In this lecture, we learned how to combine two strings into a single string with a feature called string concatenation. Two strings can be combined with the concatenator operator, which is written with the `.` character.

```
const FULL_NAME = "John Smith";

var_dump("Hello, my name is" . FULL_NAME);
```

This feature can be helpful for inserting constants into a string since string interpolation does not support constants.

Section 3: Adding Logic

In this section of the course, we looked at features in PHP for using data to control the logic of our program.

Terminology: Expressions

In this lecture, we took the time to understand some programming terminology, which will be useful in upcoming lectures. Specifically, we learned about expression. **An expression is any line of code that produces a value. It's as simple as that.**

Resources

- Echo

Control Structures

In this lecture, we learned how to control the logic flow of our program with If statements. An if statement accepts an expression that must evaluate to a boolean value. If the value is `true`, a block of code is executed. Otherwise, nothing happens.

```
$score = 95;

if ($score > 90) {
  var_dump("A");
} else if ($score > 80) {
  var_dump("B");
} elseif ($score > 80) {
  var_dump("C");
} else {
  var_dump("F");
}
```

We can also add the `else if` / `elseif` keyword to perform an alternative condition. If prior conditions fail, the conditions from these statements will be checked. Once a condition passes, other conditions are ignored. An `else` statement can be added for default behavior. You can compare values in an expression.

Resources

- If Statement

Switch Statements

In this lecture, we learned an alternative solution to If statements called switch statements. Switch statements don't have as much flexibility as If statements. They only compare values that match. If a match is found, the code below it is executed.

A switch statement can be written with the `switch` keyword. With this keyword, we can provide a value to match. Afterward, we can add a list of values to compare with the value from the switch statement by using the `case` keyword.

```php
$paymentStatus = 1;

switch ($paymentStatus) {
  case 1:
    var_dump("Success");
    break;
  case 2:
    var_dump("Denied");
    break;
  default:
    var_dump("Unknown");
}
```

In this example, we're also adding the `break` keyword. Once PHP finds a match, it'll execute written below the case statement. This includes code from other case statements. To prevent that from happening, we can end the switch statement with the `break` keyword.

Something to keep in mind with switch statements is that data types do not have to match. For example, take the following:

```php
$paymentStatus = "2";

switch ($paymentStatus) {
  case 1:
    var_dump("Success");
    break;
  case 2:
    var_dump("Denied");
    break;
  default:
    var_dump("Unknown");
}
```

The `$paymentStatus` variable stores a string. However, the cases are integers. Regardless, the values can still match because PHP will typecast the values during comparison.

Match Expressions

In this lecture, we learned about match expressions, which are a feature to compare values and return the results. We can write a match expression with the `match` keyword. This keyword accepts the value to compare with a list of values inside a pair of curly brackets like so.

```php
$paymentStatus = 2;

$message = match ($paymentStatus) {
  1 => "Success",
  2 => "Denied",
  default => "Unknown"
};
```

We can add as many values to compare with. A default value can be added when a match can't be found with the `default` keyword. Match expressions must always return a value. Otherwise, PHP will throw an error. It's always considered good practice to have a default value.

Something to keep in mind is that comparisons are strict. The values must have the same value and data type. If the values are the same but different types the match won't be registered.

Functions

In this lecture, we learned about functions. Functions are blocks of code that are reusable. By using functions, we don't have to copy and paste the same solution over multiple files.

Functions can be defined with the `function` keyword followed by the name, pair of parentheses, and curly brackets. Function names follow the same rules as variable names, with the exception of the name starting with the `$` character.

Here's an example of the variable definition:

```php
function getStatus()
{
  $paymentStatus = 2;

  $message = match ($paymentStatus) {
    1 => "Success",
    2 => "Denied",
    default => "Unknown"
  };

  var_dump($message);
}
```

Functions can be invoked by writing the name with a pair of parentheses. Invoking a function is the process of running the code inside the function. By default, PHP does not execute the code unless it's invoked.

```
getStatus();
```

Function Parameters

In this lecture, we learned how to pass on data to a function by using parameters. Parameters are variables defined in a function's parentheses that can be updated when a function is called. For example, a parameter can be defined like so.

```
function getStatus($paymentStatus)
{
  // Some code...
}
```

We can pass on data to a function like so:

```
getStatus(1);
```

The `$paymentStatus` parameter will hold the value 1. If we do not pass on data, PHP will throw an error.

We can add additional parameters by separating each parameter with a comma. In addition, parameters may have optional values. If a function is not provided data, PHP will fall back to the default value.

```
function getStatus($paymentStatus, $showMessage = true)
{
  // Some code...
}
```

In some cases, you may hear the word **argument** to describe a parameter. Parameters and arguments can be interchangeable terms, but there is a difference between them. A parameter describes the variable in the function definition, whereas an argument describes the value itself passed into the function.

Function Return Values

In this lecture, we learned how to expose data outside of a function. Data defined inside a function is only available to a function. If we want to expose data outside the function, we must use the `return` keyword followed by the value.

```
function getStatus($paymentStatus, $showMessage = true)
{
  $message = match ($paymentStatus) {
    1 => "Success",
    2 => "Denied",
    default => "Unknown"
  };

  if ($showMessage) {
    var_dump($message);
  }

  return $message;
}
```

Keep in mind, PHP stops executing logic from within a function once a value is returned. Typically, it's advised to return a value at the end of the function.

The value returned by the function can be stored in a variable like so:

```
$statusMessage = getStatus(1);
```

This allows us to use the message returned by the function in other areas outside of the function. Returning values is optional but can be a great way to use data after a function has finished executing.

Type Hinting & Union Types

In this lecture, we discovered type hinting for enforcing data types in our function's parameters and return types. PHP can guess the data type of our variables, but we have the power to explicitly set the data type for debugging.

A function's data type can be set by adding a `:` followed by the type like so:

```
function getStatus($paymentStatus, $showMessage = true): string
```

In some cases, we may want to return null from our function. We can add the `?` character before the return type like so:

```
function getStatus($paymentStatus, $showMessage = true): ?string
```

Alternatively, our functions may not return data at all. In these cases, we can use the `void` data type, which was designed for functions that don't return anything.

```
function getStatus($paymentStatus, $showMessage = true): void
```

We're not limited to setting the return type. Parameters can be type hinted too. The data type must be added before the parameter name like so.

```
function getStatus(int $paymentStatus, bool $showMessage = true): ?string
```

Multiple data types can be assigned by separating them with the `|` character. These are known as union types.

```
function getStatus(int|float $paymentStatus, bool $showMessage = true): ?string
```

Lastly, if you don't want to chain data types and would rather accept any data type, you can use the `mixed` type.

```
function getStatus(mixed $paymentStatus, bool $showMessage = true): ?string
```

Strict Types

In this lecture, we enforced strict types for functions by using the `declare` directive. This keyword allows us to enable specific PHP features. PHP is a weakly typed language. Even if we use type hinting, that doesn't mean PHP will stop us from passing in a value with an invalid data type.

For debugging purposes, it's considered good practice to enable strict typing.

```
declare(strict_types=1);
```

Adding this bit of code will apply strict typing to a specific file. If you have another PHP file in your application, that file won't have this feature enabled. You must add it to every file that you want strict types.

While Loop

In this lecture, we learned how to repeat a series of actions based on a condition with the `while` loop. The `while` loop accepts a condition. If the condition evaluates to `true`, a block of code gets executed. This process is repeated until the condition evaluates to `false`.

```
$a = 1;

while ($a <= 15) {
  echo $a;
}
```

We're using a new keyword called `echo`, which outputs a value. Unlike the `var_dump()` function, the datatype is not rendered with the value.

This code creates an infinite loop. Infinite loops consume resources, which can cause a system to crash. Since the `$a` variable is never updated, the condition evaluates to `true`. It's considered good practice to update the variable on each iteration.

```
while ($a <= 15) {
  echo $a;

  $a++;
}
```

We're using the `++` increment operator. This operator adds 1 to a value. This should prevent the loop from running infinitely.

In some cases, you may want to execute the block of code at least once. You can use the `do while` loop to achieve this behavior.

```
do {
  echo $a;
```

```
  $a++;
} while ($a <= 15);
```

The condition in the `while` keyword is performed **after** the block of code has been executed.

For Loop

In this lecture, we looked at an alternative to the `while` loop called the `for` loop. Similar to the `while` loop, we can use a condition to determine when a loop should run. However, rather than partially writing the logic pertaining to the loop outside of the parentheses, everything is centralized in one location.

```
for ($i = 1; $i <= 15; $i++) {
  echo $i;
}
```

There are three steps for `for` loops.

1. Initializer - A variable that can be initialized for iterating through the loop.
2. Condition - An expression that evaluates to a boolean. If `true`, the block of code is executed.
3. Increment - An expression that should update the variable from the first step.

PHP always executes the initializer once before checking the condition. Afterward, it'll execute the block of code. After the block of code finishes running, the third expression is executed. PHP checks the condition and repeats the process.

In some cases, you might want to skip an iteration or stop the loop completely. You can do so with the `continue` or `break` keywords, respectively.

```
for ($i = 1; $i <= 15; $i++) {
  if ($i == 6) {
    continue;
  }

  echo $i;
}
```

A new comparison operator was introduced called `==`. This operator compares two values for equality. In this example, we're checking if the `$i` variable is equal to `6`. If it is, the current iteration is skipped. PHP will not echo the number and move on to the next iteration.

If you want to stop the loop altogether, you can use the `break` keyword.

Foreach Loop

In this lecture, we learned about the `foreach` loop, which allows us to loop through an array. This loop accepts the array to loop through, followed by the `as` keyword and a variable to store a reference to each item in the array.

```
$names = ["John", "Jane", "Sam"];

foreach ($names as $name) {
  var_dump($name);
}
```

In some cases, you may want the key to the current item. You can update the expression to include a variable for the key like so.

```
$names = ["John", "Jane", "Sam"];

foreach ($names as $key => $name) {
  var_dump($key);
  var_dump($name);
}
```

Short-Circuiting

In this lecture, we learned about a feature called short-circuiting. PHP will not bother checking other conditions if a previous condition evaluates to `false`. This increases performance since PHP does not perform additional logic.

Take the following example:

```
function example()
{
  echo "example() invoked";

  return true;
```

```
}

var_dump(false && example());
```

Since the first condition is `false`, the `example()` function is never executed. If you were to run the above code, you would not see the message from the function appear in the output.

Section 4: Beginner Challenges

In this section, we tried a few challenges to get us to start thinking like a developer.

Getting Started with Challenges

This lecture does not have any notes. The content is meant to be consumed via video.

Coding Exercise: Resistor Colors

If you want to build something using a Raspberry Pi, you'll probably use resistors. For this exercise, you need to know two things about them:

- Each resistor has a resistance value.
- Resistors are small - so small, in fact, that if you printed the resistance value on them, it would be hard to read.

To get around this problem, manufacturers print color-coded bands onto the resistors to denote their resistance values. Each band has a position and a numeric value.

The first 2 bands of a resistor have a simple encoding scheme: each color maps to a single number.

In this exercise, you are going to create a helpful program so that you don't have to remember the values of the bands.

These colors are encoded as follows:

- black: 0
- brown: 1
- red: 2
- orange: 3
- yellow: 4
- green: 5
- blue: 6
- violet: 7
- grey: 8
- white: 9

The goal of this exercise is to create a way to look up the numerical value associated with a particular color band. Mnemonics map the colors to the numbers, that, when stored as an array, happen to map to their index in the array.

More information on the color encoding of resistors can be found in the Electronic color code Wikipedia article.

Starter Code

```
<?php

// 1. Create function for accepting the color as a string.
// 2. Create an array resistor colors with their relative numeric code
// 3. Return a numeric code based on the color's name

function colorCode($color)
{

}
```

Coding Solution: Resistor Colors

```
<?php

// 1. Create function for accepting the color as a string.
// 2. Create an array resistor colors with their relative numeric code
// 3. Return a numeric code based on the color's name

declare(strict_types=1);

function colorCode(string $color): int
{
```

```
    $colors = [
      "black" => 0,
      "brown" => 1,
      "red" => 2,
      "orange" => 3,
      "yellow" => 4,
      "green" => 5,
      "blue" => 6,
      "violet" => 7,
      "grey" => 8,
      "white" => 9
    ];

    return $colors[$color];
}
```

Coding Exercise: Two Fer

Two-fer or 2-fer is short for two for one. One for you and one for me.

Given a name, return a string with the message:

```
One for name, one for me.
```

Where "name" is the given name.

However, if the name is missing, return the string:

```
One for you, one for me.
```

Here are some examples:

- Alice = `One for Alice, one for me.`
- Bob = `One for Bob, one for me.`
- `One for you, one for me.`
- Zaphod = `One for Zaphod, one for me.`

Starter Code

```
<?php

declare(strict_types=1);

function twoFer(string $name): string
{

}
```

Coding Solution: Two Fer

```
<?php

// 1. Update $name parameter to be optional by adding a default value.
// 2. Return a string using string interpolation to inject the $name variable into the phrase.

declare(strict_types=1);

function twoFer(string $name = "you"): string
{
    return "One for {$name}, one for me.";
}
```

Coding Exercise: Leap Year

Given a year, report if it is a leap year.

The tricky thing here is that a leap year in the Gregorian calendar occurs:

- on every year that is evenly divisible by 4
- except every year that is evenly divisible by 100
- unless the year is also evenly divisible by 400

For example, 1997 is not a leap year, but 1996 is. 1900 is not a leap year, but 2000 is.

For a delightful, four minute explanation of the whole leap year phenomenon, go watch this youtube video.

Starter Code

```php
<?php

declare(strict_types=1);

function isLeap(int $year): bool
{

}
```

Coding Solution: Leap Year

```php
<?php

// 1. Check if year isn't evenly divisible by 4. If so, return false.
// 2. Check if year is evenly divisible by 100 AND
// isn't evenly divisible by 400. If so, return false.
// 3. return true if all conditions fail.

declare(strict_types=1);

function isLeap(int $year): bool
{
  if ($year % 4 !== 0) {
    return false;
  }

  if ($year % 100 === 0 && $year % 400 !== 0) {
    return false;
  }

  return true;
}
```

Section 5: Filling in the Gaps

In this section, we covered a wide variety of topics in PHP that were missing from the previous 4 sections.

Predefined Constants

In this lecture, we explored some predefined constants by PHP. These are some of the more common ones.

- `PHP_VERSION` - Outputs the currently installed PHP version.
- `PHP_INT_MAX` - The largest integer supported in this build of PHP.
- `PHP_INT_MIN` - The smallest integer supported in this build of PHP.
- `PHP_FLOAT_MAX` - Largest representable floating point number.
- `PHP_FLOAT_MIN` - Smallest representable positive floating point number.

In addition, PHP has magic constants, which are constants that have dynamic values but cannot be updated by us. The most common magic constants are the following:

- `__LINE__` - The current line number of the file.
- `__FILE__` - The full path and filename of the file with symlinks are resolved. If used inside an include, the name of the included file is returned.
- `__DIR__` - The directory of the file. If used inside an include, the directory of the included file is returned. This is equivalent to dirname(**FILE**). This directory name does not have a trailing slash unless it is the root directory.

Resources

- Predefined Constants
- Magic Constants

Alternative Syntax for Constants

In this lecture, we learned how to define constants with the `define()` function. It has two parameters, which are the name of the constant and the value. Here's an example.

```php
define("FOO", "Hello World!");
```

```
echo FOO;
```

Other than how the constant is created, using the constant is similar to any other constant.

The main difference between the `define()` function and `const` keyword is that the `define()` function can be used in a conditional statement, whereas the `const` keyword cannot.

```
if (!defined("FOO")) {
  define("FOO", "Hello World!");
}
```

It's common practice to verify that a constant has been created by using the `defined()` function, which accepts the name of the constant to check. If the constant isn't created, we can proceed to create one.

Unsetting Variables

In this lecture, we learned how to release memory from our program by using the `unset()` function. This function accepts the variable to delete from your program. For example.

```
$name = "John";

unset($name);

echo $name; // <~ Throws undefined error
```

In the above example, the `$name` variable gets defined, unset, and then throws an error whenever we try to reference the variable afterward since it no longer exists in our program. PHP throws an undefined error.

In addition, you can remove specific items from an array using this method like so.

```
$names = ["John", "Jane", "Alice"];

unset($names[1]);
```

This will remove the second item from the array but does not reindex the array. If you want to reindex the array, you can use the `array_values` function.

```
$names = array_values($names);
```

You can verify the array was reindexed by using the `print_r()` function.

```
print_r($names);
```

This function outputs the items in array. It's different from the `var_dump()` function since it does not output the data types of each item in the array.

Reading the PHP Documentation

In this lecture, we explored reading the documentation for PHP. There are no notes for this lecture.

Resources

- PHP Manual

Rounding Numbers

In this lecture, we learned how to round numbers. PHP offers three functions for rounding numbers. The first of which is `floor()` and `ceil()`. The `floor()` function rounds a number down, and the `ceil()` function rounds a number up. Both round to the nearest whole number.

```
echo floor(4.5); // Result: 4
echo ceil(4.5); // Result 5
```

The `round()` function allows us to round a number while also adding precision to keep a certain number of digits in the decimal portion of a value. It accepts the number to round, precision, and whether to round up or down when the decimal value is halfway through to the next number.

```
echo round(4.455, 2); // Result: 4.6
```

Resources

- ceil()
- floor()
- round()

Alternative if statement syntax

In this lecture, we learned about an alternative syntax for writing conditional statements. Instead of curly brackets, we can use a `:` character after the condition to denote the beginning of a block of code. The ending of the block of code can be written with the `endif` keyword.

```
<?php $permission = 1; ?>

<?php if ($permission === 1) : ?>
  <h1>Hello Admin</h1>
<?php elseif ($permission === 2) : ?>
  <h1>Hello Mod</h1>
<?php else: ?>
  <h1>Hello Guest</h1>
<?php endif; ?>
```

Another difference between using curly brackets and keywords is that the `else if` keyword can't be used. We must stick with the `elseif` keyword.

Avoiding Functions in Conditions

In this lecture, we learned that you should avoid using functions in conditions whenever possible. Some functions can take a while to execute. If you're executing an expensive function multiple times because of a condition, this can severely affect the overall performance of your app.

For example, take the following:

```
<?php
function getPermission() {
  sleep(2);

  return 2;
}
?>

<?php if (getPermission() === 1) : ?>
  <h1>Hello Admin</h1>
<?php elseif (getPermission() === 2) : ?>
  <h1>Hello Mod</h1>
<?php else: ?>
  <h1>Hello Guest</h1>
<?php endif; ?>
```

In this example, the `getPermission()` function takes two seconds to execute cause of the `sleep()` function. We're calling it twice from both conditions, which can cause our application to take longer to respond.

A better solution would be to outsource the value returned by our function in a variable so that it's only called once.

```
<?php
function getPermission() {
  sleep(2);

  return 2;
}

$permission = getPermission();

?>

<?php if ($permission === 1) : ?>
  <h1>Hello Admin</h1>
<?php elseif ($permission === 2) : ?>
  <h1>Hello Mod</h1>
<?php else: ?>
  <h1>Hello Guest</h1>
<?php endif; ?>
```

Loops

This problem can also affect loops, too.

```php
function getUsers() {
  sleep(2);

  return ["John", "Jane"];
}

for ($i = 0; $i < count(getUsers()); $i++) {
  echo $i;
}
```

In this example, the `getUsers()` function is called on each iteration of the loop, which needs 2 seconds to finish executing.

