

PHP

Module 1 Basics of WD and SQL

Module 2 Core PHP and OOPS

Module 3 Introduction of Laravel PHP Framework

Module 4 Database Operation-migration

Module 5 User inputs with Blade Template

Module 7 Practical Implementation with Project

Module 7 Controllers and Routes for URLs and APIs

HTML & CSS

- HTML is the main markup language for describing the structure of web pages.
- HTML stands for HyperText Markup Language. HTML is the basic building block of World Wide Web.
- Hypertext is text displayed on a computer or other electronic device with references to other text that the user can immediately access, usually by a mouse click or key press.
- Apart from text, hypertext may contain tables, lists, forms, images, and other presentational elements. It is an easy-to-use and flexible format to share information over the Internet.
- Markup languages use sets of markup tags to characterize text elements within a document, which gives instructions to the web browsers on how the document should appear.



What You Can Do with HTML

- There are lot more things you can do with HTML.
 - You can publish documents online with text, images, lists, tables, etc.
 - You can access web resources such as images, videos or other HTML document via hyperlinks.
 - You can create forms to collect user inputs like name, e-mail address, comments, etc.
 - You can include images, videos, sound clips, flash movies, applications and other HTML documents directly inside an HTML document.
 - You can create offline version of your website that work without internet.
 - You can store data in the user's web browser and access later on.
 - You can find the current location of your website's visitor.

Step 1: Creating the HTML file

- Step 2: Type some HTML code

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <title>A simple HTML document</title>
    </head>

    <body>
        <p>Hello World!<p>
    </body>

</html>
```

HTML Elements

- **HTML Element Syntax**
- An HTML element is an individual component of an HTML document. It represents semantics, or meaning. For example, the `title` element represents the title of the document.
- Most HTML elements are written with a *start tag* (or opening tag) and an *end tag* (or closing tag), with content in between. Elements can also contain attributes that defines its additional properties. For example, a paragraph, which is represented by the `p` element, would be written as:

The diagram illustrates the structure of an HTML paragraph element (`<p>`). A pink arrow labeled "Start tag" points to the opening tag `<p`. A green arrow labeled "Attribute" points to the attribute `class="foo"`. A red arrow labeled "Content" points to the text "This is a paragraph.". A pink arrow labeled "End tag" points to the closing tag `>`.

```
<p class="foo"> This is a paragraph. </p>
```

HTML Styles

Styling HTML Elements

- HTML is quite limited when it comes to the presentation of a web page. It was originally designed as a simple way of presenting information. [CSS \(Cascading Style Sheets\)](#) was introduced in December 1996 by the [World Wide Web Consortium \(W3C\)](#) to provide a better way to style HTML elements.

Adding Styles to HTML Elements

- Style information can either be attached as a separate document or embedded in the HTML document itself. These are the three methods of implementing styling information to an HTML document.

- **Inline styles** — Using the `style` attribute in the HTML start tag.
- **Embedded style** — Using the `<style>` element in the head section of the document.
- **External style sheet** — Using the `<link>` element, pointing to an external CSS files.

HTML Styles

- **Inline Styles**
- Inline styles are used to apply the unique style rules to an element, by putting the CSS rules directly into the start tag. It can be attached to an element using the `style` attribute.
- The `style` attribute includes a series of CSS property and value pairs. Each `property: value` pair is separated by a semicolon (`;`), just as you would write into an embedded or external style sheet. But it needs to be all in one line i.e. no line break after the semicolon.
- The following example demonstrates how to set the `color` and `font-size` of the text:
 - `<h1 style="color:red; font-size:30px;">This is a heading</h1>`

HTML Styles

- **External Style Sheets**

- An external style sheet is ideal when the style is applied to many pages.
- An external style sheet holds all the style rules in a separate document that you can link from any HTML document on your site. External style sheets are the most flexible because with an external style sheet, you can change the look of an entire website by updating just one file.
- You can attach external style sheets in two ways — *linking* and *importing*:

- **Linking External Style Sheets**

- An external style sheet can be linked to an HTML document using the `<link>` tag.
- The `<link>` tag goes inside the `<head>` section, as shown here:

```
<head> <link rel="stylesheet" href="css/style.css"> </head>
```

HTML Images

• Inserting Images into Web Pages

- Images enhance visual appearance of the web pages by making them more interesting and colorful.
- The `` tag is used to insert images in the HTML documents. It is an empty element and contains attributes only. The syntax of the `` tag can be given with:
``
- **The following example inserts three images on the web page:**
- ``
- ``
- ``

HTML Tables

• Creating Tables in HTML

- HTML table allows you to arrange data into rows and columns. They are commonly used to display tabular data like product listings, customer's details, financial reports, and so on.
- You can create a table using the `<table>` element. Inside the `<table>` element, you can use the `<tr>` elements to create rows, and to create columns inside a row you can use the `<td>` elements. You can also define a cell as a header for a group of table cells using the `<th>` element.

HTML Table (Default Style)

No.	Name	Age
1	Peter Parker	16
2	Clark Kent	34
3	Harry Potter	11

HTML Lists

• Working with HTML Lists

HTML lists are used to present list of information in well formed and semantic way. There are three different types of list in HTML and each one has a specific purpose and meaning.

- **Unordered list** — Used to create a list of related items, in no particular order.
 - **Ordered list** — Used to create a list of related items, in a specific order.
 - **Description list** — Used to create a list of terms and their descriptions.
-
- In the following sections we will cover all the three types of list one by one:

HTML Forms

- **What is HTML Form**

- HTML Forms are required to collect different kinds of user inputs, such as contact details like name, email address, phone numbers, or details like credit card information, etc.
- Forms contain special elements called controls like inputbox, checkboxes, radio-buttons, submit buttons, etc. Users generally complete a form by modifying its controls e.g. entering text, selecting items, etc. and submitting this form to a web server for further processing.
- The `<form>` tag is used to create an HTML form. Here's a simple example of a login form:

Username:

Password:

Bootstrap 4

- **What is Bootstrap?**

- Bootstrap is a free front-end framework for faster and easier web development
- Bootstrap includes HTML and CSS based design templates for typography, forms, buttons, tables, navigation, modals, image carousels and many other, as well as optional JavaScript plugins
- Bootstrap also gives you the ability to easily create responsive designs

Why Use Bootstrap?

Advantages of Bootstrap:

- Easy to use: Anybody with just basic knowledge of HTML and CSS can start using Bootstrap
- Responsive features: Bootstrap's responsive CSS adjusts to phones, tablets, and desktops
- Mobile-first approach: In Bootstrap, mobile-first styles are part of the core framework
- Browser compatibility: Bootstrap 4 is compatible with all modern browsers (Chrome, Firefox, Internet Explorer 10+, Edge, Safari, and Opera)

Bootstrap 4

• Where to Get Bootstrap 4?

There are two ways to start using Bootstrap 4 on your own web site.

You can:

- Include Bootstrap 4 from a CDN
- Download Bootstrap 4 from getbootstrap.com

• Bootstrap 4 CDN

- If you don't want to download and host Bootstrap 4 yourself, you can include it from a CDN (Content Delivery Network).
- MaxCDN provides CDN support for Bootstrap's CSS and JavaScript. You must also include jQuery:

Bootstrap 4

```
<!-- Latest compiled and minified CSS -->

<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">

<!-- jQuery library -->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

<!-- Popper JS -->

<script
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>

<!-- Latest compiled JavaScript -->

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
```

Bootstrap 4

• Fixed Container

- Use the `.container` class to create a responsive, fixed-width container.
- Note that its width (`max-width`) will change on different screen sizes:
- Open the example below and resize the browser window to see that the container width will change at different breakpoints:

```
<div class="container">  
  <h1>My First Bootstrap Page</h1>  
  <p>This is some text.</p>  
</div>
```

	Extra small <576px	Small ≥576px	Medium ≥768px	Large ≥992px	Extra large ≥1200px
max-width	100%	540px	720px	960px	1140px

Bootstrap 4

• Fluid Container

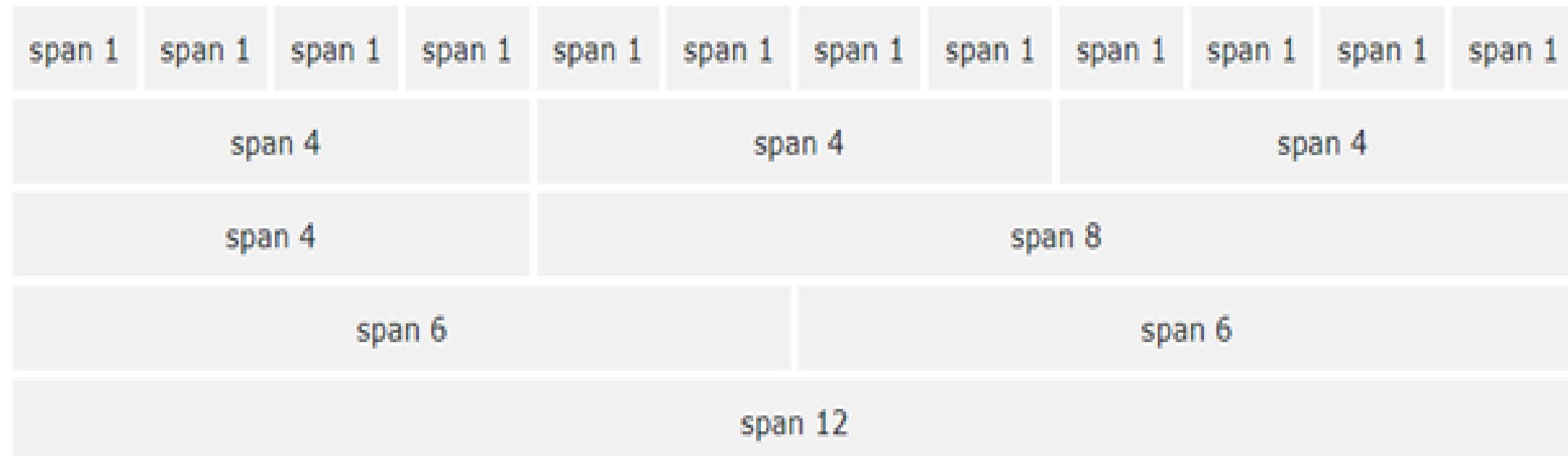
- Use the `.container-fluid` class to create a full width container, that will always span the entire width of the screen (width is always 100%):