To fix this, we can outsource the value returned by the function in a variable called `$userCount` and use that variable from within the condition like so.

```php
function getUsers() {
  sleep(2);

  return ["John", "Jane"];
}

$userCount = count(getUsers());
for ($i = 0; $i < $userCount; $i++) {
  echo $i;
}
```

Including PHP Files

In this lecture, we learned how to include code from other files by using the `include`, `include_once`, `require`, and `require_once` keywords. Each of these keywords can include a PHP file in another PHP file.

```php
include "example.php";
include_once "example.php";
require "example.php";
require_once "example.php";
```

The main difference between these keywords is that the `include` keyword(s) throw a warning if a file can't be found. Warnings don't interrupt the rest of the script from running.

The `require` keyword(s) throws a fatal error, which does stop the rest of the script from running. The keywords with the word `_once` only include a file once. If a file has been included before, it will not be included again.

Variadic Functions

A variadic function is a function that accepts an unlimited number of arguments. You can define a variadic function by adding the `...` operator before the parameter name like so.

```php
function sum (int|float ...$num) {
  return array_sum($num);
}
```

In this example, the `sum()` function accepts an unlimited set of numbers. All values are stored in an array called `$num`. Lastly, we're calculating the sum using the `array_sum()` function.

Named Arguments

In this lecture, we learned how to assign values to specific parameters by using named arguments. By default, PHP assigns values to parameters in the order they're presented. In some cases, we may want to set specific values to specific parameters like so.

```php
someFunction(x: 10, y, 5);
```

We can add the name of the parameter before the value with a `:` separating the parameter name and value. So, even if the order does not match, PHP won't have a problem assigning the value to the correct parameter.

Keep in mind, this feature is only available in PHP 8 and up.

Resources

- setcookies() Function

Global Variables

In this lecture, we learned how to use global variables. By default, functions do not have access to variables defined outside of the function. If we want to use a variable, we must use the `global` keyword followed by a list of variables we'd like to use. Multiple variables can be specified by separating them with a comma.

Here's an example:

```
$x = 5;

function foo() {
  global $x;

  echo $x;
}

foo();
```

Global variables can be convenient, but developers often avoid them. It's recommended to pass in the value to the function and have the function return a new value. Global variables can make your application behave unreliably.

Static Variables

In this lecture, we learned about static variables. Static variables are a feature that allows variables defined in functions to retain their values after a function has finished running.

By default, variables defined in a function are destroyed after a function is finished running. As a result, the value gets reset when the function is called again.

By adding the `static` keyword to a variable, the variable will retain the value like so.

```
function foo() {
  static $x = 1;
  return $x++;
}

echo foo() . "<br>"; // Outputs: 1
echo foo() . "<br>"; // Outputs: 2
echo foo() . "<br>"; // Outputs: 3
```

Anonymous and Arrow Functions

In this lecture, we learned about anonymous functions, which are functions without names. Typically, anonymous functions are assigned to variables or passed into functions that accept anonymous functions.

We can define an anonymous function like so.

```
$multiply = function ($num) {
  return $num * $multiplier;
};
```

By storing it in a variable, we can pass on this function as an argument to another function. For example:

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

echo sum(10, 5, $multiply);
```

We're referring to the function as `$callback`. Callback is a common naming convention for functions that get passed into another function that can be called at a later time.

In some cases, you may want to access variables in the parent scope. You can do so with the `use` keyword after the parameter list.

```
$multiplier = 2;
$multiply = function ($num) use ($multiplier) {
  return $num * $multiplier;
};
```

Unlike global variables, PHP creates a unique copy of the variable, so the original variable does not get modified if you attempt to update the value.

A shorter way of writing anonymous functions is to use arrow functions. Here's the same example as an arrow function.

```
$multiply = fn ($num) => $num * $multiplier;
```

Unilke before, we don't have to use the `use` keyword since all variables in the parent scope are accessible to the arrow function. In addition, the code to the right of the `=>` character is treated as an expression. The value evaluated by the expression is the return value.

Keep in mind, arrow functions cannot have a body. You may only write an expression, and arrow functions always return a value. Regular anonymous functions can optionally return values.

Callable Type

In this lecture, we learned about the `callable` type, which is the data type you can add to a parameter that holds a function. Here's an example:

```
function sum (int|float $a, int|float $b, callable $callback) {
  return $callback($a + $b);
}
```

You have a few options when passing in a function. You can either pass in a variable that holds the function, write the function directly, or pass in the name of a regular function.

```
echo sum(10, 5, $someFunction);
echo sum(10, 5, fn($num) => $num * 2);
echo sum(10, 5, 'someOtherFunction');
```

Passing by Reference

In this lecture, we learned how to pass in a variable by reference. If we were to add the `&` character before the name of the parameter, PHP does not create a unique copy of the value and assign it to the parameter.

Instead, PHP will allow us to reference the original variable through the parameter. If we modify the reference, the original variable gets modified too.

```
$cup = "empty";

function fillCup(&$cupParam) {
  $cupParam = "filled";
}

fillCup($cup);

echo $cup; // Output: filled
```

In this example, the `$cup` variable will be set to `filled` since the value gets manipulated from within the `fillCup()` function.

Array Functions

In this lecture, we explored various functions for manipulating arrays. PHP offers a wide assortment of functions. Here are some of the most common ones you may use from time to time.

The `isset()` and `array_key_exists()` functions can be used to verify that a specific key in a function has a value. However, the `isset()` function does not consider `null` or `false` values to be acceptable.

Take the following:

```
$users = ["John", "Jane", "Bob", null];

if (isset($users[3])) {
  echo "User found!";
}
```

In this example, the `isset()` function will return `false` since `$users[3]` contains a `null` value. Despite having a value, this index is considered to be empty. If we want the condition to pass for arrays that have `null` values, we can use the `array_key_exists()` function.

```
$users = ["John", "Jane", "Bob", null];

if (array_key_exists(3, $users)) {
  echo "User found!";
}
```

Another function worth looking at is the `array_filter()` function, which will allow us to filter the items in the array. By default, this function removes empty values in an array.

```
$users = ["John", "Jane", "Bob", null];

$users = array_filter($users);
```

In the example above, the `array_filter()` function returns the `$users` array without the `null` value. If we want to customize the filtering, we can pass in a callback function. This callback function will be given each item in the array and can return a boolean to determine if the value should remain in the array.

```
$users = ["John", "Jane", "Bob", null];

$users = array_filter($users, fn($user) => $user !== "Bob");
```

In this example, `"Bob"` gets filtered out of the array.

The `array_map()` function can be used to modify each item in the array. It has two arguments, which are the array and a callback function that will be given each item in the array. The callback function must return the value or a modified version of the value.

```
$users = ["John", "Jane", "Bob", null];

$users = array_map($users, fn($user) => strtoupper($user));
```

In this example, each user is run through the `strtoupper()` function to convert lowercase letters into uppercase.

The `array_merge()` function can be used to merge two or more arrays into a single array.

```
$users = ["John", "Jane", "Bob", null];

$users = array_merge($users, ["Sam", "Jessica"]);
```

In this example, the second array will be appended to the array creating the following output:

```
["John", "Jane", "Bob", null, "Sam", "Jessica"]
```

We can also sort arrays using PHP's various sorting functions. Refer to the resource section for a link to a complete list of sorting functions.

The first function you'll often use is the `sort()` function, which sorts an array in ascending order.

```
sort($users);
```

Keep in mind, the `sort()` function does not return an array. Instead, it references an array and can modify the existing array.

Another thing to know is that the `sort()` function reindexes your array. If you wish to keep the original indexes, you can use the `asort()` function instead.

```
asort($users);
```

Resources

- Array Functions
- array_filter() Function
- array_merge() - Function
- sort() Function
- Sorting Arrays

Destructuring Arrays

In this lecture, we learned how to destructure an array to extract specific values. There are two ways to grab items from an array. Firstly, we can use the `list` keyword like so.

```
$numbers = [10, 20];

list($a, $b) = $numbers;

echo $a;
```

PHP extracts the values in the order they're defined. So, the `$a` variable will have the value of `10`.

A shorthand syntax is available by using `[]`.

```
[$a, $b] = $numbers;
```

In addition, we can grab named keys from an array like so.

```
$numbers = ["example" => 10, 20];

["example" => $a, 0 => $b ] = $numbers;
```

Working with Files

In this lecture, we explored the various functions for working with file data. The first function we looked at is the `scandir()` function.

```
scandir(__DIR__);
```

This function returns a complete list of files and directories in a specific directory. You can use `__DIR__` constant to check the current directory you're in.

In almost all cases, PHP will also add the following items to the array of files/directories.

- `.` - The current directory.
- `..` - The parent directory.

In addition, we looked at the `mkdir()` and `rmdir()` functions for creating and removing directories, respectively. Both functions accept the directory to create/delete.

```
mkdir("foo");
rmdir("foo");
```

Before working with a file, you should always verify that it exists by using the `file_exists()` function, which accepts the path to a file. It'll return a boolean on whether the file was found or not.

```
file_exists("example.txt");
```

The file size can be retrieved with the `filesize()` function. This function also accepts a path to the file.

```
filesize("example.txt");
```

Data can be written to a file by using the `file_put_contents()` function. This function accepts the name of the file and the data to insert into the file.

```
file_put_contents("example.txt", "hello world");
```

Sometimes, you may find yourself reading file info from the same file multiple times. Behind the scenes, PHP caches the results of some of its filesystem functions. If you want to clear the cache, you can use the `clearstatcache()` function.

Lastly, you can read the contents of a file by using the `file_get_contents()` function.

```
echo file_get_contents("example.txt");
```

Resources

- Filesystem Functions

Section 6: More PHP Challenges

In this section, we tried a few challenges to get us to start thinking like a developer.

Exploring the Challenges

This lecture does not have any notes. The content is meant to be consumed via video.

Coding Exercise: Robot Name

Manage robot factory settings. When a robot comes off the factory floor, it has no name.

The first time you turn on a robot, a random name is generated in the format of two uppercase letters followed by three digits, such as **RX837** or **BC811**.

Create a function that generates names with this format. The names must be random: they should not follow a predictable sequence. 🤖

Hints

Here are the steps you can take to complete this exercise:

1. Generate an array of characters A - Z. Check out the links below for a function to use.
2. Randomize the items in the array. Check out the links below for a function to use.
3. Generate a random number between 100 - 999. Check out the links below for a function to use.
4. Return a string with the first two items from the array and a random number.

Check out these functions for helping you curate a solution to this challenge:

- range() - For generating letters A - Z
- shuffle() - For randomizing the letters.
- mt_rand() - For generating a random number between 100 - 999

Starter Code

```php
<?php

declare(strict_types=1);

function getNewName(): string
{

}
```

Coding Solution: Robot Name

```php
<?php

declare(strict_types=1);

function getNewName(): string
{
    $letters = range("A", "Z");
    shuffle($letters);

    $number = mt_rand(100, 999);

    return "{$letters[0]}{$letters[1]}{$number}";
}
```

Coding Exercise: Armstrong Numbers

An Armstrong number is a number that is the sum of its own digits, each raised to the power of the number of digits.

For example:

- 9 is an Armstrong number, because `9 = 9^1 = 9`
- 10 is not an Armstrong number, because `10 != 1^2 + 0^2 = 1`
- 153 is an Armstrong number, because: `153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153`
- 154 is not an Armstrong number, because: `154 != 1^3 + 5^3 + 4^3 = 1 + 125 + 64 = 190`

Write some code to determine whether a number is an Armstrong number.

Warning: Please do not use arrow functions. They're not supported on Udemy.

Hints

Here are the steps you can take to complete this exercise:

1. Convert the string into an array with each character as an individual value. Check out the links below for a function to use.
2. Loop through the array. Check out the links below for a function to use.

3. Calculate exponentiation with the number in the current iteration with the number of items in the array. Check out the links below for a function to use.
4. Calculate the sum of all values in the array and check if it equals the original number. Check out the links below for a function to use.
5. Return the result as a boolean

Check out these functions to help you curate a solution to this challenge:

- str_split() - To help you grab each digit in the number
- array_map() - To help you go through each number
- count() - To help you count the items in the array
- array_sum() - To help you calculate the sum of all numbers in an array

Starter Code

```php
<?php

declare(strict_types=1);

function isArmstrongNumber(int $number): bool
{

}
```

Coding Solution: Armstrong Numbers

```php
<?php

declare(strict_types=1);

function isArmstrongNumber(int $number): bool
{
    $digits = str_split((string) $number);

    $digits = array_map(function($digit) use ($digits) {
      return $digit ** count($digits);
    }, $digits);

    return array_sum($digits) === $number;
}
```

Coding Exercise: Series

Given a string of digits, output all the contiguous substrings of length n in that string in the order that they appear.

For example, the string "49142" has the following 3-digit series:

- "491"
- "914"
- "142"

And the following 4-digit series:

- "4914"
- "9142"

And if you ask for a 6-digit series from a 5-digit string OR if you ask for less than 1 digit, return an empty array.

Note that these series are only required to occupy adjacent positions in the input; the digits need not be *numerically consecutive*.

Hints

Here are the steps you can take to complete this exercise:

1. Create a conditional statement to check if length of the string is longer than the requested size or smaller than 1.
2. Return an empty array if either condition is true.
3. Otherwise, create an array to store the results.
4. Start looping through series minus the size.
5. Insert a sub-string into the results
6. Return the results

Check out these functions to help you curate a solution to this challenge:

- Google: "PHP How to get the number of characters in a string"
- substr() - To help you get a portion of a string

Starter Code

```
<?php

declare(strict_types=1);

function slices(string $series, int $size) : array
{

}
```

Coding Solution: Series

```
<?php

declare(strict_types=1);

function slices(string $series, int $size) : array
{
    $seriesLength = strlen($series);

    if ($size > $seriesLength || $size < 1) {
        return [];
    }

    $results = [];

    for ($i = 0; $i <= $seriesLength - $size; $i++) {
        $results[] = substr($series, $i, $size);
    }

    return $results;
}
```

Section: 7: Object-Oriented Programming

In this section, we started exploring the basics of object-oriented programming in PHP.

What is OOP (Object-Oriented Programming)?

In this lecture, we talked about programming paradigms and how they apply to OOP. There are no notes for this lecture.

Classes

In this lecture, we learned about classes. Classes are blueprints for creating objects. An object can be anything we want. We can use an object to represent a UI element or a database entry.

Classes can be defined with the `class` keyword.

```
class Account {

}
```

It's standard practice to name your class after the name of your file and to use pascalcasing.

Since classes are just blueprints, we can't interact with the data inside of a class until it has been instantiated. Instantiation is the process of creating an object from a class. We can instantiate a class using the `new` keyword like so.

```
$myAccount = new Account;
$johnsAccount = new Account;
```

We can even create multiple instances.

Properties

In this lecture, we talked about properties, which are variables defined inside a class. We must define a property by using an access modifier followed by the name of the variable.

```
class Account {
  public $name;
```

```
  public $balance;
}
```

An access modifier determines how accessible the property is outside of the class. The `public` keyword allows for a property to be updated anywhere from our script.

We can access the property using the `->` from the respective instance.

```
$myAccount = new Account;

$myAccount->balance = 10;

var_dump($myAccount->balance);
```

Data types can be applied to a property after the access modifier.

```
public string $name;
public float $balance;
```

If we don't assign a value to a property, the value will be `uninitialized`. We can set properties to any type of value, but we can't use complex operations, such as setting a property to a function.

```
public float $balance = intval(5.5); // Not allowed
```

Resources

- Access Modifiers

Magic Methods

In this lecture, we learned about magic methods. A method is the terminology for a function defined inside a class. Magic methods are functions defined inside a class that is automatically called by PHP.

There are various magic methods available, but the most common method is `__construct()`. Most magic methods begin with two `_` characters. It's recommended to avoid using two `_` characters in your own methods to reduce confusion.

```
class Account
{
  public string $name;
  public float $balance;

  public function __construct(
    string $newName,
    float $newBalance
  ) {
    $this->name = $newName;
    $this->balance = $newBalance;
  }
}
```

In this example, the `__construct()` method has the `public` access modifier. This allows for our method to be accessible anywhere. Whenever we create a new instance of the `Account` class, the `__construct()` method gets called.

We can pass on data to this method like so.

```
$myAccount = new Account("John", 100);
```

Resources

- Magic Methods

Constructor Property Promotion

In this lecture, we learned of a shorter way of defining properties and setting their values from within a `__construct()` method. If we add the access modifier to the parameter, PHP will automatically treat the parameter as a class property and set the value passed into the method as the value for the respective parameter.

The solution from the previous lecture can be shortened to this:

```
class Account
{
  public function __construct(
    public string $name,
    public float $balance
  ) { }
}
```

It'll result in the same behavior as before.

Custom Methods

In this lecture, we learned how to define a custom method in a class. One of the benefits of using a method in a class is the ability to access the current instance's properties. For example, take the following:

```
class Account
{
  public function __construct(
    public string $name,
    public float $balance
  ) { }

  public function deposit(float $amount) {
    $this->balance += $amount;
  }
}
```

In this example, we have a method called `deposit()`. This method update the `$balance` property via the `$this` keyword. As you can see, we never have to worry about updating the incorrect property. Methods have access to the property values of the current instance.

In addition, we can return the current instance from our methods to allow method chaining. For example, let's say we updated the `deposit()` method to this:

```
public function deposit(float $amount) {
  $this->balance += $amount;

  return $this;
}
```

Returning the `$this` keyword allows us to chain methods like so:

```
$myAccount = new Account('John', 20);

$myAccount->deposit(50)->deposit(30);
```

Null-safe Operator

In this lecture, we talked about the null-safe operator, which allows us to access a property method without worrying about PHP throwing an error because the instance does not exist. If we attempt to access a property or method on a `null` value, PHP throws an error.

```
$myAccount = null;

$myAccount?->deposit(500);
```

The null-safe operator will check if the value is `null` before accessing a property or method. If a variable is `null`, PHP will not bother running the next portion of code, which prevents an error from being thrown.

Understanding Namespaces

In this lecture, we talked about namespaces. There are no notes for this lecture.

Creating a Namespace

In this lecture, we created a namespace by using the `namespace` keyword. Following this keyword, we must provide a name. Namespaces follow the same naming rules for classes or functions. Generally, most developers use pascalcasing for namespaces.

```
namespace App;

class Account {
```

```
  // Some code here...
}
```

We can access a class from within a namespace by typing the namespace and class like so.

```
$myAccount = new App\Account();
```

However, you may not want to type out the entire path. You can use the `use` keyword to import a class from a namespace. This will allow you to create an instance with just the class name.

```
use App\Account;

$myAccount = new Account();
```

Working with Namespaces

In this lecture, we went over various things to be aware of when using namespaces. Firstly, it's common practice for the namespace to mimick a project's directory. For example, if a class exists in a folder called `App`. The namespace should be called `App`. Not required but recommended to make it easier to find classes.

The second thing to be aware of is that we don't need to use the `use` statement or type out the complete namespace when classes exist in the same namespace.

Another thing to know is that PHP will only resolve classes to the current namespace. For example, let's say we wanted to use a class called `DateTime` from a file in the `App` namespace.

```
new DateTime();
```

PHP will throw an error because it'll attempt to look for the class with the following path: `App\DateTime`. In order to use a class from a different namespace, such as the global namespace, we must add the `\` character before the classname like so.

```
new \DateTime();
```

Alternatively, we can import the class with the `use` keyword without the `\` character.

```
use DateTime;

new DateTime();
```

It's important that the `use` statement to be in the same file as where we're using the class. The `use` statement only applies to the current file.

Lastly, we can add aliases for classes with the `as` keyword.

```
use DateTime as DT;

new DT();
```

Resources

- Name Resolution Rules
- Namespace FAQ

Autoloading Classes

In this lecture, we learned how to automate the process of loading PHP files with classes. There's a function called `spl_autoload_register()` that we can call for intercepting classes that are used in a program, but haven't been defined.

We can pass in a function that will be responsible for loading a specific class, which is passed on to our function as a parameter.

```
spl_autoload_register(function($class) {
  $formattedClass = str_replace("\\", "/", $class);
  $path = "{$formattedClass}.php";
  require $path;
});
```

In the above example, the first thing we're doing is replacing the `\` character with a `/` character since the `require` statement does not allow `\` characters. To accomplish this, we're using the `str_replace()` function, which accepts the string to search for in a given string, the string to

replace it with, and the string to search.

In the first argument, we're escaping the `\` character by using `\\` since the `\` character escapes a string. Escaping a string is the process of telling PHP not to close the string with the closing double-quote.

Next, we created a variable called `$path`, which contains the path to the file that contains the class. Lastly, we use the `require` keyword to load the file.

Using Constants in Classes

In this lecture, we used a constant in a class. Classes do not support the `define()` function for constants. We must use the `const` keyword like so.

```
class Account {
  public const INTEREST_RATE = 2;
}
```

Next, we can access a constant by using the scope resolution operator (`::`).

```
Account::INTEREST_RATE;
```

Alternatively, you can also access a constant from an instance of a class.

```
$myAccount = new Account();
$myAccount::INTEREST_RATE;
```

Static Properties and Methods

In this lecture, we learned how to use static properties and methods. Static properties are similar to constants, except they can have their values modified. We can define a static property with the `static` keyword. This keyword can be positioned before or after an access modifier.

```
class Account {
  public static int $count = 0;
}
```

We can access the static property like so.

```
Account::$count;
```

Alternatively, you can also access a static property from an instance of a class.

```
$myAccount = new Account();
$myAccount::$count;
```

Typically, static properties are avoided since they can be altered anywhere, just like global variables. On the other hand, static methods are popular as a means of creating utility methods. Take the following:

```
class Utility {
  public static function printArray(array $array) {
    echo '<pre>';
    print_r($array);
    echo '</pre>';
  }
}
```

Since the method is static, we can invoke the method without requiring an instance like so.

```
Utility::printArray([1, 2, 3]);
```

OOP Principle: Encapsulation

In this lecture, we learned about an object-oriented programming principle called encapsulation. Encapsulation is the idea of restricting data and methods to a single class.

In your classes, you can prevent outside sources from modifying properties by using the `private` access modifier.

```
public function __construct(
  private string $name,
  private float $balance
) { }
```

By changing the modifier to `private`, only the current class can modify a given property.

But what if we want to read or update the property? In this case, we can create getter or setter methods, which are just methods that can be used for accessing a specific property. Here's an example of a getter.

```
public function getBalance() {
  return "$" . $this->balance;
}
```

It's common practice to set the name of the method to the word "get" followed by the name of the property to grab. One of the main benefits of getters is being able to format values before returning them.

As for setters, they're methods for updating a property. It's standard practice to set the name to the word "set" followed by the name of the property.

```
public function setBalance(float $newBalance) {
  if ($newBalance < 0) return;

  $this->balance = $newBalance;
}
```

The main benefit of setters is that we can validate data before updating a property. Overall, getters and setters give us more power over how our properties are read and set.

We're not limited to private properties. Methods can be private too. This can be useful when you want to subdivide a method into multiple actions without exposing a method outside of a class.

```
public function setBalance(float $newBalance) {
  if ($newBalance < 0) return;

  $this->balance = $newBalance;
  $this->sendEmail();
}

private function sendEmail() {}
```

In the following example, the `sendEmail()` method is private, which will be responsible for sending an email to the user if the balance is successfully updated.

OOP Principle: Abstraction

In this lecture, we talked about the idea of abstraction. Abstraction is the idea of hiding complexities away into a method so that outside sources don't have to worry about them. Most people know how to use coffee machines or cars, but they don't know the inner mechanisms of these machines. Regardless, they're still able to use them.

This idea can be applied to programming. We can define classes to perform complex actions and then use them elsewhere without having to understand the complete implementation details of a class's methods.

OOP Principle: Inheritance

In this lecture, we learned about inheritance. The idea of inheritance is that a class can be derived from another class. Properties and methods from the parent class will be available in the child class as its own properties and methods.

Take the following:

```
class Toaster {
  public int $slots = 2;

  public function toast() {
    echo "toasting bread";
  }
}

class ToasterPremium {
  public int $slots = 4;

  public function toast() {
    echo "toasting bread";
```

```
  }
}
```

In this example, we have two classes with the same property and method, which is redundant. To reduce the amount of code you can write, you can use the `extends` keyword for a class to inherit properties and methods from another class.

```
class Toaster {
  public int $slots = 2;

  public function toast() {
    echo "toasting bread";
  }
}

class ToasterPremium extends Toaster {
  public int $slots = 4;
}
```

In the above example, the `ToasterPremium` class inherits from the `Toaster` class. Therefore, it's not necessary to add the `toast()` method again.

In addition, if we want to override properties or methods from the parent class, we're allowed to do so. PHP prioritizes the child class properties and methods over the parent.

One thing to keep in mind is that the `$this` keyword points to the current instance, even if it's being used in the parent class.

Protected Modifier

In this lecture, we learned about the protected modifier. PHP allows you to override properties, but if a property is `private`, child classes won't be able to change the value of a parent class. You can get the best of public and private properties by using the `protected` modifier.

The `protected` modifier allows child classes to modify parent properties but prevent code outside of a class from changing the property value.

```
protected int $balance = 5;
```

Overriding Methods

In this lecture, we learned about overriding methods. We're allowed to override methods from parent classes. The overriding method must have the same method signature, which shares the name and parameter list.

It's completely possible to invoke a parent's method after overriding it by using the `parent` keyword. For example, let's say a parent class had a method called `foo()`. We can call it like so.

```
parent::foo();
```

If you ever call the original method, you should always call it first. Otherwise, you may get unexpected behavior in your classes.

Lastly, you can use the `final` keyword to prevent a class from being inherited.

```
final class SomeClass {

}
```

Or, you can apply it to a method to prevent the method from being overridden.

```
class SomeClass {
  final public function someMethod() {

  }
}
```

Abstract Classes and Methods

In this lecture, we learn how to create abstract methods and classes. There's some terminology to be aware of when using abstract classes that you'll come across in documentation or tutorials.

- **Implementation** - Refers to a function or method written based on an idea.
- **Concrete Class** - A class that fully implements its methods.
- **Abstract Class** - A class that does not or partially implements its methods.

Abstract classes can be created by using the `abstract` keyword.

```
abstract class Product {

}
```

By defining an abstract class, it cannot be instantiated. So, doing the following produces an error.

```
new Product();
```

Our abstract classes can have implemented methods and unimplemented methods like so.

```
abstract class Product {
  public function turnOn() {
    echo "Turning on product";
  }

  abstract public function setup();
}
```

For unimplemented methods, you can use specify them with the `abstract` keyword. If we extend an abstract class, the abstract method must be implemented from the child class.

```
class SomeProduct extends Product {
  public function setup() {
    echo "setting up product";
  }
}
```

Interfaces

In this lecture, we learned how to use interfaces, which are a feature in PHP for defining how a class should be implemented. Interfaces can contain method definitions that should be implemented by a class. We can create an interface with the `interface` keyword.

```
interface RestaurantInterface {
  public function prepareFood();
}
```

It's common practice to add the word "Interface" to the name. Alternatively, you can begin the name with the capital letter "I." This would produce an interface name such as `IRestaurant`. Either naming convention is valid.

A few things to know about interfaces.

1. Properties are not allowed
2. Methods must be public
3. Methods are automatically abstract. You do not need to add the `abstract` keyword
4. Constants are supported

We can apply an interface to a class by adding the `implements` keyword to the class. Multiple interfaces can be added to a class by separating them with a comma.

```
class RestaurantOne implements RestaurantInterface {
  public function prepareFood() {
    echo "preparing food";
  }
}
```

OOP Principle: Polymorphism

In this lecture, we learned about the final OOP principle, polymorphism. Polymorphism is a principle that states that a variable can have various types. For example, let's say that we applied the `RestaurantInterface` data type to a parameter in a method like so.

```
class FoodApp {
  public function __construct(RestaurantInterface $restaurant) {
    $restaurant->prepareFood();
  }
}
```

By doing so, any class that implements the `RestaurantInterface` interface can be passed into the `__construct()` method of the `FoodApp` class.

Let's assume we had two classes called `RestaurantOne` and `RestaurantTwo` that implement the `RestaurantInterface` interface. As a result, you can pass in instances of these classes to the `FoodApp` class like so.

```
new FoodApp(new RestaurantOne);
new FoodApp(new RestaurantTwo);
```

Both examples are valid.

Polymorphism can also be used with abstract classes. Here are the differences between abstract classes and interfaces.

Abstract Classes

- Methods may have implementations
- Properties allowed
- Public, private, protected methods allowed
- Only one class can be extended

Interfaces

- Methods may not have implementations.
- Properties not allowed. Constants are allowed
- Only public methods are allowed
- Multiple interfaces can be implemented

Anonymous Classes

In this lecture, we learned about anonymous classes, which are just classes that don't have a name. For example, let's say we wanted to pass an anonymous class to the `FoodApp` class. We can do so with the following code:

```
$restaurant = new FoodApp(new class implements RestaurantInterface {
  public function prepareFood() {
    echo "{$this->name} preparing food";
  }
});
```

Similar to regular classes, anonymous classes support interfaces, inheritance, methods, and properties. What if we want to pass on data to a `__construct()` method? We can do so by adding the argument list after the `class` keyword like so.

```
$restaurant = new FoodApp(new class("popup restaurant") implements RestaurantInterface {
  public function __construct(
    public string $name
  ) {}

  public function prepareFood() {
    echo "{$this->name} preparing food";
  }
});
```

Docblocks

In this lecture, we learned about docblocks, which is a format for documenting your functions and classes. They're written as a multiline comment like so.

```
/**
 * Neatly prints an array
 *
 * Outputs an array surrounded with <pre> tags for formatting.
 *
 * @param array $array The array to output
 */
public static function printArray(array $array) {
  echo '<pre>';
  print_r($array);
  echo '</pre>';
}
```

In this example, we're documenting a method called `printArray()`. Docblocks have a few sections. Firstly, you can provide a summary, which should be a sentence long.

If you need to add more information, you can add a description. The summary and description should be separated by a new line. Descriptions don't have limits as to how long they can be.

Afterward, you can start adding tags, which allow you to describe specific parts of your method. For example, you can use the `@param` tag to describe your parameters. The have the following format.

```
@param [<Type>] [name] [<description>]
```

Docblock is slowly losing popularity. However, it can be beneficial to know since it's prevalent in projects that use older versions of PHP. In addition, they can be helpful for providing human-readable descriptions of your function's behavior, parameters, and return values.

Resources

- Docblock

Throwing Exceptions

In this lecture, we learned how to throw an exception from PHP. We can use the `throw` keyword followed by a new instance of the `Exception` class.

```
throw new \Exception("Some error message");
```

The `Exception` class will format your message in the output along with the filename, line number, and other pieces of information related to the error. PHP has other classes for throwing exceptions, you can refer to the resource section for a list of them.

For example, we can use the `InvalidArgumentException` class for arguments that are invalid for a method call.

```
throw new \InvalidArgumentException("Some error message");
```

Why do we say throw an exception? Because it implies that an exception can be caught and can be handled for a more custom experience, such as redirecting the user instead of completely stopping the script.

Resources

- List of Exception Classes

Custom Exceptions

In this lecture, we learned how to create a custom exception class instead of using PHP's exception classes. Custom exceptions can be created to customize the errors thrown by our script. All exceptions derive from the `Exception` class. So, all you have to do is create a class that extends the `Exception` class.

```
class EmptyArrayException extends \Exception {
  protected $message = "Array is empty";
}
```

You can override any of the properties from the class. Check out the resource section for a link to the `Exception` class for a complete list of properties.

You can use the new exception like so.

```
throw new EmptyArrayException();
```

Resources

- Exception Class

Catching Exceptions

In this lecture, we learned how to catch an exception by using the `try catch` statement. If we want to catch errors, we must surround the code with the `try` block.

```
try {
  Utility::printArray([]);
}
```

Next, we must provide a `catch` block.

```
try {
  Utility::printArray([]);
} catch(EmptyArrayException|InvalidArgumentException $e) {
  echo "Custom Exception: {$e->getMessage()} <br />";
```

```
} catch (Exception $e) {
  echo "Default Exception: {$e->getMessage()} <br />";
}
```

In the `catch` block's parameter list, we can accept the error that was thrown. In addition, if we add the type, we can apply the `catch` block to specific exceptions. Multiple exceptions can be handled by a single `catch` block by separating them with a `|` character.

If we want to handle all exceptions, we can add a `catch` block where the `$e` parameter is `Exception` since all exceptions derive from this class.

Lastly, we can add the `finally` block that will be executed regardless if an error gets thrown or not.

```
try {
  Utility::printArray([]);
} catch(EmptyArrayException|InvalidArgumentException $e) {
  echo "Custom Exception: {$e->getMessage()} <br />";
} catch (Exception $e) {
  echo "Default Exception: {$e->getMessage()} <br />";
} finally {
  echo "Finally block <br />";
}
```

The DateTime Class

In this lecture, we looked at how to work with dates by using the `DateTime` class. If we create a new instance of this class, the current date and time are returned encased in an object.

```
$date = new DateTime();

var_dump($date);
```

We can pass in a specific date as a string to the method like so.

```
$date = new DateTime("04/10/2022");
```

By default, dates are interpreted in American format. If we want a date to be interpreted in European format, we must replace the `/` characters with either `.` or `-` like so.

```
$date = new DateTime("04.10.2022");
```

It's possible to alter the timezone by passing in an instance of the `DateTimeZone` class as the second argument to the `DateTime` class. The `DateTimeZone` class accepts a valid timezone, refer to the links in the resource section for a complete list of supported timezones.

```
$timezone = new DateTimeZone("America/Chicago");
$date = new DateTime("04/10/2022", $timezone);
```

The `DateTime` class offers various methods for manipulating the date.

```
$date = new DateTime("04/10/2022", $timezone);
$date->setTimezone(new DateTimeZone("Europe/Paris"))
  ->setDate(2023, 6, 15)
  ->setTime(9, 30);
```

- `setTimezone()` - Change the timezone. Accepts a new instance of the `DateTimeZone` class.
- `setDate()` - Changes the date. Accepts the year, month, and day.
- `setTime()` - Changes the time. Accepts the hour and minutes.

You can output the date alone by using the `format()` method, which accepts a series of placeholders.

```
$date->format("F j Y")
```

Resources

- DateTime Class
- Date Function
- Time Function
- Timezones
- Carbon

Iterator and Iterable Type

In this lecture, we learned how to loop through an object using PHP. Here's some terminology that we use in the lecture:

- **Iterate** – The action of repeating something (aka looping)
- **Iterable** – An object that can be iterated. Arrays are also iterable.

Let's assume we had the following class:

```
class CurrentWeek {
  public \DateTime $date;
  public int $daysFrom = 0;

  public function __construct() {
    $this->date = new \DateTime();
  }
}
```

By default, if we pass in an object to the `foreach` loop, PHP will loop through the properties. If the properties are public, they'll be retrieved. Protected or private properties cannot be looped through.

```
$currentWeek = new CurrentWeek();

foreach ($currentWeek as $key => $value) {
  var_dump($key, $value);
  echo "<br>";
}
```

We can customize how an object is iterated by implementing the `Iterator` interface in our class.

```
class CurrentWeek implements \Iterator {}
```

Inside our class, we must define five methods.

```
class CurrentWeek implements \Iterator {
  private \DateTime $date;
  private int $daysFrom = 0;

  public function __construct() {
    $this->date = new \DateTime();
  }

  public function current() : mixed {
    return $this->date->format("F j Y");
  }

  public function key() : mixed {
    return $this->daysFrom;
  }

  public function next() : void {
    $this->date->modify("tomorrow");
    $this->daysFrom++;
  }

  public function rewind() : void {
    $this->date->modify("today");
    $this->daysFrom = 0;
  }

  public function valid() : bool {
    return $this->daysFrom < 7;
  }
}
```

- `current()` - This function must return a value that will be used in the current iteration of the loop.
- `key()` - This function must return a key that will be associated with the current value.
- `next()` - This function runs after the current iteration is finished running. We can use this opportunity to move on to the next value.
- `rewind()` - This function is called when a new loop is started. Mainly used for resetting the values.
- `valid()` - This function is called after the `rewind()` and `next()` functions to check if the current value is valid.

In our example, we're using the `modify()` method from the `DateTime` class. This method can be used to modify the date using a string format. It's the same type of value you can pass into a new instance of the `DateTime` class.

In some cases, you may want to accept an object that can be looped through. PHP has a data type called `iterable` that can be used like so.

```
function foo (iterable $iterable) {
  foreach ($iterable as $key => $value) {
```

```
    var_dump($key, $value);
    echo "<br>";
  }
}
```

Resources

- Iterator Interface
- Supported Date and Time Formats

Section 8: OOP Challenges

In this section, we tried a few challenges to get us to start thinking like a developer.

OOP Challenges Overview

This lecture does not have any notes. The content is meant to be consumed via video.

Coding Exercise: Nucleotide Count

Each of us inherits from our biological parents a set of chemical instructions known as DNA that influence how our bodies are constructed. All known life depends on DNA!

Note: You do not need to understand anything about nucleotides or DNA to complete this exercise.

DNA is a long chain of other chemicals, and the most important are the four nucleotides, adenine, cytosine, guanine, and thymine. A single DNA chain can contain billions of these four nucleotides, and the order in which they occur is important! We call the order of these nucleotides in a bit of DNA a "DNA sequence."

We represent a DNA sequence as an ordered collection of these four nucleotides, and a common way to do that is with a string of characters such as "ATTACG" for a DNA sequence of 6 nucleotides. 'A' for adenine, 'C' for cytosine, 'G' for guanine, and 'T' for thymine.

Given a string representing a DNA sequence, count how many of each nucleotide is present.

For example:

```
"GATTACA" -> 'A': 3, 'C': 1, 'G': 1, 'T': 2
```

In addition, please convert the `nucleotideCount` method to a static method.

Starter Code

```
<?php

declare(strict_types=1);

class DNA {
    public function nucleotideCount(string $input): array
    {

    }
}
```

Coding Solution: Nucleotide Count

```
<?php

declare(strict_types=1);

class DNA {
    public static function nucleotideCount(string $input): array
    {
        return [
            'A' => substr_count($input, 'A'),
            'C' => substr_count($input, 'C'),
            'T' => substr_count($input, 'T'),
            'G' => substr_count($input, 'G'),
        ];
    }
}
```

Coding Exercise: Grade School

Given students' names along with the grade that they are in, create a roster for the school.

In the end, you should be able to:

- Add a student's name to the roster for a grade
  - "Add Jim to grade 2."
  - "OK."
- Get a list of all students enrolled in a grade
  - "Which students are in grade 2?"
  - "We've only got Jim just now."
- Get a sorted list of all students in all grades. Grades should sort as 1, 2, 3, etc., and students within a grade should be sorted alphabetically by name.
  - "Who all is enrolled in school right now?"
  - "Let me think. We have Anna, Barb, and Charlie in grade 1, Alex, Peter, and Zoe in grade 2 and Jim in grade 5. So the answer is: Anna, Barb, Charlie, Alex, Peter, Zoe and Jim"

Note that all our students only have one name. (It's a small town, what do you want?)

Note: Please don't use constructor property promotion, as it doesn't work in Udemy.

Starter Code

```php
<?php

declare(strict_types=1);

class School
{
    public function add(string $name, int $grade): void
    {

    }

    public function grade($grade) : array
    {

    }

    public function studentsByGradeAlphabetical(): array
    {

    }
}
```

Coding Solution: Grade School

```php
<?php

declare(strict_types=1);

class School
{
    private $students = [];

    public function add(string $name, int $grade): void
    {
        $this->students[$grade][] = $name;
    }

    public function grade($grade) : array
    {
        return $this->students[$grade] ?? [];
    }

    public function studentsByGradeAlphabetical(): array
    {
        ksort($this->students);

        return array_map(function ($grade) {
            sort($grade);

            return $grade;
        }, $this->students);
    }
}
```

Section 9: Framework Foundation

In this section, we started working on preparing the foundation of our application by structuring our project and adding namespaces.

Project Overview

In this lecture, we discussed what we'll be building for the rest of this course. Two projects will be built, a custom expense-tracking application and a framework.

What is a framework?

A framework is a set of tools and solutions for common problems. Most projects have the same requirements, from authentication to form validation. Frameworks provide a basic foundation to save you time from repeatedly writing the same solution.

Text Editors

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- Visual Studio Code
- PHPStorm

Understanding the LAMP stack

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- XAMPP
- MAMP
- WAMP
- Laragon

Exploring XAMPP

This lecture does not have any notes. The content is meant to be consumed via video.

The htdocs Folder

In this lecture, we learned about the `htdocs` folder, which is short for **hypertext documents**. It contains the files that will be served to the browser when a user navigates to `http://localhost`. Localhost refers to the current machine. It's not specific to PHP or Apache.

We created our project in the `htdocs` folder called `phpiggy`. Within this folder, we created another folder called `public`. There is no official standard project structure. PHP allows you to structure your project to how you see fit.

Typically, developers create a folder called `public` for files that should always be accessible to the public. Files outside this directory should be inaccessible via URL.

Within this folder, we created a file called `index.php` with a simple text message.

Configuring Virtual Hosts in Apache

In this lecture, we learned how to create a virtual host in Apache. Virtual hosts are different URLs that can point to a single server. By using a virtual host, we can have multiple sites using XAMPP instead of one.

A virtual host can be added by modifying Apache's configuration. You can find this file via the control panel by going to **Config > Apache (https.conf)**.

In this file, comments are written with the `#` symbol. Anything else is considered a directive. Directives are instructions for configuring the settings. Apache has a directive called `Include` for loading additional configuration files. Here's an example that loads a configuration file for virtual hosts.