```
<div class="container-fluid">
  <h1>My First Bootstrap Page</h1>

  <p>This is some text.</p>
</div>
```

Bootstrap 4 Grid System

- Bootstrap's grid system is built with flexbox and allows up to 12 columns across the page.
- If you do not want to use all 12 columns individually, you can group the columns together to create wider columns:



Bootstrap 4 Tables

- **Bootstrap 4 Basic Table**
- A basic Bootstrap 4 table has a light padding and horizontal dividers.

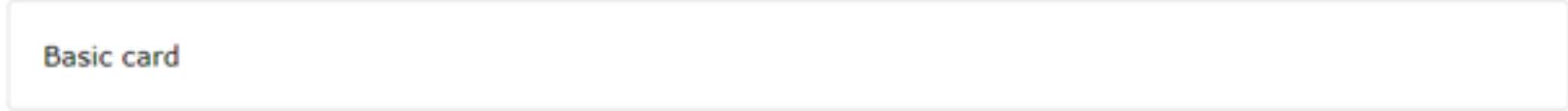
Class	Description
.table-active	Grey: Applies the hover color to the table row or table cell
.table-secondary	Grey: Indicates a slightly less important action
.table-light	Light grey table or table row background
.table-dark	Dark grey table or table row background
5 more rows	

Bootstrap 4 Cards

- **Cards**

- A card in Bootstrap 4 is a bordered box with some padding around its content. It includes options for headers, footers, content, colors, etc.

- **Basic Card**



Basic card

card has a .card-

Bootstrap 4 Forms

- **Bootstrap 4's Default Settings**

- Form controls automatically receive some global styling with Bootstrap:
- All textual `<input>`, `<textarea>`, and `<select>` elements with class `.form-control` have a width of 100%.

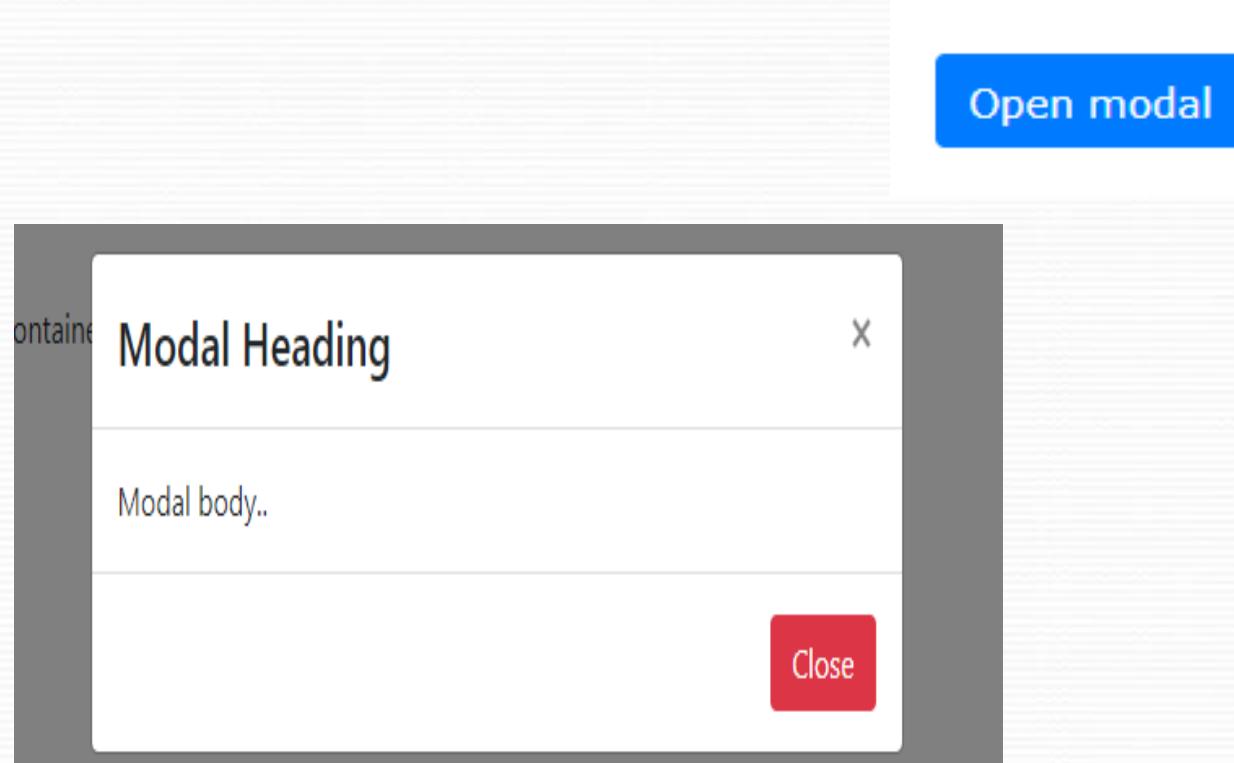
- **Bootstrap 4 Form Layouts**

- Bootstrap provides two types of form layouts:
 - Stacked (full-width) form
 - Inline form

The image shows a user interface for a login or sign-up form. It features a 'Email:' label above a text input field containing 'Enter email'. Below it is a 'Password:' label above another text input field containing 'Enter password'. A 'Remember me' checkbox is positioned below the password field. At the bottom right is a blue 'Submit' button.

Bootstrap 4 Modal

- The Modal component is a dialog box/popup window that is displayed on top of the current page:



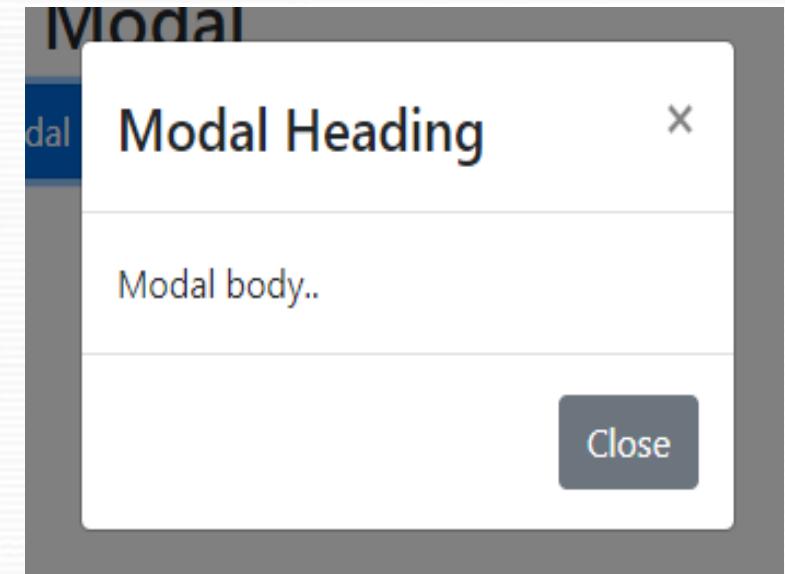
Bootstrap 4 Modal

- **Modal Size**

- Change the size of the modal by adding the `.modal-sm` class for small modals, `.modal-lg` class for large modals, or `.modal-xl` for extra large modals.
- Add the size class to the `<div>` element with class `.modal-dialog`:

- **Small Modal**

- `<div class="modal-dialog modal-sm">`

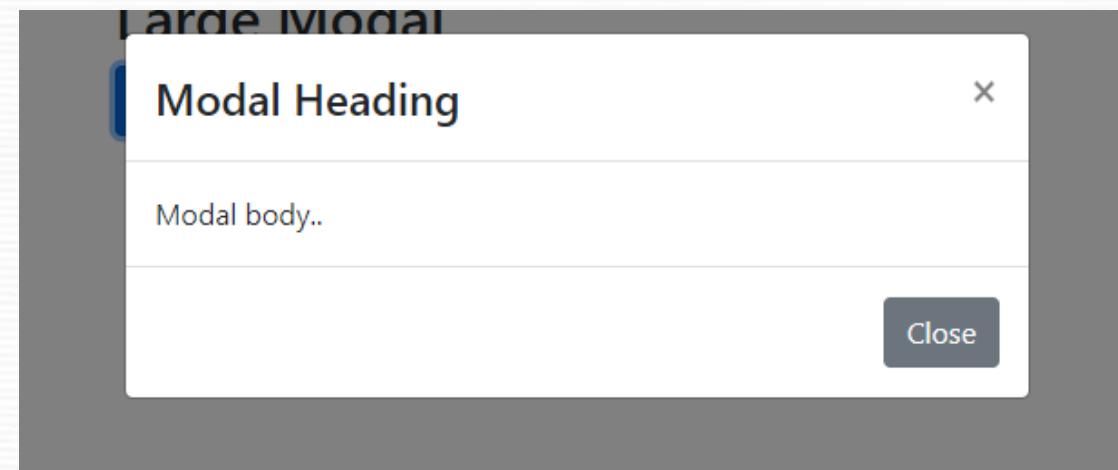


Bootstrap 4 Modal

- **Modal Size**

- Change the size of the modal by adding the `.modal-sm` class for small modals, `.modal-lg` class for large modals, or `.modal-xl` for extra large modals.
- Add the size class to the `<div>` element with class `.modal-dialog`:

- **Large Modal**



Bootstrap 4 Modal

- **Extra Large Modal**

```
<div class="modal-dialog modal-xl">
```

- **Centered Modal**

Center the modal vertically and horizontally within the page, with the `.modal-dialog-centered` class:

```
<div class="modal-dialog modal-dialog-centered">
```

- **Scrolling Modal**

```
<div class="modal-dialog modal-dialog-scrollable">
```

Database

- **Database** is a collection of related data and data is a collection of facts and figures that can be processed to produce information.
- Mostly data represents recordable facts. Data aids in producing information, which is based on facts. For example, if we have data about marks obtained by all students, we can then conclude about toppers and average marks.
- A database management system stores data in such a way that it becomes easier to retrieve, manipulate, and produce information.
- Characteristics
- Real-world entity
- Relation-based tables
- Isolation of data and application
- Less redundancy
- Consistency
- Multiple views
- Security

Database

Constraints

- terms that needs to be satisfied on data
- For ex all students must have unique roll number
- Can be defined as primary key, foreign key, unique key etc
- **Primary key** – column of table whose value can be used to uniquely identify records **Foreign key** – column inside table that is primary key of another table

Database

- **Unique key** – like primary key can be used to uniquely identify a record
- Difference between primary key and unique key is primary key will never allow null whereas unique key will allow it once
- **Normalization**
 - Normalization is the process of organizing the data in the database.

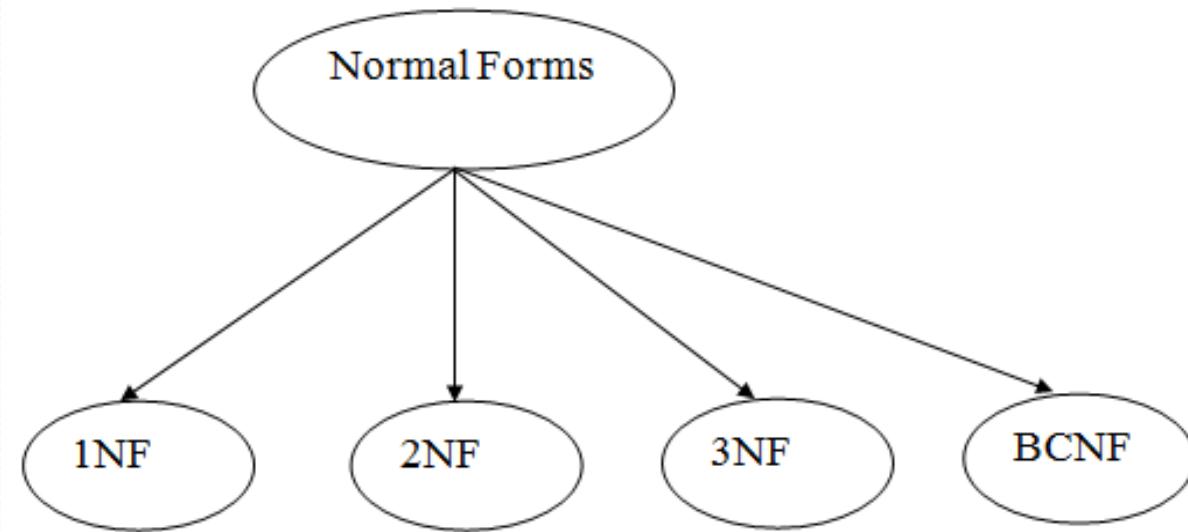
Database

- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- The normal form is used to reduce redundancy from the database table.

Database

- Normalization
- Types of Normal Forms

There are the four types of normal forms:



Database

- **Normalization**

Normal Form

- **1NF**

A relation is in 1NF if it contains an atomic value.

- **2NF**

A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.

- **3NF**

A relation will be in 3NF if it is in 2NF and no transition dependency exists..

SQL

- SQL is followed by a unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQL by listing all the basic SQL Syntax.
- All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and all the statements end with a semicolon (;).
- The most important point to be noted here is that SQL is case insensitive, which means SELECT and select have same meaning in SQL statements. Whereas, MySQL makes difference in table names. So, if you are working with MySQL, then you need to give table names as they exist in the database

SQL

- **Various Syntax in SQL**

All the examples given in this tutorial have been tested with a MySQL server.

- **SQL SELECT Statement**

```
SELECT column1, column2....columnN  
FROM    table_name;
```

- **SQL DISTINCT Clause**

```
SELECT DISTINCT column1, column2....columnN  
FROM    table_name;
```

SQL

- **SQL WHERE Clause**

```
SELECT column1, column2....columnN  
FROM    table_name  
WHERE   CONDITION;
```

- **SQL AND/OR Clause**

```
SELECT column1, column2....columnN  
FROM    table_name  
WHERE   CONDITION-1 {AND|OR} CONDITION-2;
```

- **SQL IN Clause**

```
SELECT column1, column2....columnN  
FROM    table_name  
WHERE   column_name IN (val-1, val-2,...val-N);
```

SQL

- **SQL BETWEEN Clause**

```
SELECT column1, column2....columnN  
FROM    table_name  
WHERE   column_name BETWEEN val-1 AND val-2;
```

- **SQL LIKE Clause**

```
SELECT column1, column2....columnN  
FROM    table_name  
WHERE   column_name LIKE { PATTERN } ;
```

SQL

- **SQL ORDER BY Clause**

```
SELECT column1, column2....columnN  
FROM    table_name  
WHERE   CONDITION  
ORDER BY column_name {ASC|DESC};
```

- **SQL GROUP BY Clause**

```
SELECT SUM(column_name)  
FROM    table_name  
WHERE   CONDITION  
GROUP BY column_name;
```

SQL

- **SQL COUNT Clause**

```
SELECT COUNT(column_name)  
FROM table_name  
WHERE CONDITION;
```

- **SQL HAVING Clause**

```
SELECT SUM(column_name)  
FROM table_name  
WHERE CONDITION  
GROUP BY column_name  
HAVING (arithmetic function condition);
```

SQL

- **SQL CREATE TABLE Statement**

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    .....
    columnN datatype,
    PRIMARY KEY( one or more columns )
);
```

- **SQL DROP TABLE Statement**

- ```
DROP TABLE table_name;
```

# SQL

- **SQL CREATE INDEX Statement**

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2,...columnN);
```

- **SQL DROP INDEX Statement**

```
ALTER TABLE table_name
DROP INDEX index_name;
```

- **SQL DESC Statement**

```
DESC table_name;
```

- **SQL TRUNCATE TABLE Statement**

```
TRUNCATE TABLE table_name;
```

# SQL

- **SQL ALTER TABLE Statement**

```
ALTER TABLE table_name {ADD|DROP|MODIFY} column_name {data_type};
```

- **SQL ALTER TABLE Statement (Rename)**

```
ALTER TABLE table_name RENAME TO new_table_name;
```

- **SQL INSERT INTO Statement**

```
INSERT INTO table_name(column1, column2....columnN)
```

```
VALUES (value1, value2....valueN);
```

- **SQL UPDATE Statement**

```
UPDATE table_name
```

```
SET column1 = value1, column2 = value2....columnN=valueN
```

```
[WHERE CONDITION];
```

# SQL

- **SQL DELETE Statement**

```
DELETE FROM table_name
WHERE {CONDITION};
```

- **SQL CREATE DATABASE Statement**

```
CREATE DATABASE database_name;
```

- **SQL DROP DATABASE Statement**

```
DROP DATABASE database_name;
```

- **SQL USE Statement**

```
USE database_name;
```

# SQL

- **SQL DELETE Statement**

```
DELETE FROM table_name
WHERE {CONDITION};
```

- **SQL CREATE DATABASE Statement**