```
# Virtual hosts
Include conf/extra/httpd-vhosts.conf
```

In the `httpd-vhosts.conf`, you can add a virtual host with the following configuration:

```
<VirtualHost *:80>
    # ServerAdmin webmaster@dummy-host2.example.com
    DocumentRoot "D:/xampp/htdocs/phpiggy/public"
    ServerName phpiggy.local
    # ErrorLog "logs/dummy-host2.example.com-error.log"
    # CustomLog "logs/dummy-host2.example.com-access.log" common
</VirtualHost>
```

The `<VirtualHost>` is a container for applying directives to a specific directory in your project as opposed to applying directives to the entire server.

In this example, we're setting the following directives.

- `DocumentRoot` - The directory to point to
- `ServerName` - The URL that will be associated with this directory

In addition, we must update the `hosts` file to have our machine point to the Apache web server like so.

```
127.0.0.1 phpiggy.local
```

You can refer to the resource section for info on how to find the `hosts` file. After making those changes, you can view your site by going to: http://phpiggy.local

Resources

- How to Edit Hosts File

Configuring PHP

In this lecture, we learned how to configure PHP. Firstly, you can view the current PHP configuration by using the `phpinfo()` function.

However, if you want to update the configuration settings, you can do so by updating a file called `php.ini`. In XAMPP, you can navigate to **Config > PHP (php.ini)** to view this file.

Alternatively, you can use the `ini_set()` function to update a specific setting. There are two arguments, the name of the setting and the value.

```php
ini_set("memory_limit", "255M");
```

You can view a specific setting by using the `ini_get()` function, which accepts the name of the setting.

```php
ini_get("memory_limit");
```

Resources

- php.ini Directives

Creating an Application Class

In this lecture, we created an `App` class for acting as the glue for our Frameworks tools. If developers want to use all our frameworks tools, we can use the `App` class to simplify the process so that they don't have to configure each tool independently and then connect them together. Here's what our `App` class looks like initially for testing purposes.

```php
declare(strict_types=1);

namespace Framework;

class App
{
  public function run()
  {
    echo "App is running!";
  }
}
```

Any code related to our framework will be placed inside the `Framework` namespace.

Bootstrapping an Application

In this lecture, we bootstrapped our application in a separate file so that we can load our project's configuration in any file that wanted to initialize our app. We have a file called `bootstrap.php` with the following code:

```php
declare(strict_types=1);

include __DIR__ . "/../Framework/App.php";

use Framework\App;

$app = new App();

return $app;
```

From this file, we're returning the instance. We won't be calling the `run()` method, which is responsible for starting the application after it has been configured.

We're executing this method from the `index.php` file, which is responsible for displaying the page's contents in the browser.

```php
$app = require __DIR__ . '/../src/App/bootstrap.php';

$app->run();
```

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.

Understanding PSR

In this lecture, we talked about PSR, which stands for PHP Standards Recommendations. PHP allows developers to format and structure their code any way they'd like. Over the years, developers have created a set of standards known as PSR. There are various standards depending on what you're trying to accomplish. The larger your project, the more standards you're likely to add to your project.

The PHP community has a standard called PSR-4 for setting a standard for autoloading files. We'll be implementing this standard with the help of Composer.

Installing Composer

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- Packagist
- Composer

JSON Crash Course

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 **J**ava**S**cript **O**bject **N**otation. 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.

```json
{
  "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

Initializing Composer

In this lecture, we learned how to initialize composer in our project. Before we can use Composer, we must have a configuration file. Firstly, we can view a complete list of commands available in composer by running the `composer` command. You can view a list of options available for a command by adding the `--help` flag option like so.

```
composer init --help
```

A configuration file can be created manually, but it's recommended to use the `composer init` command to prevent typos. The following questions may be asked:

```
Package name (<vendor>/<name>): example/phpiggy
Description []: A PHP application for tracking expenses
Author: Your name
Minimum Stability []:
License []:
Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]? no
Autoload? no
Do you confirm generation [yes]? yes
```

After a few moments, a `composer.json` file should be available in your project.

Resources

- composer.json Schema

Generating Autoload Files

In this lecture, we generated the files to help us autoload our classes in our project. First, we must configure Composer to the namespaces and directories where the class files can be found. We must add the `autoload` object to configure these settings. Next, we must add the `psr-4` object. There are different standards available for autoloading files, which are supported by Composer. Therefore, we must specify the standard we'd like to use.

```
{
  "autoload": {
    "psr-4": {
      "Framework\\": "src/Framework",
      "App\\": "src/App/"
    }
  }
}
```

Within this object, we can add a list of properties. The property name must contain the namespace. The value must contain the directory where the namespace is located.

Lastly, we can generate the files by running the following command:

```
composer dump-autoload
```

After running this command, a new directory called `vendor` becomes available. This is where Composer stores its generated files. If you look inside the `vendor/composer/autoloader-psr4.php` file, you'll find an array of our namespaces, which tells Composer where to find our class files with these namespaces.

Including Autoload Files

In this lecture, instead of directly including the `App.php` file, we included the `autoload.php` file from the `vendor` directory like so from the `bootstrap.php` file.

```
require __DIR__ . '/../../vendor/autoload.php';
```

Now, if we use any of our classes from the `App` or `Framework` namespaces, they'll automatically be included in our project.

What is Git?

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 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.

Using GitHub

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- GitHub Desktop
- Git and GitHub Tutorial

Exploring Git Files

In this lecture, we explored the Git files in our project. The first file is the `.gitattributes` file. This file contains a list of files and settings to be applied to those files when they're committed to a repository.

```
# Auto detect text files and perform LF normalization
* text=auto
```

This file configures the new lines in our code. Characters, that are invisible to us, exist in our file for creating new lines. There are different characters available for creating new lines, which are LF and CRLF. This configuration sets the default character to LF since it's supported across multiple operating systems and editors.

Up next, we have the `.gitignore` file. This file contains a list of files that shouldn't be committed to your repo. It's recommended to not include the `vendor` folder since the dependencies can be reinstalled with the `composer install` command. It also keeps your project file size small.

```
composer.phar
/vendor/

# Commit your application's lock file https://getcomposer.org/doc/01-basic-usage.md#commit-your-composer-lock-file-to-version-cont
# You may choose to ignore a library lock file http://getcomposer.org/doc/02-libraries.md#lock-file
# composer.lock
```

Resources

- Git Attributes
- PHPiggy GitHub Repository

Section 10: Routing

In this section, we worked on the routing feature for our application.

Understanding Routing

This lecture does not have any notes. The content is meant to be consumed via video.

Apache Mod Rewrite

In this lecture, we learned how to intercept all requests so that the `index.php` file handles them by modifying the `httpd.conf` file. In this file, we added the following:

```
<Directory "D:/xampp/htdocs/phpiggy/public">
    RewriteEngine On

    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f

    RewriteRule ^ /index.php [L]
</IfModule>
```

Firstly, we're using the `Directory` to apply these settings to our project's directory. Next, we're enabling the module by using `RewriteEngine On`. Afterward, we check if the request is for a specific file or directory using the `RewriteCond` directive. Apache creates a variable called `%{REQUEST_FILENAME}` that contains the file being requested.

Lastly, if both conditions pass, we're using the `RewriteRule` directive to accept any request and let the `index.php` file handle it instead of allowing Apache to send back the file. The `[L]` flag prevents any other additional rewrite rules from applying to our request as an extra precaution.

In the `index.php` file, we printed the superglobal variable `$_SERVER`. This variable contains information on the server, including the requested URI under a key called `request_uri`.

```php
echo "<pre>";
print_r($_SERVER);
echo "</pre>";
```

Resources

- Apache Module mod_rewrite
- Server Superglobal Variable

The htaccess File

In this lecture, we learned how to use the `.htaccess` file to modify Apache's configuration. Unlike the main configuration file, the `.htaccess` file can be created in a specific directory, and those settings would only be applied to that directory.

Here are some additional differences:

- The main configuration supports all directives.
- The `.htaccess` partially supports directives.
- The main configuration file can be applied to the entire server or a specific directory.
- The `.htaccess` file only applies to the current directory.
- The main configuration file runs once when server is started.
- The `.htaccess` file runs on every request.

Behind the scenes, XAMPP is configured to reject access to the `.htaccess` file is accessed directly. If you're not using XAMPP, you can use the following configuration to prevent access.

```
#
# The following lines prevent .htaccess and .htpasswd files from being
# viewed by Web clients.
#
<Files ".ht*">
    Require all denied
</Files>
```

If you don't plan on using the `.htaccess` file, you should disable it by using the `AllowOverride` directive.

```
AllowOverride none
```

Resources

- htaccess File

Sugar Functions

In this lecture, we talked about sugar functions, which are just functions to act as a shortcut for specific actions, such as outputting a variable's contents with `<pre>` tags. Here's an example of a function we defined for dumping a variable's contents.

```php
function dd($value)
{
  echo '<pre>';
  var_dump($value);
  echo '</pre>';
```

```
  die();
}
```

Creating a Router Class

In this lecture, we prepared `Router` class in our framework. After doing so, we instantiated this class from the `App` class. The instance gets stored in a property like so:

```
private Router $router;

public function __construct()
{
  $this->router = new Router();
}
```

The property is private to prevent the instance from being overriden.

Adding Routes

In this lecture, we updated our `Router` class to store routes. A route refers to the path to visit a page. It's the portion of the URL after the domain. In our class, we have a property called `$routes` to store the routes. Next, we defined a method called `add()` to register new routes. New routes are associative arrays with the path.

```
class Router
{
  private array $routes = [];

  public function add(string $path)
  {
    $this->routes[] = [
      'path' => $path
    ];
  }
}
```

In addition, we had to update the `App` class to have the same method since the `$router` property is private.

```
public function add(string $path)
{
  $this->router->add($path);
}
```

Understanding HTTP Methods

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- HTTP Methods

Supporting HTTP Methods in Routes

In this lecture, we decided to start supporting HTTP methods in our routes. This allows us to handle the same route with different actions. In the `Router::add()` method, we update the array to the following:

```
$this->routes[] = [
  'path' => $path,
  'method' => strtoupper($method)
];
```

Before inserting the `$method` parameter, we're passing it through the `strtoupper()` function. This function is defined by PHP. It'll transform the characters in a string into all uppercase letters.

Normalizing Paths

In this lecture, we talked about normalizing paths. During route registration, developer may format their paths differently. For example, let's say we a route for a list of team members. Possible paths could include:

- `about/team`
- `/about/team`

- `/about/team/`

Inconsistencies can lead to issues. Normalizing is the process of standardizing a value by having consistent formatting. We decided to define a method to normalize our paths. All paths will begin and end with a `/`.

In this method, we're accepting the `$path` and returning the formatted version. First, we're removing `/` character from the beginning and end of a string with the `trim()` function. Next, we appended `/` characters to the path before returning it.

```
private function normalizePath(string $path): string
{
  $path = trim($path, '/');
  $path = "/{$path}/";

  return $path;
}
```

In the `add()` method, we updated the `$path` variable before updating the `$routes` array.

```
$path = $this->normalizePath($path);
```

Exploring Regular Expressions

In this lecture, we learned about regular expressions. Regular expressions are patterns that we can run against strings to check for matches. PHP provides basic functions for string manipulation, but they may not always suffice. For this reason, regular expressions were introduced for more complex matches.

We used a regular expression to help us normalize a path by removing consecutive `/` characters.

`[/]{2,}`

In this expression, we're checking for the `/` character. A list of characters can be written inside the `[]`. The `{2,}` portion of the expression allows to check if they are two or more `/` characters. Otherwise, we can leave the `/` character alone before replacing it.

Resources

- String Functions
- Regex101

Regular Expressions in PHP

In this lecture, we decided to use the regular expression from the previous lecture. We used it to replace multiple `/` with a single `/` character. To perform this action, we used the `preg_replace()` function. It accepts the regular expression, the replacement value, and the string to search.

```
$path = preg_replace('#[/]{2,}#', '/', $path);
```

Expressions must start and end with the `#` character to denote the beginning and end of the expression. The return value of this function will be the string with the replaced values.

Resources

- PCRE Functions

MVC Design Pattern

This lecture does not have any notes. The content is meant to be consumed via video.

Creating a Controller

In this lecture, we created a controller, which is a separate file with a class definition responsible for loading a specific page. We created the following directories: `src/App/Controllers`. Inside this newly created directory, we created a file called `HomeController.php` with the following code:

```
declare(strict_types=1);

namespace App\Controllers;

class HomeController
{
  public function home()
  {
```

```
    echo 'Homepage';
  }
}
```

In this example, we're adding the `App\Controllers` namespace to contain the `HomeController` class. Most of the logic should be familiar to you as all we're doing is creating a method called `home` for rendering the contents of the home page.

Registering Controllers

After creating this controller, we updated the `bootstrap.php` file to use this controller when the user visits the `/` homepage.

```
$app->get('/', ['\App\Controllers\HomeController', 'home']);
```

In this example, we're passing in the class and method names as strings. It's more efficient to allow the router to instantiate the class instead of passing in a direct instance. This is more efficient as you won't need an instance for every controller unless the user is visiting the corresponding route.

Next, we updated our `App` class to accept this information. All we're doing is passing on the controller to the router to store it with the route.

```
public function get(string $path, array $controller)
{
  $this->router->add('GET', $path, $controller);
}
```

Here's how the router stores the controller.

```
public function add(string $method, string $path, array $controller)
{
  $path = $this->normalizePath($path);

  $this->routes[] = [
    'path' => $path,
    'method' => strtoupper($method),
    'controller' => $controller
  ];
}
```

Class Magic Constant

In this lecture, we used the `class` constant, which is available in all classes and defined by PHP on your behalf. This constant contains the full namespace of a class. Instead of writing the full path yourself, you can use this constant. We updated the `get()` method by passing in an array, which contains the `HomeController::class` constant and the name of the method to call.

```
use App\Controllers\HomeController;

$app->get('/', [HomeController::class, 'home']);
```

Dispatching a Route

In this lecture, we dispatched a route. Dispatching describes the process of sending something off to a location. In our case, we're sending content to the browser based on the route. We defined a method called `dispatch()`, which accepts a path and method. In the method, we're formatting the path and method to be consistent with the values in our routes.

```
public function dispatch(string $path, string $method)
{
  $path = $this->normalizePath($path);
  $method = strtoupper($method);

  echo "{$path} {$method}"
}
```

Lastly, in the `run()` method of the `App` class, we called the `dispatch()` method.

```
public function run()
{
  $path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
  $method = $_SERVER['REQUEST_METHOD'];

  $this->router->dispatch($path, $method);
}
```

We're grabbing the path from the `$_SERVER` superglobal variable. However, this variable stores the full URL. We're only interested in the path. To grab the path only, we passed the value into the `parse_url()` function. This function extracts information on the URL. Since we're only interested in the path, we passed in the `PHP_URL_PATH` constant to instruct the function to only return the path.

As for the method, we just simplified the name to `$name` instead of `$_SERVER['REQUEST_METHOD']`. Lastly, we called the `dispatch()` method with these variables.

Finding Matches with Regular Expressions

In this lecture, we started to loop through the routes in the `$routes` property of the `Router` class. On each iteration, we checked if the path from the current route matched the path passed into the `dispatch()` method. In addition, we checked if the methods match. If either of these conditions doesn't return a match, we should not proceed to render the content.

```
foreach ($this->routes as $route) {
  if (!preg_match("#^{$route['path']}$#", $path) || $route['method'] !== $method) {
    continue;
  }

  echo 'route found!';
}
```

In the example above, we're using a regular expression. We're performing an exact match with this expression. The `^` character in the expression makes sure the path begins with the value. The `$` character in the expression makes sure the path ends with the value.

Instantiating Classes with Strings

In this lecture, we instantiated the controller class if we found a match in the router. First, we destructured the class name and method name from the route.

```
[$class, $func] = $route['controller'];
```

Afterward, we created an instance of the class with the `new` keyword. It's completely acceptable to provide a string containing the class name for this keyword. PHP is more than capable of resolving the class. The instance gets stored in a variable called `$controllerInstance`.

```
$controllerInstance = new $class;
```

Lastly, we called the method specified in our route. It's perfectly acceptable to use a variable containing the method name after the `->` operator. PHP will resolve the method name.

```
$controllerInstance->{$func}();
```

PSR-12 Auto Formatting

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- PSR-12

Section 11: Template Engines

In this section, we worked on adding a template engine to our framework for rendering HTML content with dynamic values.

Understanding Template Engines

In this lecture, we learned about why template engines are useful. You can use pure PHP to create templates, but there are benefits to using a template engine.

1. Easier for frontend developers to understand your templates since the syntax is easier to read than pure PHP.
2. Template engines offer a layer of security to prevent malicious data from being outputted onto the page.

Twig by Symfony and Blade by Laravel is the most popular libraries for adding a template engine to your app. For our app, we decided to create a custom template engine.

Creating a Template Engine Class

There are no notes for this lecture.