```
CREATE DATABASE database_name;
```

- **SQL DROP DATABASE Statement**

```
DROP DATABASE database_name;
```

- **SQL USE Statement**

```
USE database_name;
```

# Module 2

## Core PHP and OOPS

# Introduction to PHP

- What
- PHP stand for hypertext preprocessor. PHP old name is Personal Home Page
- PHP code may be embedded into HTML code, or it can be used in combination with various web template systems, web content management system and web frameworks.

# Introduction to PHP

## Why PHP

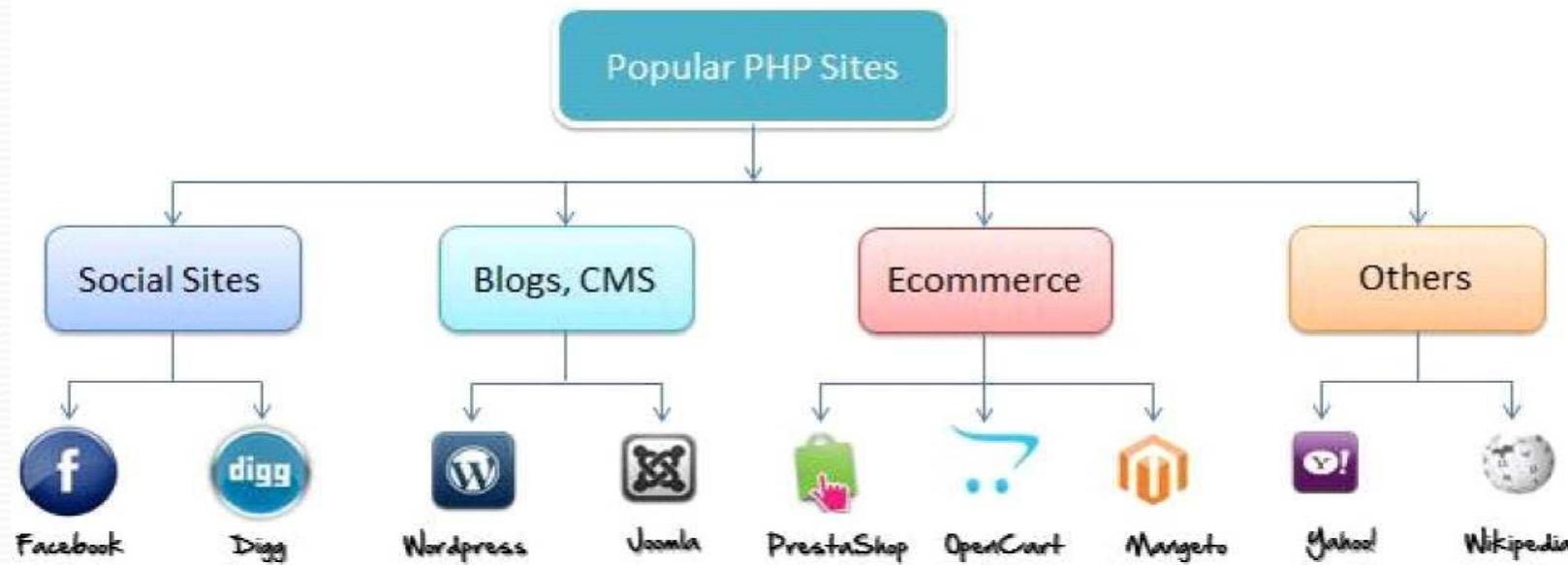
- PHP is **open source and free.**
- Short learning curve compared to other languages such as JSP, ASP etc.
- Large community document
- Most web hosting servers support PHP by default unlike other languages such as

ASP that need IIS. This makes PHP a cost effective choice.

- PHP is regular updated to keep abreast with the latest technology trends.
- Other benefit that you get with PHP is that it's a **server side scripting language**; this means you only need to install it on the server and client computers requesting for resources from the server do not need to have PHP installed; only a web browser would be enough.
- PHP has **in built support for working hand in hand with MySQL**; this doesn't mean you can't use PHP with other database management systems

# Introduction to PHP

- What is PHP used for & Market Place
- In terms of market share, there are over 20 million websites and application on the internet developed using PHP scripting language.



# Features of PHP

- **Scalar Type Hints & Return Types:** PHP7 will allow developers to declare what kind of return type function is expected to return a value. It is almost similar as Type hinting parameters as below.
- **Spaceship Operator:** PHP7 will introduce a new operator called spaceship operator (<= >) otherwise called combined comparison operator. It can be used mostly in sorting and combined comparison. It works like strcmp() or version\_compare() or ?? added or null.
- **Null Coalesce Operator :** The coalesce operator to the core, which returns the first operand if exists

# Features of PHP

- **Facilitates Error Handling:** The new **Engine Exceptions** will allow you to replace these kind of errors with exceptions. The new **\EngineException** objects don't extend the **\Exception** BaseClass. This ensures backward compatibility and results in two different kinds of exceptions in error
- handling: **traditional** and **engine exceptions**
- **Adds Anonymous Classes:** An anonymous class is a class without a name. The object it instantiates has the same functionality as an object of a named class. If anonymous classes are used well, they can **speed up coding as well execution time**
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.1PHPIintro/3.1.1%20PHP7.php>

# PHP Syntax

- The script starts with `<?php` and ends with `?>`. These tags are also called ‘Canonical PHP tags’.
- Every PHP command ends with a semi-colon (`;`). Let’s look at the *hello world* program in PHP:

## Syntax:

```
<?php # Here echo command is used to print ?>
```

- **Refer Example Link:**

- <https://github.com/TopsCode/PHP/tree/master/Module3/3.1PHPIntro/3.1.2%20PHPSyntax.php>

## PHP Variables

**What:** a variable does not need to be declared before adding a value to it. PHP automatically converts the variable to the correct data type, depending on its value.

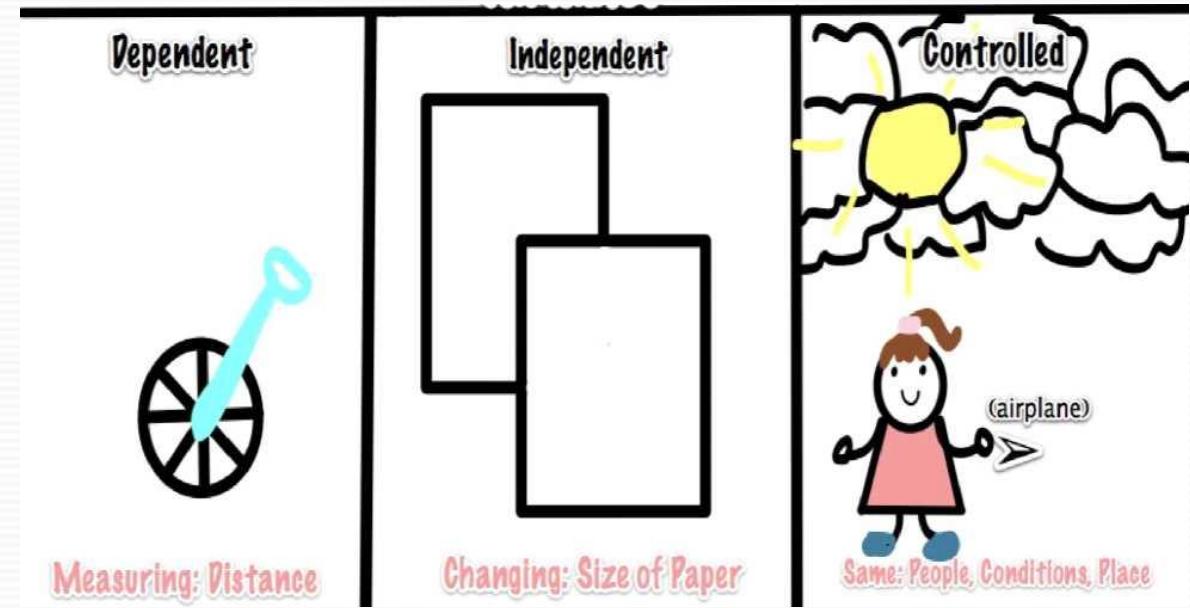
- When a variable is set it can be used over and over again in your script

## Rules Of Variables

Names (also called identifiers)

Must always begin with a dollar-sign (\$)

generally start with a letter and can contain letters, numbers, and underscore characters — I



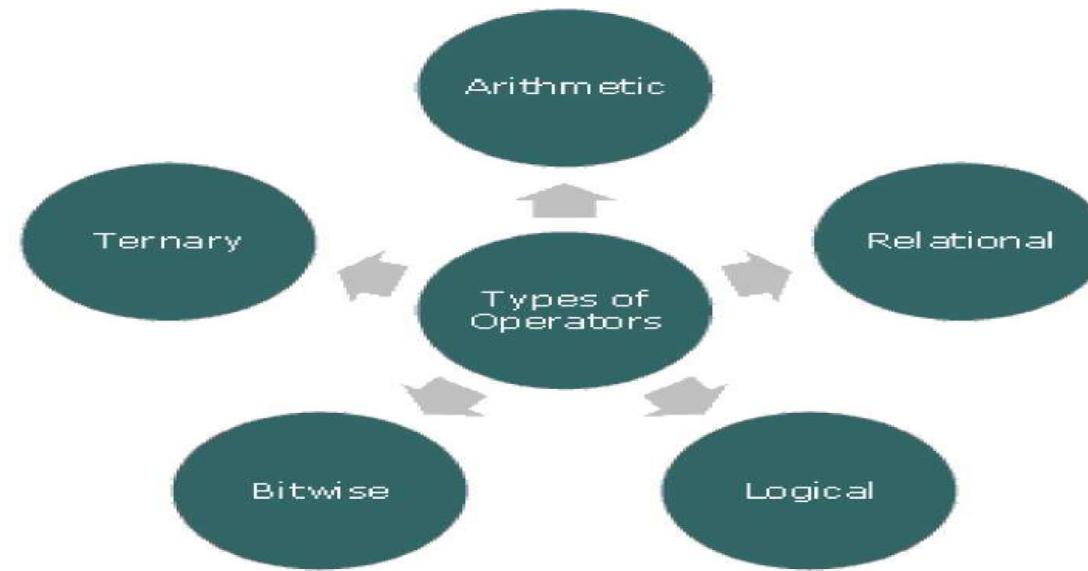
### Refer Example Link:

<https://github.com/TopsCode/PHP/tree/master/Module3/3.2Variable/3.2.1%20Variable.php>

# Variables

- Names are case sensitive
- Values can be numbers, strings, Boolean, etc.
- change as the program executes
- **PHP is a Loosely Typed Language**
- In PHP a variable does not need to be declared before being set.
- In the example above, you see that you do not have to tell PHP which data type the variable is.
- PHP automatically converts the variable to the correct data type, depending on how they are set.
- In a strongly typed programming language, you have to declare (define) the type and name of the variable before using it.
- In PHP the variable is declared automatically when you use it.

# PHP Operators



**What:** An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

# Arithmetic Operators

| Operator | Description                  | Example      | Result   |
|----------|------------------------------|--------------|----------|
| +        | Addition                     | $2+2$        | 4        |
| -        | Subtraction                  | $x=2 \ 5-x$  | 3        |
| *        | Multiplication               | $x=4 \ x*5$  | 20       |
| /        | Division                     | $15/5 \ 5/2$ | 3<br>2.5 |
| %        | Modulus (division remainder) | $10 \% 8$    | 2        |
| ++       | Increment                    | $x=5 \ X++$  | $x=6$    |
| -        | Decrement                    | $x=5 \ Y--$  | $x=4$    |

## Refer Example Link:

<https://github.com/TopsCode/PHP/tree/master/Module3/3.3Operators/3.3.1%20Arithmeti c%20Operators.php>

# Assignment Operators

| Operator | Example | Is The Same As |
|----------|---------|----------------|
| =        | $x=y$   | $x=y$          |
| $+=$     | $x+=y$  | $x=x+y$        |
| $-=$     | $x-=y$  | $x=x-y$        |
| $*=$     | $x*=y$  | $x=x*y$        |
| $/=$     | $x/=y$  | $x=x/y$        |
| $,=$     | $x.=y$  | $x=x.y$        |
| $%=$     | $x\%=y$ | $x=x\%y$       |

# Comparison Operators

| Operator            | Description                 | Example                               |
|---------------------|-----------------------------|---------------------------------------|
| <code>= =</code>    | is equal to                 | <code>5 == 8</code> returns false     |
| <code>!=</code>     | is not equal                | <code>5 != 8</code> returns true      |
| <code>&gt;</code>   | is greater than             | <code>5 &gt; 8</code> returns false   |
| <code>&lt;</code>   | is less than                | <code>5 &lt; 8</code> returns true    |
| <code>:&gt;=</code> | is greater than or equal to | <code>5 :&gt;= 8</code> returns false |
| <code>&lt;=</code>  | is less than or equal to    | <code>5 &lt;= 8</code> returns true   |

**Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module3/3.3Operators/3.3.2%20Assignment%20Operators.php>

# Logical Operators

| Operator | Description | Example                                            |
|----------|-------------|----------------------------------------------------|
| &B.      | and         | x=6<br>y=3<br>$(x < 10 \&& y > 1)$ returns true    |
|          | or          | x=6<br>y=3<br>$(x==5 \mid\mid y==5)$ returns false |
| !        | not         | x=6<br>y=3<br>$!(x==y)$ returns true               |

**Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.3Operators/3.3.3%20Logical%20Operators.php>

# Global Variables

- **What:** A global variable is a programming language construct, a variable type that is declared outside any function and is accessible to all functions throughout the program
- The PHP super global variables are:
  - `$GLOBALS`
  - `$ SERVER`
  - `$_REQUEST`
  - `$_POST`
  - `$_GET`
  - `$_FILES`
  - `$_ENV`
  - `$_COOKIE`
  - `$_SESSION`

# Global Variables

- **GLOBALS**
- **What:** \$GLOBALS is a PHP super global variable which is used to access global variables from anywhere in the PHP script (also from within functions or methods).
- PHP stores all global variables in an array called \$GLOBALS[index]. The index holds the name of the variable.
- The example below shows how to use the super global variable \$GLOBALS

**Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.4SuperGlobal/3.4.1%20Globals.php>

# PHP Global Variables

## **`$_SERVER`**

- **What:** `$_SERVER` is a PHP super global variable which holds information about headers, paths, and script locations.
- The example below shows how to use some of the elements in `$_SERVER`:
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/tree/master/Module3/3.4SuperGlobal/3.4.2%20Server.php>

# \$\_SERVER

|                                             |                                                                                                       |
|---------------------------------------------|-------------------------------------------------------------------------------------------------------|
| <code>\$_SERVER['PHP_SELF']</code>          | Returns the filename of the currently executing script                                                |
| <code>\$_SERVER['GATEWAY_INTERFACE']</code> | Returns the version of the Common Gateway Interface (CGI) the server is using                         |
| <code>\$_SERVER['SERVER_ADDR']</code>       | Returns the IP address of the host server                                                             |
| <code>\$_SERVER['SERVER_NAME']</code>       | Returns the name of the host server (such as <a href="http://www.tops-int.com">www.tops-int.com</a> ) |
| <code>\$_SERVER['SERVER_SOFTWARE']</code>   | Returns the server identification string (such as Apache/2.2.24)                                      |
| <code>\$_SERVER['SERVER_PROTOCOL']</code>   | Returns the name and revision of the information protocol (such as HTTP/1.1)                          |

# \$\_SERVER

|                                            |                                                                                            |
|--------------------------------------------|--------------------------------------------------------------------------------------------|
| <code>\$_SERVER['REQUEST_METHOD']</code>   | Returns the request method used to access the page (such as POST)                          |
| <code>\$_SERVER['REQUEST_TIME']</code>     | Returns the timestamp of the start of the request<br>(such as<br>1377687496)               |
| <code>\$_SERVER['QUERY_STRING']</code>     | Returns the query string if the page is accessed via a query string                        |
| <code>\$_SERVER['HTTP_ACCEPT']</code>      | Returns the Accept header from the current request                                         |
| <code>\$_SERVER['SERVER_SIGNATURE']</code> | Returns the server version and virtual host name which are added to server-generated pages |
| <code>\$_SERVER['PATH_TRANSLATED']</code>  | Returns the file system based path to the current script                                   |
| <code>\$_SERVER['SCRIPT_NAME']</code>      | Returns the path of the current script                                                     |
| <code>\$_SERVER['SCRIPT_URI']</code>       | Returns the URI of the current page                                                        |

# \$\_SERVER

| Element / Code                   | Description                                                                                                                                                                                                                                                     |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \$_SERVER['HTTP_ACCEPT_CHARSET'] | Returns the Accept_Charset header from the current request(such as utf-8,ISO-8859-1)                                                                                                                                                                            |
| \$_SERVER['HTTP_HOST']           | Returns the Host header from the current request                                                                                                                                                                                                                |
| \$_SERVER['HTTP_REFERER']        | Returns the complete URL of the current page (not reliable because not all user-agents support it)                                                                                                                                                              |
| \$_SERVER['HTTP S']              | Is the script queried through a secure HTTP Protocol                                                                                                                                                                                                            |
| \$_SERVER['REMOTE_ADDR']         | Returns the IP address from where the user is viewing the current page                                                                                                                                                                                          |
| \$_SERVER['REMOTE_HOST']         | Returns the Host name from where the user is viewing the current page                                                                                                                                                                                           |
| \$_SERVER['REMOTE_PORT']         | Returns the port being used on the user's machine to communicate with the web server                                                                                                                                                                            |
| \$_SERVER['SCRIPT_FILENAME']     | Returns the absolute pathname of the currently executing script                                                                                                                                                                                                 |
| \$_SERVER['SERVER_ADMIN']        | Returns the value given to the SERVER ADMIN directive in the web server configuration file (if your script runs on a virtualhost, it will be the value defined for that virtual host) (such as <a href="mailto:someone@tops-int.com">someone@tops-int.com</a> ) |
| \$_SERVER['SERVER_PORT']         | Returns the port on the server machine being used by the webserver for communication (such as 80)                                                                                                                                                               |

# \$\_COOKIE

- Cookies are small text files loaded from a server to a client computer storing some
- information regarding the client computer, so that when the same page from the server is visited by the user, necessary information can be collected from the cookie itself, decreasing the latency to open the page
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.4SuperGlobal/3.4.3%20Cookie.php>

# \$\_SESSION

- Sessions are wonderful ways to pass variables. All you need to do is start a session by `session_start();` Then all the variables you store within a `$_SESSION`, you can access it from anywhere in the server
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.4SuperGlobal/3.4.4%20Session.php>

# \$\_FILES

- \$\_FILES is a super global variable which can be used to upload files. Here we will see an example in which our php script checks if the form to upload the file is being submitted and generates an message if true
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.4SuperGlobal/3.4.5%20Files.php>

# **\$\_GET**

- The `$_GET` variable is used to collect values from a form with `method="get"`.
- **The `$_GET` Variable**
- The `$_GET` variable is an array of variable names and values sent by the HTTP GET method.
- The `$_GET` variable is used to collect values from a form with `method="get"`. Information sent from a form with the GET method is visible to everyone (it will be displayed in the browser's address bar) and it has limits on the amount of information to send (max. 100 characters).
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.4SuperGlobal/3.4.6%20Get.php>

# **\$\_POST**

- The `$_POST` variable is used to collect values from a form with `method="post"`.
- The `$_POST` variable is an array of variable names and values sent by the HTTP POST method.
- The `$_POST` variable is used to collect values from a form with `method="post"`. Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send.
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.4SuperGlobal/3.4.7%20Post.php>

# The If...Else Statement

- If you want to execute some code if a condition is true and another code if a condition is false, use the if....else statement
- PHP If & Elseif Statements
- If you want to execute some code if one of several conditions are true use the elseif statement
- **Syntax:**

```
if (condition)
code to be executed if condition is true;
elseif(condition)
code to be executed if condition is true;
else
code to be executed if condition is false;
```

- **Refer Example Link:** <https://github.com/TopsCode/PHP/blob/master/Module3/3.5IFCondition/3.5.1IF.php>

# PHP Switch Statement

- The Switch statement in PHP is used to perform one of several different actions based on one of several different conditions.
- If you want to select one of many blocks of code to be executed, use the Switch statement.
- The switch statement is used to avoid long blocks of if..elseif..else code.

# PHP Switch Statement

## Syntax