Setting a Base Path

In this lecture, we updated our `TemplateEngine` class to store a path to a directory in our project where templates can be found. Not all projects will store templates in the same directory. Therefore this information should be configurable. In the `TemplateEngine` class, we added the `__construct()` method and added a property called `$basePath`.

```
public function __construct(private string $basePath)
{
}
```

Next, we defined a class to store paths to various areas in our app called `Paths`.

```
namespace App\Config;

class Paths
{
  public const VIEWS = __DIR__ . "/../views";
  // public const ROOT = __DIR__ . "/../../../";
  // public const STORAGE_CACHE = __DIR__ . "/../../../storage/cache";
  // public const ENTITY = __DIR__ . "/../Entity";
  // public const SOURCE = __DIR__ . "/../../";
  // public const STORAGE_UPLOADS = __DIR__ . "/../../../storage/uploads";
}
```

Lastly, we used the `VIEWS` constant like so.

```
$this->view = new TemplateEngine(Paths::VIEWS);
```

Rendering a Template

In this lecture, we rendered the template using the `TemplateEngine` class from our controller. First, we defined a method called `render()` method, which accepts the template name. Inside this method, we included the template by using the `include` keyword. The path to a template is the `$basePath` property with the `$template` parameter.

```
public function render(string $template)
{
  include "{$this->basePath}/{$template}";
}
```

We can grab a template like so.

```
$this->view->render("index.php");
```

Extracting Arrays

In this lecture, we updated our engine to display data. First, we updated the `render` method's signature by accepting an array of data.

```
public function render(string $template, array $data = [])
```

By default, the method's parameters are accessible to templates. However, they must be accessed through the `$data` variable. For convenience, you may want to extract each item in the array into a separate variable. You can do so with the `extract` method. This method accepts the associative array to extract and how it should handle existing variables.

```
extract($data, EXTR_SKIP);
```

In this example, we're extracting the `$data` variable and using the `EXTR_SKIP` constant to tell PHP not to overwrite existing variables.

Resources

- extract() function

Understanding Output Buffering

In this lecture, we used output buffers so that we could 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 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. 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

Creating an Output Buffer

In this lecture, we created an output buffer for our templates. We can enable this behavior by calling the `ob_start()` function like so:

```
ob_start();
```

In our `render()` method, we used output buffers to store the HTML without it being sent to the browser.

```
public function render(string $template, array $data = [])
{
  extract($data, EXTR_SKIP);

  ob_start();
  include $this->resolve($template);
  $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, the content may get sent too early.

Lastly, we returned the `$content` variable. By using output buffers, we can perform additional actions on the template after it has been retrieved.

Loading Assets

In this lecture, we loaded our site's CSS by creating a file in the `public/assets` directory called `main.css`. You can refer to the resource section of this lecture for the CSS. It's important that the CSS is added to the `public` directory. Otherwise, you won't be able to access the file through HTTP.

Resources

- HTML and CSS Course
- Homepage Template

Adding Partials

In this lecture, we learned how to split our template into separate files by using partials. Partials are not entire templates. They're meant to contain portions of a larger template like headers, footers, and sidebars.

It's common practice to store partials in a folder called `partials`. Not required but recommended. Next, a common naming convention for partial template files is to prefix the name with an `_` character. For example, if we had a partial for the header, the filename would be `_header.php`.

In our engine, we decided to create a method to help us easily generate an absolute path from our templates called `resolve()`. It accepts the path and returns the path with the base path from the instance.

```
public function resolve(string $path)
{
  return "{$this->basePath}/{$path}";
}
```

After creating a partial, we can load the partial with the `include` tag like so.

```
<?php include $this->resolve("partials/_header.php"); ?>
```

Exercise: Creating an About Page

There are no notes for this lecture.

Resources

- About Template

Autoloading Functions

In this lecture, we learned how to autoload functions. Functions can be imported into a file by using `use function`. Here's an example of how we imported functions for registering routes.

```
use function App\Config\registerRoutes;
```

Since there is no standard for auto importing functions, we must manually instruct Composer to autoload the function by adding an array called `files` to the `autoload` option in the `composer.json` file. This array must contain a list of files to load.

```
{
  "autoload": {
    "psr-4": {
      "Framework\\": "src/Framework",
      "App\\": "src/App/"
    },
    "files": ["src/App/Config/Routes.php"]
  }
}
```

Lastly, you must update your autoload files using the `composer dump-autoload` command.

Section 12: Containers and Dependency Injection

In this section, we designed our own container for dependency injection in our application.

Understanding Dependency Injection

In this lecture, we learned about dependency injection. Dependency injection is a design pattern where a container manages objects and can provide instances to any class that needs them.

Creating a Container

In this lecture, we created a container, which is an object that contains instructions for creating instances of our classes. Here's what our class looks like.

```
class Container
{
}
```

Next, we instantiated this class from the `App` class.

```
private Container $container;

public function __construct()
{
```

```
    $this->container = new Container();
}
```

External Definitions File

In this lecture, we learned how to outsource our definitions into a separate file. We created a file called `container-definitions.php` with the following code:

```
declare(strict_types=1);

return [
];
```

This file returns an array. Definitions refer to the list of dependencies available in our application. Next, we added a method called `addDefinitions()` in the `Container` class to accept the array of definitions.

```
public function addDefinitions(array $newDefinitions)
{
    dd($newDefinitions);
}
```

Afterward, we updated the `App` class `__construct` method to accept a path to the file with the container definitions. In this example, we're checking if a path was provided. If it was, we'll proceed to include the file and pass on the array to the container through the `addDefinitions` method.

```
public function __construct(string $containerDefinitionsPath = null)
{
    $this->router = new Router();
    $this->container = new Container();

    if ($containerDefinitionsPath) {
        $containerDefinitions = include $containerDefinitionsPath;
        $this->container->addDefinitions($containerDefinitions);
    }
}
```

Lastly, we updated our `Paths` class to contain the path to the `src` directory.

```
class Paths
{
    public const VIEWS = __DIR__ . "/../views";
    public const SOURCE = __DIR__ . "/../../";
}
```

Factory Design Pattern

In this lecture, we created a factory function for creating an instance of our own class. We can add instructions for creating this class in our container definitions. Inside the definitions file, we added the following:

```
use Framework\TemplateEngine;
use App\Config\Paths;

return [
    TemplateEngine::class => fn () => new TemplateEngine(Paths::VIEWS)
];
```

It's common practice to use the class name as the name for our definition. The value is a function that returns the instance, but why a function? The factory design pattern is a pattern where a function handles the process of creating an instance. The main benefit of this pattern is that we don't have to create an instance immediately. We can wait until the instance is needed before creating it, thus saving memory.

Merging Arrays

In this lecture, we learned how to merge two arrays. We're merging the definitions stored in the container with the definitions passed into the `addDefinitions` method like so.

```
 public function addDefinitions(array $newDefinitions)
{
    $this->definitions = array_merge($this->definitions, $newDefinitions);
}
```

The `array_merge()` function accepts an unlimited number of arrays and returns a merged array. An alternative syntax is to use the `...` operator like so.

```
$this->definitions = [...$this->definitions, ...$newDefinitions];
```

The main difference is that the spread operator is faster than the `array_merge()` function, but it is less readable. Other than that, the behavior is the exact same.

Reflective Programming

In this lecture, we learned about reflective programming. Most programming languages allow you to inspect the code of your application. This is referred to as reflection. PHP supports reflection. We're using the reflection API so that we can inspect our classes for dependencies.

A class can be inspected by creating a new instance of the `ReflectClass` class. This class accepts the name of the class to inspect.

```
public function resolve(string $classname)
{
  $reflectionClass = new ReflectionClass($classname);

  dd($reflectionClass);
}
```

Validating Classes

In this lecture, we validated the class given to the `ReflectionClass`. Sometimes, a container may receive an abstract class, which cannot be instantiated. In this case, we should check that a class can be instantiated. If it can't, we should throw a custom `ContainerException` like so.

```
if (!$reflectionClass->isInstantiable()) {
  throw new ContainerException("Class {$classname} is not instantiable");
}
```

The `isInstantiable()` method will tell us if the class can be instantiated.

Resources

- ReflectionClass

Validating the Constructor Method

In this lecture, we validated the `__construct()` method of our classes. In some cases, a class may not have this method defined. If not, there isn't a point in checking for dependencies since this method contains the list of dependencies. First, we updated the `__construct()` method in the `HomeController` class by moving the `$view` property to the parameter list like so.

```
public function __construct(private TemplateEngine $view)
{
}
```

Next, we grabbed the `construct` method with the `getConstructor()` method. This method returns `null` if the method doesn't exist. We can proceed to use a conditional statement to check for a value. If it's `null`, the instance of the class is returned since we don't have to check for dependencies.

```
$constructor = $reflectionClass->getConstructor();

if (!$constructor) {
  return new $classname;
}
```

Retrieving the Constructor Parameters

In this lecture, we grabbed a list of parameters from the `__construct()` method. The `__construct()` method contains a list of dependencies for a class. To get this information, we can use the `getParameters()` method from an instance of the `ReflectionMethod` class. This method returns an array of parameters.

If there aren't any parameters, the array will be empty. We used the `count()` function to count the items in the array. If there aren't any items, we don't have to check for dependencies. We can proceed to instantiate the class and return the instance.

```
$parameters = $constructor->getParameters();
```

```
if (count($parameters) === 0) {
  return new $classname;
}
```

Resources

- ReflectionMethod

Validating Parameters

In this lecture, we validated the parameters in the `__construct()` method of a given controller. First, we looped through them.

```
foreach ($params as $param) {

};
```

Next, we grabbed the name of the data type of the parameter. The name will be necessary for error messages in case the parameters don't pass validation. The type will be important for checking if the data type is built in or exists.

```
$name = $param->getName();
$type = $param->getType();
```

The first validation we performed was checking if a type hint was given to a parameter. If not, then we threw an error.

```
if (!$type) {
  throw new ContainerException(
    "Failed to resolve class {$classname} because param {$name} is missing a type hint"
  );
}
```

Lastly, we checked if the type is an instance of the `ReflectionNamedType` class. This class is given to a parameter when a type hint is given to a class and is not a union type or intersection type. Lastly, we called the `isBuiltIn()` method to verify we're not given a primitive type like `string` or `bool`. If either of these conditions fails, we won't be able to instantiate a dependency for our controller.

```
if (!$type instanceof ReflectionNamedType || $type->isBuiltin()) {
  throw new ContainerException(
    "Failed to resolve class {$classname} because invalid param {$name}"
  );
}
```

Invoking Factory Functions

In this lecture, we invoked the factory function for a given dependency. We decided to outsource this logic to a method called `get`. In this method, we're accepting the ID of the dependency to figure out what factory function to call from our array of definitions.

```
public function get(string $id)
{
  if (!array_key_exists($id, $this->definitions)) {
    throw new ContainerException("Class {$id} does not exist in container");
  }

  $factory = $this->definitions[$id];
  $dependency = $factory(); // $factory($this);

  return $dependency;
}
```

In this method, we validate the ID by checking if it exists within the array. If it doesn't, we'll throw an exception.

Afterward, we grab the factory function, invoke it, and return the result.

Lastly, we can store the instance in the `$dependencies` array like so.

```
$dependencies[] = $this->get($type->getName());
```

Instantiating a Class with Dependencies

In this lecture, we instantiated the controller with the dependencies. The reflection API has a simple method called `newInstanceArgs` to help us create an instance. It accepts an array of arguments, which are passed on to the respective class's `__construct()` method.

```
return $reflectionClass->newInstanceArgs($dependencies);
```

Understanding Middleware

There are no notes for this lecture.

Supporting Router Middleware

In this lecture, we added support for middleware from our router. Since routers are responsible for rendering controllers, it's the appropriate place for adding middleware.

First, we updated the `Router` class to store an array of middleware.

```
private array $middlewares = [];
```

Next, we defined a method in our class to register new middleware.

```
public function addMiddleware(string $middleware)
{
  $this->middlewares[] = $middleware;
}
```

The `App` class was also updated to add middleware too.

```
public function addMiddleware(string $middleware)
{
  $this->router->addMiddleware($middleware);
}
```

Adding Middleware

There are no notes for this lecture.

Creating Middleware

In this lecture, we created our first middleware called `TemplateDataMiddleware`. It's common practice to add the word "Middleware" to the class to help other developers identify middleware. For middleware, we created a dedicated namespace called `App\Middleware`.

Lastly, we registered this class through the `Middleware` file like so.

```
$app->addMiddleware(TemplateDataMiddleware::class);
```

We're passing in the class name since we want to be able to resolve dependencies. It's better to let the router resolve dependencies instead of instantiating the class ourselves.

Interface Contracts

In this lecture, we talked about contracts. Interfaces are sometimes referred to as contracts since they force a class to have a specific implementation. We decided to define a contract for middleware for consistency. It's called `MiddlewareInterface`.

```
namespace Framework\Contracts;

interface MiddlewareInterface
{
  public function process(callable $next);
}
```

Next, we implemented this interface on the `TemplateDataMiddleware` class.

```
use Framework\Contracts\MiddlewareInterface;

class TemplateDataMiddleware implements MiddlewareInterface
{
  public function process(callable $next)
  {
  }
}
```

Chaining Callback Functions

In this lecture, we learned how to chain multiple functions. Imagine we had the following code.

```
$action = function () {
  echo "Main Content <br>";
};

$action();
```

In some cases, we may have an array of functions we'd like to execute before executing the main function. Here's an example of an array of functions.

```
$funcs = [
  function ($next) {
    echo "a <br>";
    $next();
  },
  function ($next) {
    echo "b <br>";
    $next();
  },
  function ($next) {
    echo "c <br>";
    $next();
  }
];
```

In each function, we're going to accept the next function to run. This allows us to run code before and after the next function.

We can loop through the array of functions like so.

```
foreach($funcs as $func) {
  $action = fn () => $func($action);
}
```

We're overriding the previous function but passing it onto the next function so that it isn't lost forever. By doing so, we'll still be able to invoke the previous function. This is how middleware is implemented in most cases. In the next lecture, we'll apply the same logic to our framework.

Looping through Middleware

In this lecture, we refactored our `Router` class to use middleware by looping through them and creating nested callback functions. Firstly, we placed the controller inside a function.

```
$action = fn () => $controllerInstance->$func(); // $controllerInstance->$func($params);
```

Next, we looped through the `$middlewares` property, instantiated the middleware, and then overridden the `$action` variable with the middleware while passing on the previous function to the current action.

```
foreach ($this->middlewares as $middleware) {
  $middlewareInstance = new $middleware;
  $action = fn () => $middlewareInstance->process($action);
}
```

Lastly, we invoked the `$action` function, which starts the chain of functions.

```
$action();
```

Supporting Dependency Injection in Middleware

In this lecture, we supported dependency injection middleware by checking if the container exists. If it does, we resolved the dependencies for that middleware. Otherwise, we just went ahead with creating an instance.

```
$middlewareInstance = $container ?
  $container->resolve($middleware) :
  new $middleware;
```

Global Template Variables

In this lecture, we updated the `TemplateEngine` class to support global data. First, we defined a property for storing the global data.

```
private array $globalTemplateData = [];
```

Next, we extracted this property inside the `render()` method. It's important to extract the data after the data from the controller. Otherwise, global data may get prioritized over the component's data.

```
extract($data, EXTR_SKIP);
extract($this->globalTemplateData, EXTR_SKIP);
```

Lastly, we defined a method called `addGlobal()` for inserting new data into our engine.

```
public function addGlobal(string $key, mixed $value)
{
  $this->globalTemplateData[$key] = $value;
}
```

The `TemplateDataMiddleware` class was updated to call this method to insert a default title in case a controller does not provide one.

```
class TemplateDataMiddleware implements MiddlewareInterface
{
  public function __construct(private TemplateEngine $view)
  {
  }

  public function process(callable $next)
  {
    $this->view->addGlobal("title", "Expense tracking app");

    $next();
  }
}
```

Singleton Pattern

In this lecture, we talked about the issue with our middleware. At the moment, our template does not use the global data from the middleware. This is because our middleware and controllers receive unique instances of the global data. Therefore, they won't have access to the same data.

To solve this issue, we added the singleton pattern to the container. If an instance exists for a dependency, we'll return the existing instance instead of creating a new one.

In the `Container` class, we created a property for storing existing instances called `$resolved`.

```
private array $resolved = [];
```

Next, we stored the instance of a dependency in the array after it had been created from the `get()` method.

```
$this->resolved[$id] = $dependency;
```

Lastly, we checked if an instance exists by using the `array_key_exists()` function. If so, we'll return the instance. This process was performed from within the `get()` method before an instance of the dependency was created. It's important to return the instance. Otherwise, the instance will still get created.

```
if (array_key_exists($id, $this->resolved)) {
  return $this->resolved[$id];
}
```

Resources

- Singleton Pattern

Section 13: Form Validation

In this section, we learned how to properly validate forms by creating a custom validator and set of rules for the most common scenarios while handling errors.

Preparing the Registration Form

There are no notes for this lecture.

Resources

- Register Template

Configuring the Form

In this lecture, we took the time to prepare the form to submit data to the correct URL. A URL and HTTP method can be configured on a `<form>` element by adding the `action` and `method` attributes.

```
<form action="/some-url" method="POST"></form>
```

The `action` property can point to a specific URL, whether it's absolute or relative. This attribute is optional. If not supplied, the form gets submitted to the same page. The `method` attribute allows us to set an HTTP method. Only `GET` and `POST` are supported.

Next, form data is only sent if the input elements have the `name` attribute like so:

```
<input name="example" />
```

Resources

- Sending Form Data

Handling POST Data

In this lecture, we accepted the form data by defining a controller and registering a route. We can use the superglobal variable `$_POST` to grab post data, which is native to PHP like so.

```
var_dump($_POST);
```

Understanding Services

In this lecture, we talked about how services can keep our controllers thin. It's considered good practice for controllers to have as little logic as possible. They should be responsible for receiving requests and returning a response. The job of a service is to handle other logic from validation to processing transactions.

We created a class called `ValidatorService`, registered it with our container, and injected it into the `AuthController` class.

Creating a Validator Class

In this lecture, we created a `Validator` class in our framework, which might seem strange since we have a service in our application. The `Validator` class will contain logic for how validation should be performed, whereas the `ValidatorService` class will contain logic for what should be validated.

Typically, classes in your framework should know the "how" whereas classes in your application should know the "what." This is a general guideline that you won't always be able to follow, but it is a good place to start when deciding where to place logic in your project.

Validation Rule Contract

In this lecture, we created a contract since rules can come from the framework or application. It's not possible to cover every possible validation scenario, so it's considered good practice to allow other developers to extend the current set of validation rules with custom rules. An interface can be a useful way of enforcing a standard for rules.

```
interface RuleInterface
{
  public function validate(array $data, string $field, array $params);
  public function getMessage(array $data, string $field, array $params);
}
```

In this example, we have an interface that requires two methods. The `validate()` method will be responsible for validating the value. The `getMessage()` method will be responsible for generating an error message when validation fails. Both methods will accept the entire form data, the field to validate, and an extra set of parameters.

Lastly, we defined a method in the `Validator` class to add new rules.

```
private array $rules = [];

public function add(string $alias, RuleInterface $rule)
{
  $this->rules[$alias] = $rule;
}
```

Registering a Rule

In this lecture, we created a class requiring a field to have a value. In the `validate()` method, we're using the `empty()` function to check if the field has a value. If it doesn't, validation fails. In the `getMessage()` method, we're returning a generic error message.

```
use Framework\Contracts\RuleInterface;

class RequiredRule implements RuleInterface
{
  public function validate(array $data, string $field, array $params): bool
  {
    return !empty($data[$field]);
  }

  public function getMessage(array $data, string $field, array $params): string
  {
    return "This field is required";
  }
}
```

Lastly, we registered the rule with the `ValidatorService` class.

```
$this->validator->add('required', new RequiredRule());
```

Applying Rules to Fields

In this lecture, we updated the `validateRegister()` method to apply the `RequiredRule` to all fields. In the `validate()` method, we passed in an array of fields. The key name represents the field to validate, and the value is an array of rules to apply to the field.