```
switch (expression)
{
 case label1:
 code to be executed if expression = label1;
 break;
 case label2:
 code to be executed if expression = label2;
 break;
 default:
 code to be executed
}
```

## Refer Example Link:

<https://github.com/TopsCode/PHP/blob/master/Module3/3.6SwitchCase/3.6.1%20Switch.php>

# Loop

- Why do we want loops in our code?
- Do something for a given number of times or for every object in a collection of objects
- for every radio button in a form, see if it is checked
- for every month of the year, charge \$100 against the balance
- calculate the sum of all the numbers in a list
- Many loops are counting loops
- they do something a certain number of times

# While Loop

- The while loop will execute a block of code **if and as long as a condition is true.**

## Syntax

```
while (condition)
```

```
code to be executed;
```

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module3/3.7Loops/3.7.1%20While.php>

# Do..while Loop

- The do...while loop will execute a block of code **at least once**-it then will repeat the loop **as long as a condition is true**.

## Syntax:

```
do{
 code to be executed;
} while (condition);
```

## Refer Example Link:

<https://github.com/TopsCode/PHP/blob/master/Module3/3.7Loops/3.7.2%20Dowhile.php>

# For Loop

The for loop is used when you know how many times you want to execute a statement or a list of statements.

**Syntax**

```
for (initialization; condition; increment){
 code to be executed;
}
```

## Refer Example Link:

<https://github.com/TopsCode/PHP/blob/master/Module3/3.6%20Loops/4.6.1%20For.php>

# Foreach Loop

- The for each loop is used to loop through arrays.
- For every loop, the value of the current array element is assigned to \$value (and the array pointer is moved by one) -so on the next loop, you'll be looking at the next element.

## Syntax

```
foreach(array as value){
 code to be executed;
}
```

- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.7Loops/3.7.4%20Foreach.php>

# Difference between echo and print

- **Speed.** There is a difference between the two, but speed-wise it should be irrelevant which one you use. echo is marginally faster
- **Expression.** print() behaves like a function where as echo behaves like a statement in that you can do
- **Parameter(s).** The grammar is: echo expression [, expression[,expression] ... ] But echo ( expression, expression ) is not valid. So, echo without parentheses can take multiple parameters, which get concatenated

# Types Of Errors

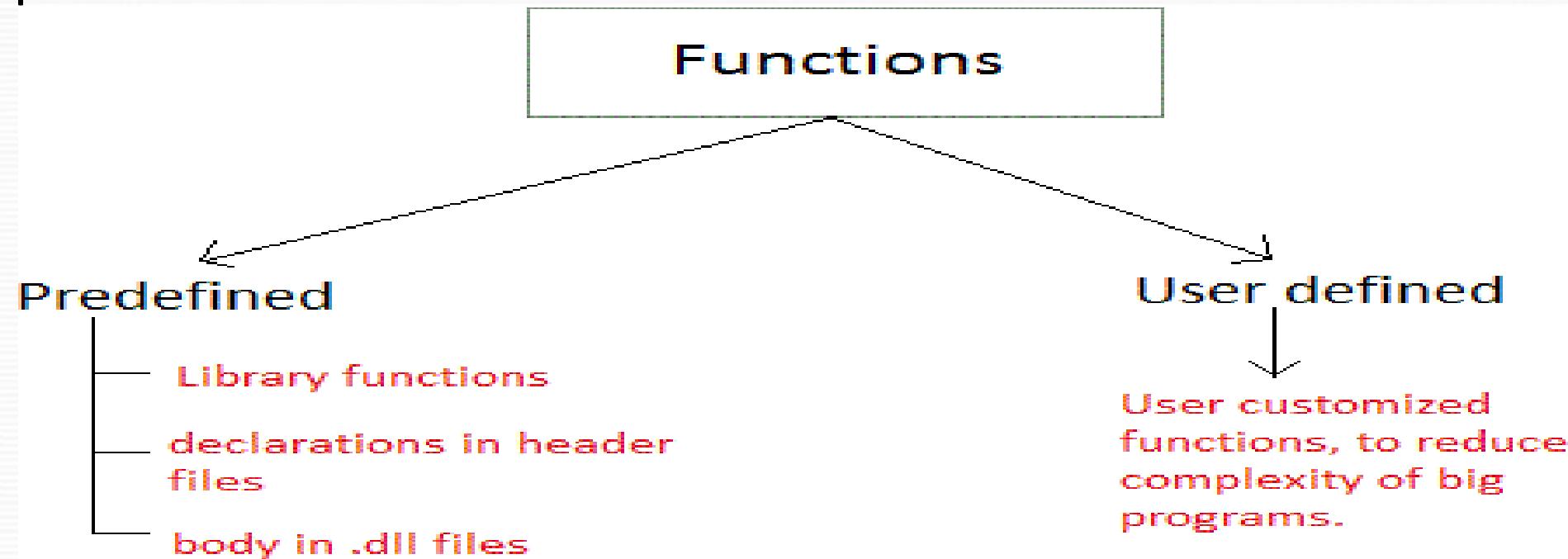
- **There are 4 types of errors in PHP**
  - Fatal Error
  - Notice Error
  - Parse Error
  - Warning Error
- **Fatal Error(Critical error)**:-An object of a non-existent class, or calling a non-existent function. These errors cause the immediate termination of the script.
- **Notice Error**:-These are trivial, non-critical errors . Accessing a variable that has not yet been defined.but they do not termination script .

# Types Of Errors

- **Parse error (Syntax error)**:- When we make mistake in PHP code like, missing semicolon or any unexpected symbol in code.Stop Script Execution.
- **Warning Error(Most Serious error)**:-To include() a file which does not exist,but they do not termination script

# PHP Functions

- **What:** A function is a self-contained block of code that performs a specific task.



# PHP Functions -Adding parameters

- Our first function (`test()`) is a very simple function. It only writes a static string.
- To add more functionality to a function, we can add parameters. A parameter is just like a variable.
- You may have noticed the parentheses after the function name, like: `test()`. The parameters are specified inside the parentheses.

# PHP Functions -Adding parameters

- Our first function (`test()`) is a very simple function. It only writes a static string.
- To add more functionality to a function, we can add parameters. A parameter is just like a variable.
- You may have noticed the parentheses after the function name, like: `test()`. The parameters are specified inside the parentheses.

# PHP Functions -Return values

- Functions can also be used to return values.

# Passing Function parameters by reference

- It is possible to pass arguments to functions by reference. This means that a reference to the variable is manipulated by the function rather than a copy of the variable's value.
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/edit/master/Module3/3.8Functions/3.8.1%20Function.php>

# Function

- **Default parameters in Function**

- You can set a parameter to have a default value if the function's caller doesn't pass it.

- **Recursive Function**

- A recursive function is a function that calls itself. Care must be taken in PHP, however, as your code must not carry out a certain number of recursive functioncalls. i.e. There must be a mechanism (IF statement, etc) that stops the recursion after the desired result has been found.

# Function

- **Returning an Array to get multiple values**
- PHP as is doesn't support returning multiple values in functions, but you can —fake|| it with arrays and a function called list . The greatest effect from this is that we
- can return anything in the array, even other arrays.

**Refer Example Link:**

<https://github.com/TopsCode/PHP/edit/master/Module3/3.8Functions/3.8.1%20Function.php>

# Function

- **Variable handling Functions**
- **var\_dump()** — It is used to dump information about a variable. This function displays structured information such as type and value of the given variable. Arrays and objects are explored recursively with values indented to show structure. This function is also effective with expressions.
- **var\_export( )** :- Outputs or returns a parseable string representation of a variable
- **isset()** — Determine if a variable is set and is not NULL
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.9VarDump/3.9.1%20Vardump.php>

# PHP Date()

- **Function Description**
- **checkdate()** Validates a Gregorian date.consider as Month-Dare-Year.
- **Date\_date\_set()**: Sets the date
- **Date\_parse()**: Returns associative array with detailed info about given date
- **Getdate()**: Returns an array that contains date and time information for a Unix timestamp
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/edit/master/Module3/3.10Date/3.10.1.php>

# PHP Date()

- **time()** Returns the Current Time as a Unix Timestamp
- **date()** Formats a Local Time/Date
- **strtotime()** Parses an English Textual Date or Time Into a Unix Timestamp

## PHP Date -Adding a Timestamp

The second parameter in the date() function specifies a timestamp. This parameter is optional. If you do not supply a timestamp, the current time will be used.

In our next example we will use the mktime() function to create a timestamp for tomorrow. The mktime() function returns the Unix timestamp for a specified date.

- **Syntax**
- `mktime(hour,minute,second,month,day,year,is_dst)`
- To go one day in the future we simply add one to the day argument of mktime():
- **Refer Example Link:** <https://github.com/TopsCode/PHP/edit/master/Module3/3.10Date/3.10.1.php>

# PHP Date()

- **date\_default\_timezone\_set**
- date\_default\_timezone\_set function sets the default timezone used by all date/time functions in a script. This function returns False if the timezone\_identifier isn't valid, otherwise True.
- **Description**
- date\_default\_timezone\_set ( string \$timezone\_identifier )
- It is notable that default timezone can also be set by using INI setting date.timezone.
- **Parameters**
- timezone\_identifier
- The timezone identifier refers to the Coordinated Universal Time (UTC) or Greenwich Mean Time (GMT) or region based time like India/Kolkata.
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.10Date/3.10.1.php>

# PHP Arrays

- **What:**An array is collection of items stored at contiguous memory locations. The idea is to store multiple items of same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

# PHP Arrays

- **Numeric Arrays**
- **What:**An array with a numeric index. Values are stored and accessed in linear fashion
- **Associative Arrays**
- **What:**An array with strings as index. This stores element values in association with key values rather than in a strict linear index order.
- **Multidimensional Arrays**
- **What:**An array containing one or more arrays and values are accessed using multiple indices
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

# PHP Arrays

- `array_combine` --Creates an array by using one array for keys and another for its values
- `Array_merge` -Merge one or more arrays
- `Array_merge_recursive` -Merge two or more arrays recursively
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>
- `array_count_values` --Counts all the values of an array
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

# PHP Arrays

Array\_search -Searches the array for a given value and returns the corresponding key if successful

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

Array\_diff -Computes the difference of arrays

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

# PHP Arrays

Array\_intersect -Returns an array containing all the values of array1 that are present in all the arguments.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

Array\_walk,Array\_Map: function runs each array element in a user-defined function. The array's keys and values are parameters in the function.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

# PHP Arrays

Array\_push/pop- function inserts one or more elements to the end of an array.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

- Array\_chunk - function splits an array into chunks of new arrays.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

Array\_Filter- function filter array value from array

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.11Array>

# Other Array functions (contd...)

`asort` -- Sort an array and maintain index association

`compact` -- Create array containing variables and their values

`count` -- Count elements in an array, or properties in an object

`current` -- Return the current element in an array

`each` -- Return the current key and value pair from an array and advance the array cursor

`end` -- Set the internal pointer of an array to its last element

an array

`in_array` -- Checks if a value exists in an array

`key` -- Fetch a key from an associative array

`ksort` -- Sort an array by key

`list` -- Assign variables as if they were an array

`next` -- Advance the internal array pointer of an array

`pos` -- Alias of `current()`

`prev` -- Rewind the internal array pointer

`range` -- Create an array containing a range of elements

# PHP Array

- **Refer Example Link:**

[https://github.com/TopsCode\(PHP/tree/master/Module3/3.11Array](https://github.com/TopsCode(PHP/tree/master/Module3/3.11Array)

# PHP String

- **What:** A string is series of characters, where a character is the same as a byte. This means that PHP only supports a 256-character set, and hence does not offer native Unicode support.

There are 4 ways to specify string in PHP.

single quoted

double quoted

heredoc syntax

newdoc syntax (since PHP 5.3)

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.12String>

# PHP String

- **The Concatenation Operator**
- There is only one string operator in PHP.
- The concatenation operator (.) is used to put two string values together.
- To concatenate two variables together, use the dot (.) operator:
- The output of the code above will be:
- Hello World 1234
- If we look at the code above you see that we used the concatenation operator two times. This is because we had to insert a third string.

# PHP String

- Between the two string variables we added a string with a single character, an empty space, to separate the two variables.

# PHP String

| Function                                   | Description                                                            |
|--------------------------------------------|------------------------------------------------------------------------|
| addcslashes()                              | Returns a string with backslashes in front of the specified characters |
| implode()                                  | Returns a string from the elements of an array                         |
| join()                                     | Alias of implode()                                                     |
| ltrim()                                    | Strips whitespace from the left side of a string                       |
| rtrim()                                    |                                                                        |
| Trim()                                     |                                                                        |
| md5()                                      | Calculates the MD5 hash of a string                                    |
| md5_file()                                 | Calculates the MD5 hash of a file                                      |
| money_format()                             | Returns a string formatted as a currency string                        |
| nl2br()                                    | Inserts HTML line breaks in front of each newline in a string          |
| <u><a href="#">Refer Example Link:</a></u> |                                                                        |

# PHP String

- <https://github.com/TopsCode/PHP/blob/master/Module3/3.12String>

# PHP string

| Function                                   | Description                                                                    |
|--------------------------------------------|--------------------------------------------------------------------------------|
| similar_text()                             | Calculates the similarity between two strings                                  |
| str_ireplace()                             | Replaces some characters in a string (case-insensitive)                        |
| str_pad()                                  | Pads a string to a new length                                                  |
| str_repeat()                               | Repeats a string a specified number of times                                   |
| str_replace()                              | Calculates the similarity between two strings                                  |
| str_shuffle()                              | Randomly shuffles all characters in a string                                   |
| str_split()                                | Splits a string into an array                                                  |
| strcasecmp()                               | Compares two strings (case-insensitive)                                        |
| stristr()                                  | Finds the first occurrence of a string inside another string (caseinsensitive) |
| <u><a href="#">Refer Example Link:</a></u> |                                                                                |

<https://github.com/TopsCode/PHP/blob/master/Module3/3.12String/>

# PHP Include and Require File

- The include() and require() statement allow you to include the code contained in a PHP file within another PHP file. Including a file produces the same result as copying the script from the file specified and pasted into the location where it is called.
- It is recommended to use the require() function instead of include(), because scripts should not
- continue executing if files are missing or misnamed.

# Include\_once and Require\_once

- The include\_once and require\_once statements will only include the file once even if asked to include it a second time i.e. if the specified file has already been included in a previous statement, the file is not included again.
- **Refer Example Link:**  
<https://github.com/TopsCode/PHP/blob/master/Module3/3.13IncludeRequire/3.13.1%20IncludeRequire.php>

# Header Function

- **What :** HTTP is the protocol (the set of 'rules') for transferring data (e.g. HTML in web pages, pictures, files) between web servers and client browsers, and usually takes place on port 80. This is where the 'http://' in website URLs comes from.
- Headers can be separated into two broad types:
- **Request** headers that your browser sends to the server when you request a file, and
- **Response** headers that the server sends to the browser when it serves the file.
- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.14Header>

# Serving different types of files and generating dynamic content using the Content-Type header

- **Why:**

The Content-Type header tells the browser what type of data the server is about to send. Using this header, you can have your PHP scripts output anything from plaintext files to images or zip files.

You can do several interesting things with this. For example, perhaps you want to send the user a pre-formatted text file rather than HTML

Or perhaps you'd like to prompt the user to download the file, rather than viewing it in the browser. With the help of the Content-Disposition header, it's easy to do, and you can even suggest a file name for the user to use:

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module3/3.14Header>

# OOPs and MVC

## SQL Injection

### **What:**

SQL injection, also known as SQLI, is a common attack vector that uses malicious SQL code for backend database manipulation to access information that was not intended to be displayed. This information may include any number of items, including sensitive company data, user lists or private customer details

### **How:**

## SQL Injection

### **Why:**

When calculating the potential cost of a SQLI, it's important to consider the loss of customer trust should personal information such as phone numbers, addresses and credit card details be stolen.

We can use `mysqli_real_escape_string()` function for sql injection

### **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module4>

# OOPs and MVC

## **Connection With MySql and Database**

### **Connecting to a MySQL Database**

Before you can access and work with data in a database, you must create a connection with mysql and particular database.

In PHP, There are main two function for connect to the mysql and Database For Connect With MySql,

#### **Syntax,**

`new Mysqli(Hostname,username,password,Databasename),`

#### **Closing a Connection**

The connection will be closed as soon as the script ends. To close the connection before,

**Refer Example Link:** <https://github.com/TopsCode/PHP/tree/master/Module4>

# OOPs and MVC

## PHP File Upload

- **Restrictions on Upload**

In this script we add some restrictions to the file upload. The user may only upload .gif or .jpeg files and the file size must be under 20 kb:

- **Saving the Uploaded File**

The examples above create a temporary copy of the uploaded files in the PHP temp folder on the server.

The temporary copied files disappears when the script ends. To store the uploaded file we need to copy it to a different location:

- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module4>

# OOPs and MVC

## PHP Sessions

- **What:** Session handling is a key concept in PHP that enables user information to be persisted across all the pages of a website or app. In this post, you'll learn the basics of session handling in PHP.

A session is a mechanism to persist information across the different web pages to identify users as they navigate a site or app

- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module4>

# OOPs and MVC

- **PHP Cookies**
- **What:** A cookie is a small text file that lets you store a small amount of data (nearly 4KB) on the user's computer. They are typically used to keeping track of information such as username that the site can retrieve to personalize the page when user visit the website next time.
- **Note:** The setcookie() function must appear BEFORE the <html> tag.
- **Syntax:** setcookie(name, value, expire, path, domain);
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module4>

# PHP File Handling

| Mode | Description                                                                                  |
|------|----------------------------------------------------------------------------------------------|
| r    | Read only. Starts at the beginning of the file                                               |
| r+   | Read/Write. Starts at the beginning of the file                                              |
| w    | Write only. Opens and clears the contents of file; or creates a new file if it doesn't exist |
| w+   | Read/Write. Opens and clears the contents of file; or creates a new file if it doesn't exist |
| a    | Append. Opens and writes to the end of the file or creates a new file if it doesn't exist    |
| a+   | Read/Append. Preserves file content by writing to the end of the file                        |
| x    | Write only. Creates a new file. Returns FALSE and an error if file already exists            |
| x+   | Read/Write. Creates a new file. Returns FALSE and an error if file already exists            |

# PHP File Handling

- **What:** When we develop a web application using PHP, quite often we need to work with external files, like reading data from a file or maybe writing user data into file etc. So it's important to know how files are handled while working on any web application.
- **LogGenerate:** Create Log file for error Handeling
- **Refer Example Link:**

<https://github.com/TopsCode/PHP/tree/master/Module5>

# PHP File Handling

- **Fopen:** The fopen() function is also used to create a file. Maybe a little confusing, but in PHP, a file is created using the same function used to open files.
- **Fwrite:** The first parameter of fwrite() contains the name of the file to write to and the
- second parameter is the string to be written.
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5>

# PHP File Handling

- **Fread:** The first parameter of fread() contains the name of the file to read from and the second parameter specifies the maximum number of bytes to read.
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# PHP Sending E-Mails

- **The PHP mail() Function**

The PHP mail() function is used to send emails from inside a script.

- **Syntax**

mail(to,subject,message,headers,parameters)

- **Note:**For the mail functions to be available, PHP requires an installed and working email system. The program to be used is defined by the configuration settings in the php.ini file

- **Refer Example Link:**

- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# Java Script

- **What:** JavaScript is an open source & most popular client side scripting language supported by all browsers. JavaScript is used mainly for enhancing the interaction of a user with the webpage
- **How:**
- Events
- **What:** Javascript has events to provide a dynamic interface to a webpage.
- These events are hooked to elements in the Document Object Model(DOM).

# Java Script

| <b>Event name</b> | <b>Event Source</b>                                              | <b>Event handler</b> |
|-------------------|------------------------------------------------------------------|----------------------|
| abort             | image                                                            | onAbort              |
| click             | checkbox, radio button,<br>submit button, reset button<br>& link | onClick              |
| change            | text field/area, list                                            | onChange             |
| dragDrop          | window                                                           | onDragDrop           |
| error             | image, window                                                    | onError              |
| keyDown           | doc, image, link                                                 | onKeyDown            |
| keyPress          | doc, image, link                                                 | onKeyPress           |
| mouseMove         | nothing                                                          | onMouseMove          |
| mouseOut          | link, image map                                                  | onMouseOut           |
| mouseOver         | link, image map                                                  | onMouseOver          |
| reset             | reset button form                                                | onReset              |
| resize            | window                                                           | onResize             |
| submit            | submit button form                                               | onSubmit             |

# Javascript

- Popup Box
- **What:** JavaScript provides different built-in functions to display popup messages for different purposes e.g. to display a simple message or display a message and take user's confirmation on it or display a popup to take a user's input value.
- There are three types of popup boxes in JavaScript.
- **Alert Box:**-- When an alert box pops up, the user will have to click "OK" to proceed.
- **Confirm Box:**-- When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed.
- **Prompt Box:**-- When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value.

# Javascript

- Form Validation
- **What:** It is important to validate the form submitted by the user because it can have inappropriate values. So, validation is must to authenticate user.
- Form data that typically are checked by a JavaScript could be:
- Required fields
- Valid user name
- Valid password
- Valid e-mail address
- Valid phone number
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# XML (EXtensible Markup Language)

- **What:** The most widely used semi-structured format for data, introduced by the W3C in 1998. XML files contain only tags and text similar to HTML.
- By providing a common method for identifying data, XML supports business-to-business transactions and has become "the" format for electronic data interchange and Web services.
- **how:**
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# Jquery Introduction

- **What:** jQuery is the most popular JavaScript library nowadays. It uses CSS selector style syntax to fetch elements in document object model (DOM) into wrapped element set, and then manipulate elements in the wrapped set with jQuery functions to archive different effect.
- Though the way of using jQuery is very easy and intuitive, we should still understand what
- is happening behind the scene to better master this JavaScript library.
- **How:**
- **Syntax:**
- `$(document).ready(function() { $(selector).action(); });`
- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# Ajax (Asynchronous Javascript And XML)

- **What:** client-sided web development technique that is used to produce interactive Web applications.

AJAX is a way of developing an application that combines the functions below, using JavaScript to tie it all together

## Jquery AJAX

- **What:** JQuery is a great tool which provides a rich set of AJAX methods to develop next generation web application.
- **Syntax:**