```
$this->validator->validate($formData, [
  'email' => ['required']
  'age' => ['required'],
  'country' => ['required'],
  'socialMediaURL' => ['required'],
  'password' => ['required'],
  'confirmPassword' => ['required'],
  'tos' => ['required']
]);
```

Next, we updated the `Validator` class to loop through this array of fields. From within the first loop, we're performing another loop to iterate through the rules applied to a field. The array of rules is aliases. We're grabbing the instance of our rule with the alias. After doing so, we called the rule's respective `validate()` method.

If an error doesn't get produced, the class moves on to the next rule.

```
foreach ($fields as $fieldName => $rules) {
  foreach ($rules as $rule) {
    $ruleValidator = $this->rules[$rule];

    if ($ruleValidator->validate($data, $fieldName, [])) {
      continue;
    }

    echo "error";
  }
}
```

Storing Validation Errors

In this lecture, we stored the validation errors if a rule fails to validate a field. In the loop, we updated an array called `$errors`. Since we want to render errors below specific fields, we're also storing the fieldname as the key name. We're calling the `getMessage()` method to get the error message for a related validation rule.

```
$errors[$fieldName][] = $ruleValidator->getMessage(
  $data,
  $fieldName,
```

```
    []
);
```

Next, we counted the errors with the `count()` function.

```
if (count($errors)) {
  dd($errors);
}
```

Custom Validation Exception

In this lecture, we created a custom exception so that it's easier to catch errors related to invalid form submissions. We're extending the `RuntimeException` class for better categorization. Runtime errors are errors that get produced while the application is running. They're not errors that need to be fixed but handled.

```
use RuntimeException;

class ValidationException extends RuntimeException
{
}
```

Resources

- RuntimeException Class

HTTP Status Codes

In this lecture, we learned how to apply HTTP status codes to our errors. HTTP status codes can be thought of as a way to categorize the response. There are various status codes. Generally, they're categorized as the following:

- 1xx - Information
- 2xx - Success
- 3xx - Redirection
- 4xx - Client error
- 5xx - Server error

We decided to add this code to our exception class by updating the `$code` property through the `__construct()` method like so:

```
public function __construct(
    int $code = 422
  ) {
    parent::__construct(code: $code);
  }
```

We're calling the parent constructor method so that the properties are applied to the class.

Resources

- HTTP Response Status Codes

Custom Middleware

In this lecture, we created custom middleware. First, we must import the appropriate interfaces to help us build the middleware.

```
use Framework\Contracts\MiddlewareInterface;
use Framework\Exceptions\ValidationException;
```

Next, we must implement the `MiddlewareInterface` in the class.

```
class ValidationExceptionMiddleware implements MiddlewareInterface
{
  public function process(callable $next)
  {
    $next();
  }
}
```

This interface forces our class to implement a method called `process()`. This method calls the `$next()` function to pass on the request to the next middleware.

Lastly, we must register our middleware by calling the `addMiddleware()` method on the app instance. This method accepts an instance of our middleware as an argument.

```
$app->addMiddleware(ValidationExceptionMiddleware::class);
```

After registering the middleware, it'll always run for every page.

Redirection with Headers

In this lecture, we decided to redirect the user if an error gets thrown. First, we defined a function called `redirectTo()` that accepts the path to redirect the user to.

```
function redirectTo(string $path)
{
  header("Location: {$path}");
  http_response_code(302);
  exit;
}
```

Next, called the `header()` method allows us to add a header to the response. In this example, we're adding the `Location` header to change the URL to redirect the user. Lastly, we're setting the status code to `302` to inform the browser the response should redirect the user.

```
try {
  $next();
} catch (ValidationException $e) {
  redirectTo("Location: /register");
}
```

Resources

- Headers

Passing on the Errors

In this lecture, we learned how to pass on the errors to our exception. First, we passed on the errors to the `ValidationException` class.

```
throw new ValidationException($errors);
```

Lastly, we can accept the errors as an argument to our `__construct()` method and use constructor property promotion to store the errors as a property to the class.

```
public function __construct(
  public array $errors,
  int $code = 422
) {
  parent::__construct($message, $code);
}
```

HTTP Referrer

In this lecture, we updated our middleware to redirect the user to the page with the form if their was an error with validating their input. First, we must grab an array of server information by using the `$_SERVER` variable.

One of the items in the array is called `HTTP_REFERER`. This item contains the URL that initiated the request. In our case, it would be the form.

```
$referer = $_SERVER['HTTP_REFERER'];
```

By using this variable, we can redirect the user back to the form page like so:

```
redirectTo($referer);
```

Resources

- $_SERVER Variable
- HTTP Referer

Understanding Sessions

In this lecture, we learned about sessions. Sessions are variables that continue to hold their value after a response has been sent. We can create sessions by using the superglobal variable `$_SESSION`. If we add values to this array, they'll be persisted. Here's an example:

```
$_SESSION['errors'] = $e->errors;
```

Enabling Sessions

In this lecture, we learned how to enable a session. For this task, we created a middleware, which we registered like so:

```
$app->addMiddleware(SessionMiddleware::class);
```

In our middleware, we enabled sessions by calling a simple function.

```
session_start();
```

Just like that, PHP will keep track of the session. You can verify a session has been started by checking the cookies in your browser. A cookie called `PHPSESSID` will be available with the ID of the session.

Handling Session Exceptions

In this lecture, we created a custom exception for when a session couldn't be started. The name of our exception is called `SessionException`.

Firstly, we don't want to start a session if one is already active. We can use the `session_status()` method to detect an active session.

```
if (session_status() === PHP_SESSION_ACTIVE) {
    throw new SessionException("Session already started");
  }
```

Secondly, we should check if headers were already sent. If they were, a session cannot be started. We can use the `headers_sent()` function to help us detect if headers were sent.

```
if (headers_sent()) {
  throw new SessionException(
    "Headers already sent."
  );
}
```

In the next lecture, we talked about why it's important to check for headers sent.

Common Session Error

In this lecture, we talked about one of the most common errors that can occur from sessions. Sessions must be enabled before you output any content. This is because sessions modify the headers of the response. If content is outputted, headers are sent to the browser. Therefore, sessions can't add headers.

There are two solutions for avoiding this error. Firstly, you can call the `session_start()` function before you output any content. Secondly, you can enable output buffering, which can be done through PHP's configuration.

We decided to improve our exception by creating a variable called `$filename` and `$line`. These variables can be passed into the `headers_sent()` function, which will update them with the filename and line number where content was already outputted. We then used these variables in our error message like so:

```
if (headers_sent($filename, $line)) {
  throw new SessionException(
    "Headers already sent. Consider enabling output buffering. Data outputted from {$filename} - Line: {$line}"
  );
}
```

Resources

- Output Buffering Configuration

Closing the Session Early

In this lecture, we closed the session early to enhance performance. If you no longer need a session, you can close it by calling the `session_write_close()` function. By closing the session, memory is freed so that it can be allocated for other requests.

```
session_write_close();
```

Injecting Errors into a Template

In this lecture, we injected errors into a template from outside a controller. Previously, we used the `render()` method to pass on data to a template. However, middleware shouldn't render a template. Luckily, the Template Engine allows us to pass on data.

First, we called the method for adding global data to a method, which is the `addGlobal()` method. This method accepts the name of the variable and the array.

```
$this->view->addGlobal('errors', $_SESSION['errors'] ?? []);
```

Lastly, we can view the errors by dumping them with the `dd()` function.

```
<?php dd($errors); ?>
```

Flashing Errors

In this lecture, we decided to flash the errors. Errors shouldn't appear anymore after they've been rendered on the template. It's recommended to destroy a variable by calling the `unset()` function. It accepts the variable to destroy. We can use it like so:

```
unset($_SESSION['errors']);
```

Displaying Errors

In this lecture, we rendered the errors by checking for an existence of a variable using the `if` statement and `array_key_exists()` function. Next, we rendered an error with a `<div>` tag.

Here's an example of an error message for the email.

```
<?php if (array_key_exists('email', $errors)) : ?>
<div class="bg-gray-100 mt-2 p-2 text-red-500">
  <?php echo $errors['email'][0]; ?>
</div>
<?php endif; ?>
```

You can use a loop to display all errors if you would like. We decided to keep it simple by displaying the first error in the array.

Validating Emails

In this lecture, we validated the email by using the `filter_var()` function. This function is defined by PHP, which can be used for sanitizing or validating values. There are two arguments, which are the value to validate/sanitize and the filter to apply.

```
(bool) filter_var($data[$field], FILTER_VALIDATE_EMAIL);
```

In this example, we're applying the `FILTER_VALIDATE_EMAIL` filter to validate the value as an email.

Resources

- filter_var() Function

Supporting Rule Parameters

In this lecture, we updated the `Validator` class to support parameters. Some rules may require additional details. Let's say we had the following: `min:18`.

Our class should be able to extract the rule alias and list of parameters. First, we created an empty array to store the parameters.

```
$ruleParams = [];
```

Next, we used the `str_contains()` function to check if the rule contains the `:` character. If it does, we should extract the parameters.

```
if (str_contains($rule, ':')) {

}
```

We can do so by using the `explode()` function and destructuring the results. The first item in the array will be the rule alias, and the second item will be the parameters.

```
[$rule, $ruleParams] = explode(':', $rule);
```

Lastly, we converted the parameters into an array by using the `explode()` function again.

```
$ruleParams = explode(',', $ruleParams);
```

Minimum Validation Rule

In this lecture, we created a rule for comparing a value against a minimum threshold. First, we checked if a threshold was given in the rule's parameters. If a parameter wasn't given, we'll throw the `InvalidArgumentException` class, which is defined by PHP.

```
if (empty($params[0])) {
  throw new InvalidArgumentException('Minimum length not specified');
}
```

Otherwise, we'll proceed to validate the value. First, we typecasted the parameter into an integer and then compared it with the field value.

```
$length = (int) $params[0];

return $data[$field] >= $length;
```

In Validation Rule

In this lecture, we created a rule for checking if a value is inside an array of values. The array of values can be passed in as a parameter. In our `ValidatorService` class, we used the rule for validating a value against an array of countries.

```
in:USA,Canada,Mexico
```

In our `InRule` class, we used the `in_array()` function, which accepts the value to find in an array.

```
in_array($data[$field], $params);
```

Exercise: URL Validation Rule

In this lecture, we created a validation rule for validating URLs by using the `filter_var()` function. We're already familiar with this function. For the second argument, we used the `FILTER_VALIDATE_URL` constant to let the function validate URLs.

```
filter_var($data[$field], FILTER_VALIDATE_URL);
```

Password Matching Rule

In this lecture, we created a rule for matching values from two fields. Firstly, we had to extract the values from both fields. We grabbed these values from the field the rule is applied on and the field specified in the parameter.

```
$fieldOne = $data[$field];
$fieldTwo = $data[$params[0]];
```

Afterward, we compared both field values and returned the result from the `validate()` method of our rule.

```
return $fieldOne === $fieldTwo;
```

Prefilling a Form

In this lecture, we prefilled a form with a user's previously submitted data for a better user experience. First, we created a middleware similar to the validation error middleware to store the form data in a session. After using this middleware, we updated our template to use the old form data.

For `<input>` elements, you can set the `value` attribute like so:

```
<input value="<?php echo $oldFormData['email']; ?>" />
```

For `<select>` elements, you can use the `selected` attribute on `<option>` elements to select a specific option.

```
<select
  name="country"
  class="block w-full mt-1 rounded-md border-gray-300 shadow-sm focus:border-indigo-300 focus:ring focus:ring-indigo-200 focus:rin
>
  <option value="USA">USA</option>
  <option value="Canada" <?php echo ($oldFormData['country'] ?? '') == "Canada" ? "selected" : ""; ?>>Canada</option>
  <option value="Mexico" <?php echo ($oldFormData['country'] ?? '') == "Canada" ? "selected" : ""; ?>>Mexico</option>
</select>
```

Lastly, for checkboxes, you can add the `checked` attribute to an `<input>` element like so.

```
<input
<?php echo $oldFormData['tos'] ?? false ? "checked" : ""; ?>
/>
```

Filtering Sensitive Data

In this lecture, we filtered the password fields from the form data passed on to the template. First, we stored the form data in a variable and defined a list of fields to exclude from the session data.

```
$oldFormData = $request->getParsedBody();
$excludedFields = ['password', 'confirmPassword'];
```

Next, we used the `array_diff_key()` function to merge two arrays where identical keys are filtered out of the result. For the `$excludedFields` variable, we flipped the key and values by using the `array_flip()` function.

```
$filteredFormData = array_diff_key($oldFormData, array_flip($excludedFields));
```

Lastly, we stored this data inside the session.

```
$_SESSION['oldFormData'] = $filteredFormData;
```

Section 14: MySQL

In this section, we started learning how to use MySQL for managing databases.

Introduction to SQL

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- MySQL
- MariaDB
- MySQL Workbench
- HeidiSQL
- Sequel Pro

Creating a Database

In this lecture, we learned how to create a database with the help of PHPMyAdmin. A single server is allowed to have multiple databases. In the tool, we created a database called `phpiggy`, which matches the name of our project. Not required but recommended.

Secondly, we configured the collation. The collation is the character set supported by the database. If you're interested in storing various characters, you should use the `utf8mb4_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. 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
```

Using PHPMyAdmin

In this lecture, we learned how to use PHPMyAdmin for creating tables, inserting data, reading data, updating data, and deleting data. We also learned about engines in SQL databases.

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.

Enabling PDO Extension

In this lecture, we took the time to check that the PDO extension was enabled. PHP is a modular language. We can enable/disable features for whatever reason. You can check out the resource section for a complete list of extensions for enabling specific features in the PHP language.

There are two extensions for interacting with databases called `mysqli` and `pdo`. The `mysqli` extension is only compatible with MySQL, whereas the `pdo` extension is compatible with multiple SQL databases. Most developers prefer the `pdo` extension since you can switch to a different database with very few changes to your codebase.

You can verify the PDO extension is installed by using the `phpinfo()` function. If the `pdo_mysql` extension is listed under the list of drivers, you should be good to.

Otherwise, you must update your environment's `php.ini` file. This line of code must be uncommented in the configuration file:

```
extension=pdo_mysql
```

Afterward, you must restart the server for the changes to take effect and verify that the extension is listed from the page generated by the `phpinfo()` function.

Resources

- PECL

Custom Composer Scripts

In this lecture, we learned how to execute custom scripts from the command line with the help of composer. In the `composer.json`, we can add an object called `scripts`. Inside this object, we can add a name for our command followed by the command to execute. You can think of our custom command as an alias or shortcut for another command.

```
{
  "scripts": {
```

```
    "phpiggy": "php cli.php"
  }
}
```

In this example, we're creating a custom command called `phpiggy` that runs the `php` command to execute a PHP script. The `php` command accepts a filename. We can run this command with composer by using the `composer run-script phpiggy` command.

Understanding DSN

Databases have a standardized format called DSN, which is short data source name. DSNs contain the connectivity details to a database. The format is the driver, followed by the host and database name.

```
driver:host=value;dbname=value
```

Resources

- Data Source Name

Connecting to a Database

In this lecture, we learned how to connect to a database using PDO. The following information is required:

- Driver - A driver for communicating with a specific database since PDO supports multiple database.
- Host - The URL or IP to where the database is hosted.
- Database Name - The name of the database since multiple databases can be hosted on a single server.
- Username - The username of an account in a database.
- Password - The password associated with the username.

We created variables for each of these values.

```
$driver = 'mysql';
$host = 'localhost';
$database = 'phpiggy';
$dsn = "{$driver}:host={$host};dbname={$database}";
$username = 'root';
$password = '';

$db = new PDO($dsn, $username, $password);
```

We can connect to a database with PHP by creating a new instance of the `PDO` class, which accepts the DSN, username, and password.

The PDOException Class

In this lecture, we learned how to catch errors when PHP is unable to connect to a database. Even if our credentials are valid, it's possible that PHP may be unable to connect to a database if the server is down. By default, the error message produced by the `PDO` class contains the username and password, which gives users access to our database.

To avoid our database credentials from being outputted onto a page, we can catch the exception. PDO throws an exception called `PDOException`. Here's an example of how to catch that exception.

```
try {
  $db = new PDO($dsn, $username, $password);
} catch (\PDOException $e) {
  die("Unable to connect to database.");
}
```

Refactoring the Database Connection

In this lecture, we created a class in our framework for handling the database connection called `Database`. Here's what the class looks like. Instead of hardcoding the values into the connection, the class accepts them as arguments to the construct method.

```
namespace Framework;

use PDO, PDOException;

class Database
{
  public PDO $connection;

  public function __construct(
    string $driver,
```

```
    array $config,
    string $username,
    string $password
  ) {
    $formattedConfig = http_build_query(data: $config, arg_separator: ';');
    $dsn = "{$driver}:{$formattedConfig}";

    try {
      $this->connection = new PDO($dsn, $username, $password);
    } catch (PDOException $e) {
      die("Unable to connect to database.");
    }
  }
}
```

Querying the Database

In this lecture, we queried the database using PDO. There's a method called `query()` that accepts a query. The value returned by this method is an instance of a class called `PDOStatement`. Oftentimes, developers will store the value in a variable called `$stmt`.

```
$query = 'SELECT * FROM products';

$stmt = $db->connection->query($query);

echo "<pre>";
var_dump($stmt->fetchAll());
echo "</pre>";
```

We can view the results by calling the `fetchAll()` method. If our query selects multiple records, all records will be presented on the page.

Fetch Modes

In this lecture, we learned how to modify the results. By default, PHP uses a mode called `PHP::FETCH_BOTH`, which returns the results in an array with numeric and named keys. We can modify the mode by passing in the mode into the `query()` or `fetchAll()` method like so.

```
$db->connection->query($query, PDO::FETCH_BOTH);
$stmt->fetchAll(PDO::FETCH_BOTH);
```

There are dozens of modes available, but here are the most popular ones.

- `PDO::FETCH_NUM` - Specifies that the fetch method shall return each row as an array indexed by column number as returned in the corresponding result set, starting at column 0.
- `PDO::FETCH_ASSOC` - Specifies that the fetch method shall return each row as an array indexed by column name as returned in the corresponding result set. If the result set contains multiple columns with the same name, `PDO::FETCH_ASSOC` returns only a single value per column name.
- `PDO::FETCH_OBJ` - Specifies that the fetch method shall return each row as an object with property names that correspond to the column names returned in the result set.

SQL Injections

In this lecture, we learned how our current solution leaves us vulnerable to an SQL injection. It's not uncommon to allow user input in a query. One solution to allow user input is to use query strings. Let's say we had the following query:

```
$query = 'SELECT * FROM products WHERE name="' . $search . '"';
```

In this example, we are using the `$search` variable to filter the results. Currently, we're not sanitizing the value, so its completely possible to use the following value to get a complete list of results:

```
$search = " OR 1=1 --"
```

In this example, we're adding an `OR` keyword to the query to add an additional conditional statement to our `WHERE` portion of the query. If `1` equals `1`, SQL will return all records in a table instead of a specific record. We're also commenting the last `"` character by using the `--` characters.

The query can be modified to drop tables, or even worse, modify a user's passwords. We can avoid these scenarios by using prepared statements, which will be covered in the next lecture.

Resources

- Superglobals

Prepared Statements

In this lecture, we learned how to avoid SQL injections by using prepared statements. A prepared statement is a query with placeholders. SQL can replace these placeholders with values while validating the values to prevent injections.

The first step is to update our query by replacing the value with a placeholder. We can use the `?` character as the placeholder.

```
$query = 'SELECT * FROM products WHERE name = ?';
```

Next, we can pass on the query to the `prepare()` method.

```
$stmt = $db->connection->prepare($query);
```

Unlike the `query()` method, the `prepare()` method does not immediately execute the query. We must do so from the `execute()` method, which accepts an array of values to replace the placeholders. The first placeholder gets replaced with the first item in the array, the second placeholder gets replaced with the second item in the array, and so on and so forth.

```
$stmt->execute([$search]);
```

We can also use named parameters, which always start with the `:` character.

```
$query = 'SELECT * FROM products WHERE name = :name';
```

In the `execute()` method, we must use an associative array to map the values to their placeholders.

```
$stmt->execute([":name" => $search]);
```

Alternatively, we can bind the values with the `bindValue()` method, which accepts the parameter name, value, and data type (optional).

```
$stmt = $db->connection->prepare($query);

$stmt->bindValue(":name", "Gloves", PDO::PARAM_STR);
```

Understanding Transactions

In this lecture, we talked about what transactions are for in SQL. They're a feature for guaranteeing that multiple queries successfully execute. If a single query fails, all previous queries can be reverted.

Not all database engines support transactions. If you're using `InnoDB`, transactions are supported. It's always good to check beforehand.

Resources

- MySQL Database Engines Comparison

Creating Transactions with PDO

In this lecture, we learned how to create a transaction. Firstly, we must call the `beginTransaction()` method. Any queries written after this method will be grouped with the transaction.

```
$db->connection->beginTransaction();
```

Once you're finished your queries, you can close the transaction by calling the `commit()` method.

```
$db->connection->commit();
```

If the previous queries fail, you can manually revert them by calling the `rollBack()` method. Before you do, you might want to check if there's an existing transaction by using the `inTransaction()` method.

```
if ($db->connection->inTransaction()) {
  $db->connection->rollBack();
}
```

This method can be useful before creating a new transaction since only one transaction can be active at a time.

Understanding Data Modeling

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- DrawSQL
- PHPiggy

Designing a User Table

In this lecture, we designed a `users` table for our database using **DrawSQL**. A couple of new things we learned about databases were the following:

- Most tables have an `id` column to identify each row in a table.
- ID columns use the `bigint` type since each record will have a unique value, and you want your table to accomodate as many records as possible.
- The primary key index can be assigned to a specific column to constrain the values to be unique and to make it faster to search for specific records by their ID.
- The auto increment option allows the database to set a unique number for the ID.
- The unsigned option disallows tables from having negative numbers.
- The unique key index can be used for constraining a column to a unique value.
- It's common practice to use snake casing for columns with multiple words.

Creating a Table in an SQL File

In this lecture, we outsourced the query for creating a table in SQL file called `database.sql`. It contains the following contents:

```
CREATE TABLE `users` (
  `id` bigint(20) UNSIGNED NOT NULL AUTO_INCREMENT,
  `email` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  `age` tinyint(3) UNSIGNED NOT NULL,
  `country` varchar(255) NOT NULL,
  `social_media_url` varchar(255) NOT NULL,
  `created_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `updated_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY (`email`)
);
```

In this query, I've introduced a few new keywords.

- `UNSIGNED` - Only allows positive numbers
- `NOT NULL` - Prevents a column from containing a `null` value
- `AUTO_INCREMENT` - Assigns a default value of `1` that is incremented by 1 for each record.
- `DEFAULT` - Allows a default value to be assigned to a record if one is not given during insertion
- `CURRENT_TIMESTAMP` - A function producing a timestamp compatible with the `datetime` data type
- `PRIMARY KEY` - Allows us to set a column as a primary key
- `UNIQUE KEY` - ALlows us to set a column as a unique key

Loading Files

In this lecture, we grabbed the contents of a file with the help of the `file_get_contents()` function. This function accepts a relative path to the file. The file's contents are returned as a string.

```
$sqlFile = file_get_contents("./database.sql");
```

Conditionally Creating Tables

In this lecture, we conditionally created a table by adding the `IF NOT EXISTS` after the `CREATE TABLE` keywords like so.

```
CREATE TABLE IF NOT EXISTS
```

By adding these keywords, our database won't throw an error if the table already exists since duplicate tables are not allowed.

Refactoring the Query

This lecture does not have any notes. The content is meant to be consumed via video.

Section 15: User Registration and Authentication

In this section, we learned how to register a user into our system and perform authentication.

Database Container Definition

This lecture does not have any notes. The content is meant to be consumed via video.

Understanding Environment Variables

In this lecture, we learned about environment variables. Environment variables are just global variables, except they're available to the entire environment instead of just our scripts. By default, environment variables are not loaded into our application.

We installed a package called `phpdotenv` to help us with loading environment variables into our application. We used the following command to install the package.

```
composer require vlucas/phpdotenv
```

Resources

- Environment Variables Article
- PHPDotEnv

Creating Environment Variables

In this lecture, we created our first environment variable. Environment variables are created in a file called `.env` that you can place in the root directory of your project.

```
DB_DRIVER=mysql
```

Environment files follow a key-value format. It's common to use uppercase letters for a variable name.

Next, we imported the `Dotenv\Dotenv` class. This class has a method called `createImmutable` to initialize the package. It accepts the path to the file.

```
use Dotenv\Dotenv;

$dotenv = Dotenv::createImmutable(Paths::ROOT);
$dotenv->load();
```

In this example, we're passing in `Paths::ROOT` constant. Afterward, we call the `load` method to load the environment file.

The environment variables are stored in a superglobal variable by PHP called `$_ENV`. Here's an example of how we used these variables in our database definition.

```
Database::class => fn () => new Database($_ENV['DB_DRIVER'], [
  'host' => $_ENV['DB_HOST'],
  'port' => $_ENV['DB_PORT'],
  'dbname' => $_ENV['DB_NAME'],
], $_ENV['DB_USER'], $_ENV['DB_PASS'])
```

Resources

- ENV Superglobal

Ignoring Environment Files

In this lecture, we instructed Git to ignore the `.env` file since environment variables can contain sensitive information. We updated the `.gitignore` file by adding the `.env` file to the list of files.

```
composer.phar
/vendor/

# Commit your application's lock file https://getcomposer.org/doc/01-basic-usage.md#commit-your-composer-lock-file-to-version-control
# You may choose to ignore a library lock file http://getcomposer.org/doc/02-libraries.md#lock-file
# composer.lock
```

`.env`

In addition, we created a `.env.example` file so that other developers can fill out the values for the environment variables when copying the project.

Passing on the Container to Definitions

In this lecture, we passed on the `Container` instance to our factory functions. By doing so, the factory functions can grab additional dependencies from the container. First, we updated the `Container` class by passing on the `$this` keyword when invoking the `$factory` function.

```
$dependency = $factory($this);
```

Next, we used the container to help us grab the database so that we can pass it onto the `UserService` instance like so.

```
UserService::class  => function (Container $container) {
  $db = $container->get(Database::class);

  return new UserService($db);
}
```

Counting Records with SQL

In this lecture, we learned how to count records from a query by using the `COUNT()` function. This function counts a specific column with non-null values. If we're not interested in counting a specific column, we can pass in a `*` character to count the records. This function can be called after the `SELECT` keyword.

```
SELECT COUNT(*) FROM users WHERE email = :email
```

Resources

- Built-in Functions

Supporting Prepared Statements

This lecture does not have any notes. The content is meant to be consumed via video.

Validating Duplicate Emails

In this lecture, we validated the user's email by checking for duplicate emails. To perform this task, we called the `fetchColumn()` method on the `PDOStatement` class. Results from queries are stored in arrays. Since we're only using the `COUNT` function, we can grab the first item in the array to grab the number of records from our query.

The `fetchColumn()` method automatically grabs the first item in the array.

```
public function count()
{
  return $this->statement->fetchColumn();
}
```

Lastly, we used the results to check if there were any accounts found. If so, we threw the `ValidationException` with an error to describe the email that was taken.

```
if ($emailCount > 0) {
  throw new ValidationException(['email' => ['Email taken']]);
}
```

Exercise: Inserting a User

This lecture does not have any notes. The content is meant to be consumed via video.

Understanding Hashing

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- MD5 Hash Generator
- Types of Password Attacks

Hashing a Password

In this lecture, we hashed a password using the `password_hash()` function. This function has three arguments.

- The password to hash.
- The algorithm to use.
- An additional set of options.

Here's an example of using this function.

```
password_hash("somepassword", PASSWORD_BCRYPT, ['cost' => 12])
```

The second argument can be configured with a predefined PHP constant. You can refer to the documentation for a complete list of supported algorithms. Bcrypt is the most popular algorithm. We can configure the bcrypt algorithm with the third argument. In this case, we're setting the cost to `12` to allow the algorithm to use more resources.

Keep in mind, the bcrypt algorithm automatically salts our password and can generate a unique hash for even the same password.

Resources

- pasword_hash() function

Preparing the Login Page

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- Login Template

Exercise: Validating the Login Form

This lecture does not have any notes. The content is meant to be consumed via video.

Validating the User's Credentials

In this lecture, we validated the user's credentials by comparing the form data with the data in a database. First, we queried the database. We can use a method called `fetch()` to select a single record. In this example, we're grabbing a record based on a user's email.

```
$user = $this->db->query("SELECT * FROM users WHERE email = :email", [
  "email" => $formData['email']
])->find();
```

We update the `Database` class to contain a method called `find()`, which calls the `fetch()` method on the connection.

```
public function find()
{
  return $this->statement->fetch();
}
```

In addition, we updated the `PDO` instance by passing in an array of configuration options to set the fetch mode to be an associative array.

```
$this->connection = new PDO($dsn, $username, $password, [
  PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC
]);
```

After retrieving the record, we verified the passwords match using the `password_verify()` function. This function accepts the password in raw text and the password in hashed form. We used the null-safe operator and typecasted the value into a string to reduce the likelihood of an error.

```
$passwordsMatch = password_verify(
  $formData['password'],
```

```
    $user['password'] ?? ''
);
```

Afterward, we performed a conditional statement to check if either variable was false, meaning the user provided invalid credentials.

```
if (!$user || !$passwordsMatch) {
  throw new ValidationException([
    'password' => 'Invalid email or password.'
  ]);
}
```

Next, we stored the ID of the user in a session variable.

```
$_SESSION['user'] = $user['id'];
```

Lastly, we redirected them to the home page.

```
redirectTo("/");
```

Understanding Session Hijacking

This lecture does not have any notes. The content is meant to be consumed via video.

Configuring Session Cookies

In this lecture, we discussed the various settings of a cookie and how to configure the session ID cookie settings. In the session middleware, we modified the following settings:

```
session_set_cookie_params([
  'secure' => $_ENV['APP_ENV'] === "production",
  'httponly' => true,
  'samesite' => 'lax'
]);
```

The `session_set_cookie_params()` function accepts an array of settings we'd like to configure. Here's what we updated.

- `secure` - Checks if the user is on a secure connection. If they aren't, a cookie won't be sent.
- `httponly` - Only allows a cookie to be used over HTTP. Disallows access to a cookie in JavaScript.
- `samesite` - Determines where a cookie can be sent. If set to `lax`, allows for a cookie to be sent over an external link.

Resources

- session_set_cookie_params() Function

Regenerating a Session ID

In this lecture, we regenerated the session ID after a user logs in. We shouldn't regenerate the ID on every request as the `session_regenerate_id()` function is unreliable at times. It's recommended to update the ID when a user logs into an application and when they log out. You can create a new ID like so:

```
session_regenerate_id();
```

Protecting Routes

In this lecture, we protected our routes from guests or authenticated users using middleware. In our middleware, we can check the `$_SESSION['user']` variable to verify the user is logged in. For example, let's say we wanted to check if the user is not logged in and redirect them to the login page if they're not, we can use the following condition:

```
if (empty($_SESSION['user'])) {
  redirectTo("/login");
}
```

We did something similar for routes that can only be accessed by guests called `GuestOnlyMiddleware`.

Resources

- Route Middleware Gist

Applying Route Middleware

In this lecture, we updated the `Router` class to support middleware for routes. Not all middleware may need to be applied to all routes. In that case, we should allow for routes to be added to specific routes. First, we updated the route registration by including a key called `middlewares` containing an array of middleware.

```
$this->routes[] = [
  'path' => $path,
  'method' => strtoupper($method),
  'controller' => $controller,
  'middlewares' => []
];
```

Next, we defined a method called `addRouteMiddleware` for adding new items to the middleware. Through this method, we're registering middleware to the last route registered with our router by using the `array_key_last()` function. This function grabs the key from the last item in the array. It accepts the array as an argument. With this information, we registered the middleware to the last item in the array.

```
public function addRouteMiddleware(string $middleware)
{
  $lastRouteKey = array_key_last($this->routes);

  $this->routes[$lastRouteKey]['middlewares'][] = $middleware;
}
```

Lastly, we merged the routes from the global middleware and route to a variable called `$allMiddleware`. We updated the loop to loop through this array.

```
$allMiddleware = [...$route['middlewares'], ...$this->middlewares];

foreach ($allMiddleware as $middleware) {
  $middlewareInstance = $container ?
    $container->resolve($middleware) :
    new $middleware;
  $action = fn () => $middlewareInstance->process($action);
}
```

Middleware can be applied to a route by chaining the `add()` method and passing in the middleware to a route like so.

```
$app->get('/', [HomeController::class, 'home'])->add(AuthRequiredMiddleware::class);
```

Logging out of the Application

In this lecture, we logged out of the application by unsetting the `$_SESSION['user']` variable and then regenerating the sesssion ID.

```
unset($_SESSION['user']);

session_regenerate_id();
```

Authenticating Registered Users

In this lecture, we authenticated the user after creating their account. Similar to before, all we did was regenerate the ID and stored the ID in a session.

```
session_regenerate_id();

$_SESSION['user'] = $user['id'];
```

Understanding CSRF

This lecture does not have any notes. The content is meant to be consumed via video.

Guarding a CSRF Token

In this lecture, we created middleware for generating tokens called `CsrfTokenMiddleware`. To generate a token, we used two functions called `random_bytes()` and `bin2hex()`. The `random_bytes()` function accepts a size. It'll return a randomly generated value in binary format. Since browsers can't read binary data, we must use the `bin2hex()` function to convert the generated value into plaintext.

```
$_SESSION['token'] = $_SESSION['token'] ?? bin2hex(random_bytes(32));
```

The token is stored as a session. We're also reusing the token so that we don't constantly create tokens on every request. Lastly, we injected the token into our templates.

```
$this->view->addGlobal('csrfToken', $_SESSION['token']);
```

Resources

- CSRF Middleware

Rendering Tokens

In this lecture, we created a partial called `_csrf.twig`. In this partial, we loaded the token as a hidden input.

```
<input type="hidden" name="token" value="<?php echo e($csrfToken); ?>" />
```

Lastly, we loaded this partial into any template that has a form like so.

```
<?php include $this->resolve("partials/_csrf.php"); ?>
```

Validating CSRF Tokens

In this lecture, we validated the token by creating a middleware called `CsrfGuardMiddleware`. In this class, before performing validation, we're checking if the HTTP method is `POST`, `PATCH`, or `DELETE`. `GET` requests are ignored.

```
$requestMethod = strtoupper($_SERVER['REQUEST_METHOD']);
$validMethods = ['POST', 'PATCH', 'DELETE'];

if (!in_array($requestMethod, $validMethods)) {
  $next();
  return;
}
```

Next, we compared the token from the request with the token from the submission. If the tokens don't match, we have an invalid token. In that case, the user is redirected to the homepage.

```
if ($_SESSION['token'] !== $_POST['token']) {
  redirectTo("/");
}
```

Lastly, we deleted the token since tokens should only be used once per form submission.

```
unset($_SESSION['token']);
```

Resources

- CSRF Guard Middleware

Conditionally Rendering Sections

In this lecture, we updated the header's links to display dynamically based on the user's authentication status. We checked if the user is logged in by checking if the `$_SESSION['user']` variable is defined.

```
<?php if (isset($_SESSION['user'])) : ?>
  <!-- Content here -->
<?php endif; ?>
```

Section 16: CRUD Transactions

In this section, we created a CRUD interface for transactions.

Designing the Transactions Table

This lecture does not have any notes. The content is meant to be consumed via video.

Understanding Database Relationships

In this lecture, we talked about database relationships. Often, tables are connected to each other. We refer to these connections as relationships. There are different types of relationships.

- **One to many** - A one-to-many relationship occurs when one record in table 1 is related to one or more records in table 2.
- **One-to-one** - A one-to-one relationship in a database occurs when each row in table 1 has only one related row in table 2.
- **Many-to-many** - A many-to-many relationship occurs when multiple records in one table are related to multiple records in another table.

Relationships are established with foreign keys. On a technical level, a foreign key is a constraint that links a column in one table to a column in a different table and ensures that a value can be added if the same value already exists.

Adding Foreign Keys

In this lecture, we created a table for storing transactions based on the schema outlined in the previous lecture. Most of the process is familiar to us, with the exception of adding a foreign key. A foreign key can be added by using the `FOREIGN KEY` keywords. This keyword is followed by the column name storing the foreign key in the current table.

Next, we must provide the table that the current table should have a relationship with by adding the `REFERENCES` keyword. This keyword is followed by the table name and the column name. Here's the entire query:

```
CREATE TABLE IF NOT EXISTS `transactions` (
  `id` bigint(20) UNSIGNED NOT NULL AUTO_INCREMENT,
  `description` varchar(255) NOT NULL,
  `amount` decimal(10,2) NOT NULL,
  `date` datetime NOT NULL,
  `created_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP(),
  `updated_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP(),
  `user_id` bigint(20) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`),
  FOREIGN KEY (`user_id`) REFERENCES `users` (`id`)
);
```

Preparing the Create Transaction Page

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- Transaction Files Gist

Validating Transactions

This lecture does not have any notes. The content is meant to be consumed via video.

Validating Maximum Character Length

In this lecture, we defined a custom rule for validating the maximum character length of a string. In our rule, we validated the parameter by checking if one is provided as it'll be necessary since we want the maximum length to be flexible. After verifying it's been provided, we used the `strlen()` function to count the characters in a string and compared it against the maximum length from the parameter.

```
if (empty($params[0])) {
  throw new InvalidArgumentException('Maximum length not specified');
}

$length = (int) $params[0];

return strlen($data[$field]) < $length;
```

We can use the rule like so.

```
lengthMax:255
```

Resources

- Length Max Rule Gist

Validating Numbers

In this lecture, we defined a custom rule for validating if a value is a number. PHP has a function called `is_numeric()` that can check either integers, floats, or strings to verify the value is a number. This function returns a boolean, which we can use to validate the value.

```
is_numeric($data[$field]);
```

Resources

- Numeric Rule Gist

Validating Dates

In this lecture, we created a custom rule for validating if a date is in a valid format. To perform this task, we used the `date_parse_from_format()` function. This function grabs information about a date in any format. We can pass in the format and date as arguments.

```
$parsedDate = date_parse_from_format($params[0], $data[$field]);
```

If the function is not able to grab the date, it's because the date and value are not compatible. In that case, the return value will contain two keys called `error_count` and `warning_count` with the number of errors with the value. We checked these keys to make sure they're set to `0`, which means there aren't errors and the value is in the valid format.

```
return $parsedDate['error_count'] === 0 && $parsedDate['warning_count'] === 0;
```

Resources

- Date Format Rule
- date_parse_from_format() Function

Creating a Transaction

In this lecture, we inserted the transaction with the data from the form submission. Most of what we did is already familiar to us. The most notable thing was the value of the date. By using the `datetime` data type, we must provide a date with the date and time. However, the form only provides the date. We decided to add the time along with the date.

```
$formattedDate = "{$formData['date']} 00:00:00";
```

You can refer to the documentation for the proper format.

Resources

- Datetime MariaDB

Retrieving Transactions

In this lecture, we retrieved a list of transactions uploaded by the user. Before doing so, we updated the `Database` class to define a method called `findAll()` to return an array of results. Behind the scenes, this method calls the `fetchAll()` method from the `PDOStatement` class to grab all results.

```
$this->statement->fetchAll();
```

Formatting Dates with SQL

In this lecture, we updated our query to format the date. The default format for the `datetime` data type is the date and time. We can customize the format by using the `DATE_FORMAT` SQL function. This function accepts the column with the date value and the new format. You can refer to the documentation for the various placeholders you can use.

Lastly, we must provide an alias/name for the new value by using the `AS` keyword followed by the name.

```
SELECT *, DATE_FORMAT(date, '%Y-%m-%d') AS formatted_date
FROM transactions
```

```
WHERE user_id = :user_id
```

After querying the database, the new format will be accessible as `formatted_date`.

Resources

- DATE_FORMAT Function

Query Parameters

In this lecture, we talked about query parameters. URLs can store data called query parameters. A query parameter can be applied to a URL by adding a `?` followed by a pair of key-values separated by the `&` character. Forms can add query parameters by setting the `method` attribute to `GET`.

```
<form method="GET"></form>
```

In addition, each input must have a `name` attribute as it'll be used in the query parameter.

```
<input name="s" />
```

Lastly, we can use the `$_GET` superglobal variable to access a query parameter.

```
$searchTerm = $_GET['s'] ?? '';
```

SQL LIKE Condition

In this lecture, we looked at the `LIKE` condition to search for values within a column. In most cases, we used the following queries to grab data from our tables.

```
SELECT * FROM transactions
WHERE description='e'
```

However, this performs an exact match. In some cases, you may want to perform a partial match. That can be performed with a `LIKE` condition.

```
SELECT * FROM transactions
WHERE description LIKE '%e%'
```

You must add the `%` to either end of the string. If added to the end, the value will be compared to the beginning of the value in the column. Whereas adding it to the beginning of the value will be compared to the end of the value in the column.

This solution isn't perfect. Typically, you would consider using a product such as **Elastisearch** to help you add advanced search features to your application. This solution is meant to be a quick and simple way to add a search feature to a site. It may not always yield the most accurate results.

Resources

- Elastisearch

Filtering Transactions

In this lecture, we created a custom filter to apply our search using the `LIKE` condition. We created the following query.

```
SELECT *, DATE_FORMAT(date, '%Y-%m-%d') AS formatted_date
FROM transactions
WHERE user_id = :user_id
AND description LIKE :description
```

In this example, we're adding an additional condition by comparing the `description` column against a value with the `LIKE` clause.

Escaping the Search Term

In this lecture, we decided to escape the search term since it's possible for transactions to have the `%` in their description. We can use the `addcslashes()` function to escape specific characters. It has two arguments, which is the string to escape a list of characters to escape.

```
$searchTerm = addcslashes($_GET['s'] ?? '', '%_');
```

SQL Limit Clause

In this lecture, we learned about the `LIMIT` clause. This clause allows us to limit the number of results. We must add this keyword at the end of the query followed by the limit. We can also add an offset to retrieve results after a specific point. Here's an example of applying the `LIMIT` clause.

```
SELECT * FROM transactions
LIMIT 3,1
```

Limiting Results

In this lecture, we applied the `LIMIT` clause with Doctrine. First, we prepared variables for storing the length and offset. We're going to store the current page in a query parameter called `p`.

```
$page = $_GET['p'] ?? 1;
$page = (int) $page;
$length = 3;
$offset = ($page - 1) * $length;
```

Next, we updated our query with the `LIMIT` and `OFFSET` keywords. The `OFFSET` keyword is an alternative solution to setting an offset. The shorthand solution for the `LIMIT` keyword is optional. The main advantage of the `OFFSET` keyword is the extra clarity.

```
SELECT *, DATE_FORMAT(date, '%Y-%m-%d') AS formatted_date
FROM transactions
WHERE user_id = :user_id
AND description LIKE :description
LIMIT {$length} OFFSET {$offset}
```

Previous Page Link

In this lecture, we worked on the previous link. For this link, we checked if the current page is greater than 1. If it is, we rendered this link. First, we prepared the data. We grabbed the search term by checking if it had a value. Otherwise, the value will be `null`.

```
$searchTerm = $_GET['s'] ?? null;
```

We're using `null` since the `http_build_query()` function will not add the parameter to the query if the value is `null`. This behavior is preferable if the user isn't performing a search.

```
http_build_query([
  'p' => $page - 1,
  's' => $searchTerm
])
```

Here's the HTML for this link.

```
<?php if ($currentPage > 1) : ?>
  <a href="/?<?php echo $prevPageQuery; ?>" class="inline-flex items-center border-t-2 border-transparent pr-1 pt-4 text-sm font-m
    <svg class="mr-3 h-5 w-5 text-gray-400" viewBox="0 0 20 20" fill="currentColor" aria-hidden="true">
      <path fill-rule="evenodd" d="M18 10a.75.75 0 01-.75.75H4.66l2.1 1.95a.75.75 0 11-1.02 1.1l-3.5-3.25a.75.75 0 010-1.1l3.5-3.2
    </svg>
    Previous
  </a>
<?php endif; ?>
```

Next Page Link

In this lecture, we grabbed the next page link by querying the database for the total number of results. In our service, we used the following queries for counting the transactions submitted by the user.

```
$transactionCount = $this->db->query(
  "SELECT COUNT(*)
  FROM transactions
  WHERE user_id = :user_id
  AND description LIKE :description",
```

```
  $params
)->count();
```

Next, we returned the results along with the list of transactions in an array like so.

```
return [$transactions, $transactionCount];
```

Lastly, we can grab the results by destructuring them.

```
[$transactions, $count] = $this->transactionService->getUserTransactions(
  $length,
  $offset
);
```

Lastly, we divided the total number of results by the length per page to know what the last page is. We surrounded this equation with the `ceil()` function to round the result to an integer.

This is how we used the `lastPage` variable to render the link for the next page.

```
<?php if ($currentPage < $lastPage) : ?>
  <a href="/?<?php echo $nextPageQuery; ?>" class="inline-flex items-center border-t-2 border-transparent pl-1 pt-4 text-sm font-m
    Next
  </a>
<?php endif; ?>
```

Page Number Links

In this lecture, we rendered links for each page. First, we had to generate query parameters for each link. We did so by creating an array of numbers using the `range()` function, which accepts a minimum and maximum size for the array. We used the `$lastPage` variable to set the maximum size.

Next, we looped through the array using the `array_map()` function, where the callback function returns a query parameter with the page number and search term.

```
$pages = $lastPage ? range(1, $lastPage) : [];

$pageLinks = array_map(fn ($pageNum) => http_build_query([
  'p' => $pageNum,
  's' => $searchTerm
]), $pages);
```

We passed on this information to our template and rendered the links by looping through them with the `foreach` keyword like so.

```
<?php foreach ($pageLinks as $pageNum => $query) : ?>
  <a href="/?<?php echo $query; ?>" class="<?php echo ($pageNum + 1) === $currentPage ? 'border-indigo-500 text-indigo-600' : 'bor
    <?php echo $pageNum + 1; ?>
  </a>
<?php endforeach; ?>
```

Preparing the Edit Route

In this lecture, we prepared the route for editing a transaction. During this process, we talked about the difference between route parameters and query parameters. They can be used interchangeably, but it's recommended to use route parameters for selecting resources whereas query parameters should be used for sorting and filtering through resources.

Replacing Strings with Regular Expressions

In this lecture, we updated the `Router` class to create a regular expression based on the path. The regular expression will be responsible for extracting the values from the path. First, we created a regular expression to replace the route parameters with the expression for replacing values.

```
$regexPath = preg_replace('#{[^/]+}#', '([^/]+)', $path);
```

In this example, we're using the `preg_replace()` function to replace the placeholders with a regular expression. The regular expression grabs any value inside the path segment.

This path was added to the route record like so.

```
$this->routes[] = [
  'path' => $path,
  'method' => strtoupper($method),
  'controller' => $controller,
  'middlewares' => [],
  'regexPath' => $regexPath,
];
```

Resources

- Regex101

Extracting Route Parameter Values

In this lecture, we used the regular expression inserted into the path by updating the `preg_match()` function. We added a third argument, which is a variable for storing the values extracted from the path.

```
preg_match("#^{$route['regexPath']}$#", $path, $paramValues)
```

Next, we removed the first item from the results since it's the entire path with the `array_shift()` function. This function accepts the array that should have its first item removed.

```
array_shift($paramValues);
```

Afterward, we used a regular expression to extract the parameter names since the `$paramValues` variable only contains the values. We're trying to connect the placeholder names with their respective values. We grabbed the names by using the `preg_match_all()` function. This function is similar to the `preg_match()` function except it'll grab all results from a regular expression whereas the `preg_match()` function only grabs one result.

```
preg_match_all('#{([^}]+)}#', $route['path'], $paramNames);
$paramNames = $paramNames[1];
```

Then, we combined the keys and values by using the `array_combine()` function. The first argument is the array of key names, and the second argument is the values to assign to each key.

```
$params = array_combine($paramNames, $paramValues);
```

Lastly, we passed on this information to our controller.

```
$action = fn () => $controllerInstance->$func($params);
```

Edit Transaction Template

In this lecture, we worked on the template for editing a transaction. The template is mostly the same as the form for creating a transaciton. The most important part of this lecture was understanding that browsers require a specific format for setting the date in an input with the `date` type. We must use the following format: `YYYY-mm-dd`

Since our date is stored with the `datetime` data type, we used the `DATE_FORMAT` function like so.

```
SELECT *, DATE_FORMAT(date, '%Y-%m-%d') AS formatted_date
FROM transactions
WHERE user_id = :user_id
AND id = :id
```

Resources

- Edit Transaction Template
- HTML Date Formats

Updating a Transaction

This lecture does not have any notes. The content is meant to be consumed via video.

Overriding HTTP Methods

In this lecture, we updated our `Router` class to support HTTP method overriding. It's a common feature found in most frameworks for allowing forms to use different HTTP methods since only `POST` or `GET` requests are allowed. In the form, we added a hidden input with the name `_method`. The value for this input will contain the new HTTP method.

```
<input type="hidden" name="_METHOD" value="DELETE" />
```

Afterward, we updated our `App` instance to register routes with the `DELETE` method.

```
public function delete(string $path, array $controller): App
{
  $this->router->add('DELETE', $path, $controller);

  return $this;
}
```

We can register routes with this method like so.

```
$app->delete('/transaction/{transaction}', [TransactionController::class, 'delete']);
```

Lastly, we can override the method in our `Router` class by checking if the `$_POST` request contains the `_method` item. If so, we used this method instead of the method sent with the browser.

```
$method = strtoupper($_POST['_METHOD'] ?? $method);
```

Deleting a Transaction

This lecture does not have any notes. The content is meant to be consumed via video.

Section 17: Handling File Uploads

In this section, we learned how to properly handle file uploads while also validating and securing them in our application.

Preparing the Receipt Controller

This lecture does not have any notes. The content is meant to be consumed via video.

Encoding File Data

In this lecture, we encoded file data during form submission. File data contains complex information, which can't be transmitted via plain text. We must instruct the browser to encode files before submission by setting the `enctype` attribute to `multipart/form-data`.

```
<form enctype="multipart/form-data" method="POST" class="grid grid-cols-1 gap-6">
</form>
```

If you're using this attribute, the `method` attribute must be set to `POST`.

Next, you can view file data in PHP by using the superglobal `$_FILES` variable. This variable contains information on the files submitted with a form, such as the file size and name.

```
dd($_FILES);
```

Exercise: Creating a Receipt Service

This lecture does not have any notes. The content is meant to be consumed via video.

Resources

- PHP Filesystem Functions

Validating a File Upload

In this lecture, we validated that a file was uploaded. We can grab the file by using the `$_FILES` variables.

```
$receiptFile = $_FILES['receipt'] ?? null;
```

Next, we performed a conditional statement to check if the file was empty. We also checked if the file had any errors. PHP has a set of constants for various errors. In this case, we're using the `UPLOAD_ERR_OK` constant to make sure there are no errors.

```
if (!$file || $file['error'] !== UPLOAD_ERR_OK) {
  throw new ValidationException(['receipt' => ['Failed to upload file.']]);
}
```

Resources

- Upload Errors

Validating File Sizes

In this lecture, we validated the file size. First, we created a variable for the maximum file size. File sizes are stored in bytes. To get the file size in megabytes, we multiplied the megabyte size by 1024 to get the kilobyte size followed by another multiplication of 1024 to get the byte size.

```
$maxFileSizeMB = 3 * 1024 * 1024;
```

Lastly, we compared the file size of the file by checking the `$file['size']` variable. If the size exceeds the threshold, we threw an exception.

```
if ($file['size'] > $maxFileSizeMB) {
  throw new ValidationException(['receipt' => ['File upload is too large.']]);
}
```

Resources

- File Sizes

Validating Filenames

In this lecture, we validated the filename by using a regular expression. The filename can be grabbed with the `$file['name']` variable.

```
$originalFilename = $file['name'];
```

Next, we used the `preg_match()` function to compare a string against a pattern. The regular expression we've written allows alphanumeric characters along with `.`, `_`, `-`, and `\s` characters.

```
if (!preg_match('/^[A-Za-z0-9._-\s]+$/', $originalFilename)) {
  throw new ValidationException(['receipt' => ['Invalid filename']]);
}
```

Validating File Mime Types

In this lecture, we validated the mime type of a file. A mime type represents the file type. You can check out the resource section for a complete list of mime types. To grab the mime type, we can use the `$file['type']` variable. In this example, we're grabbing the mime type and comparing it against an array of mime types. If the mime type can't be found in the array, we'll throw an error.

```
$clientMimeType = $file['type'];
$allowedMimeTypes = ['image/jpeg', 'image/png', 'application/pdf'];

if (!in_array($clientMimeType, $allowedMimeTypes)) {
  throw new ValidationException(['receipt' => ['Invalid file type.']]);
}
```

Resources

- Mime Types

Generating a Random Filename

In this lecture, we generated a random filename to prevent files from conflicting with each other. First, we grabbed the file extension by using the `pathinfo()` function. This function accepts the filename and the information we'd like to extract. By passing in the `PATHINFO_EXTENSION` constant, this will instruct the function to grab the extension.

```
$fileExtension = pathinfo($originalFilename, PATHINFO_EXTENSION);
```

Next, we generated a filename with the `random_bytes()` function, which accepts the size of the value. Next, converted the bytes into hex so that it can be used as a filename. Lastly, we appended the extension.

```
$newFilename = bin2hex(random_bytes(16)) . "." . $fileExtension;
```

Moving Uploaded Files

In this lecture, we wrote the file to our server to store it. We can use the `move_uploaded_file()` function. There are two arguments. Firstly, we must provide the path to the file, which is stored in the `$file['tmp_name']` variable. Secondly, we must provide the new location.

```
if (!move_uploaded_file($file["tmp_name"], $uploadPath)) {
  throw new ValidationException(['receipt' => ['Failed to upload file']]);
}
```

If the function returns `false`, this means the file could not be moved. In that case, we threw an exception to inform the user of the failure.

Designing the Receipt Table

In this lecture, we designed and created the table for receipts. Most of what we talked about is already familiar to us. The most important concept we learned was being able to delete records with foreign keys by adding the `ON DELETE CASCADE` keywords. By adding these keywords, if the transaction associated with a receipt is deleted, the receipt will be deleted too.

```
FOREIGN KEY (`transaction_id`) REFERENCES `transactions` (`id`) ON DELETE CASCADE
```

Resources

- Receipt Entity

Storing the Receipt

This lecture does not have any notes. The content is meant to be consumed via video.

Displaying Receipts

This lecture does not have any notes. The content is meant to be consumed via video.

Validating the Download Request

This lecture does not have any notes. The content is meant to be consumed via video.

Downloading Files

In this lecture, we forced the user to download a receipt after visiting the route for downloading receipts. We configured the headers on the response by calling the `header()` function. First, we set the `Content-Disposition` header to `inline` to allow the browser to display the file in the browser. Alternatively, we can use `attachment` to completely force the user to download the file. Next, we set the `filename` property to configure the name of the file.

Lastly, we set the `Content-Type` to configure the media type.

```
header("Content-Disposition: inline; filename={$receipt['original_filename']}");
header("Content-Type: {$receipt['media_type']}");
```

Lastly, we attached the file to the body. Files can be read using the `readfile()` function. It accepts the path to the file.

```
readfile($filePath);
```

Deleting a Receipt

In this lecture, we deleted a file from our system. We can do so by using the `unlink()` function. This function accepts the path to the file to delete. In this example, we passed the path to the receipt.

```
$filePath = Paths::STORAGE_UPLOADS . "/" . $receipt['storage_filename'];

unlink($filePath);
```

Section 18: Everything Else

In this section, we covered various topics related to PHP development.

Magic Numbers

In this lecture, we talked about magic numbers. They're numbers holding a special meaning where their intention is immediately clear. In these cases, using constants to store magic numbers is beneficial. They provide the following benefits.

1. Describe the value stored in the constant.
2. Support for code completion in your editor.

We created a constant for our storing a magic number for the HTTP status code to redirect users.

```
class Http
{
  public const REDIRECT_STATUS_CODE = 302;
}
```

By doing so, we'll be able to use this constant in our `redirectTo()` function like so.

```
function redirectTo(string $path)
{
  header("Location: {$path}");
  http_response_code(Http::REDIRECT_STATUS_CODE);
  exit;
}
```

Destroying Session Cookies

In this lecture, we learned how to completely destroy session data. Instead of destroying a specific value in our session data with the `unset()` function, we can use the `session_destroy()` function to destroy all session data.

```
session_destroy();
```

In addition, we can expire a cookie using the `setcookie()` function. This function can be used for creating a cookie or modify an existing cookie. In this case, we're using it to change the expiration date of the `PHPSESSID` cookie generated by PHP.

```
$params = session_get_cookie_params();
setcookie(
  "PHPSESSID",
  '',
  time() - 3600,
  $params['path'],
  $params['domain'],
  $params['secure'],
  $params['httponly']
);
```

In this example, we're setting the `PHPSESSID` cookie to an empty value. Next, we're changing the expiration time by using the `time()` function to grab the current time and subtracting from it to get a date earlier than the current time. This informs the browser the cookie expired a long time ago.

Lastly, we're passing in the same values as the original cookie, which can be grabbed via the `session_get_cookie_params()` function.

Rendering a 404 Page

In this lecture, we rendered a 404 page. First, we created a property for storing the controller responsible for handling the response of a page not found.

```
private array $errorHandler;
```

Next, we defined a method responsible for dispatching the controller when a route couldn't be found.

```
private function dispatchNotFound(?Container $container)
{
  [$class, $func] = $this->errorHandler;

  $controllerInstance = $container ?
    $container->resolve($class) :
    new $class;

  $action = fn () => $controllerInstance->$func();

  foreach ($this->middlewares as $middleware) {
    $middlewareInstance = $container ?
      $container->resolve($middleware) :
      new $middleware;
    $action = fn () => $middlewareInstance->process($action);
  }

  $action();
}
```

It's the same code as for rendering a route, but it doesn't have parameters, nor are we looping through the routes. We're just immediately using the `errorHandler` property.

Lastly, we called the `setErrorHandler()` method to register a controller and method.

```
$app->setErrorHandler([ErrorController::class, 'notFound']);
```

Section 19: Deployment

In this section, we learned how to deploy a project to a managed server from Cloudways. There are no notes for this section.

Resources

- Cloudways
- HTAccess File
- PHP Engineer