```
$.ajax(url[, options])
```

```
$.ajax([options])
```

- **Note :** Ajax requests are triggered by the JavaScript code; your code sends a request to a URL, and when the request completes, a callback function can be triggered to handle the response. Further, since the request is asynchronous, the rest of your code continues to execute while the request is being processed.

# Ajax (Asynchronous Javascript And XML)

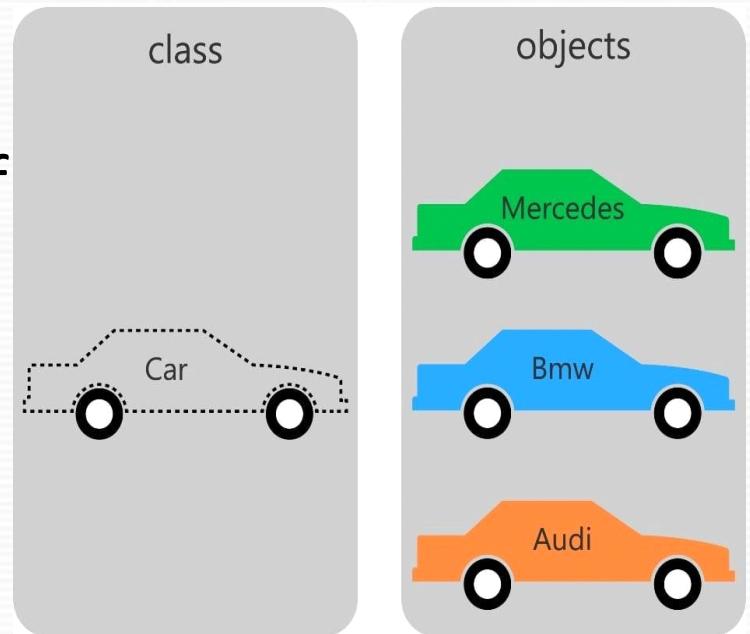
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# Ajax (Asynchronous Javascript And XML)

- Validation Using jQuery(bValidator)
- Download B-Validator from  
<http://bojanmauser.from.hr/bvalidator/download/>

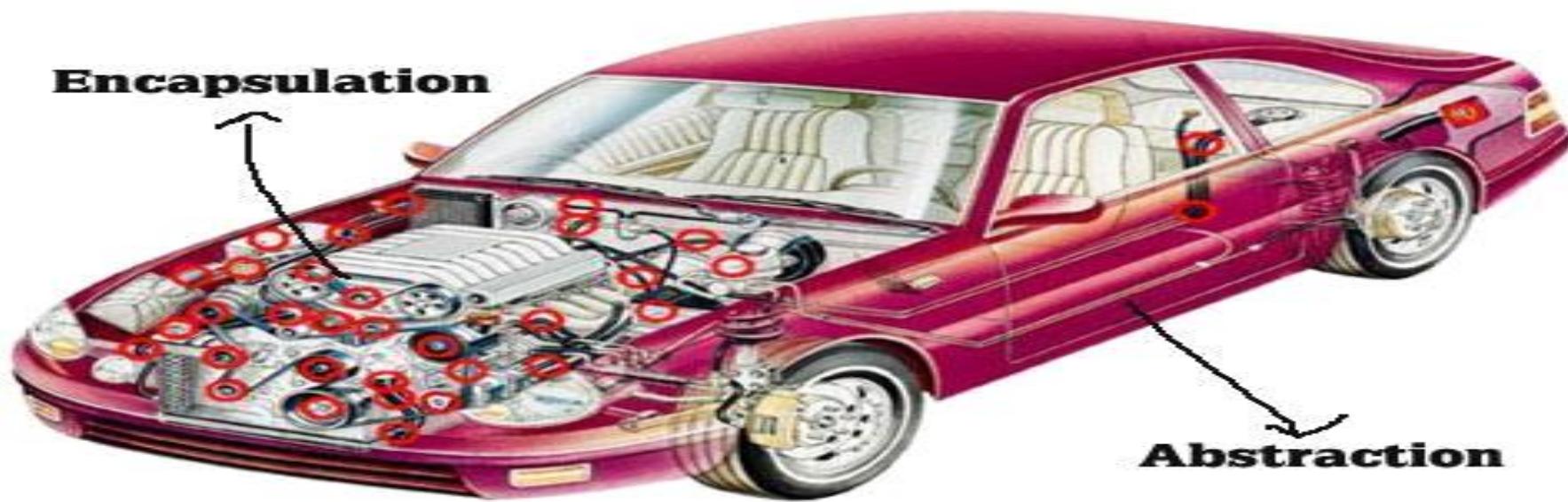
# OOPs(Object Oriented Programming Structure)

- **Class**
- **What :** Object-oriented programming (OOP) is a programming paradigm based on concepts of objects. The object may contain data as a form of instance variables and behaviours in form of methods.
- First, you need to understand two concepts class and objects.
- <https://github.com/TopsCode/PHP/blob/master/Module5/>



# OOPs(Object Oriented Programming Structure)

Encapsulation



# OOPs(Object Oriented Programming Structure)

- Encapsulation is basically **information hiding**. It describes the idea of bundling data and methods that work on that data within one unit. In here access to **data** need to be controlled using access modifiers (public, private, protected etc.) and expose them to the outside world using getters and setters.
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- Inheritance
- **What:**

Inheritance means sub-class inherits from the super-class. In animal class, there are methods and attributes that are common to all animals by using inheritance concept other child classes can use those attributes and methods in the parent class.

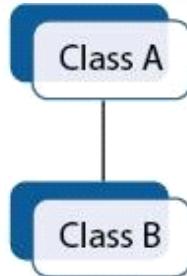
As example lion has a picture, locations and lion eat() etc.

## **Refer Example Link:**

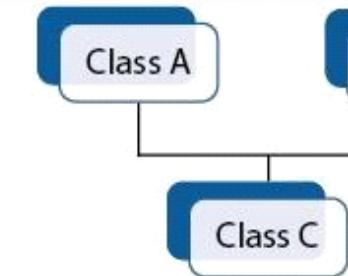
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

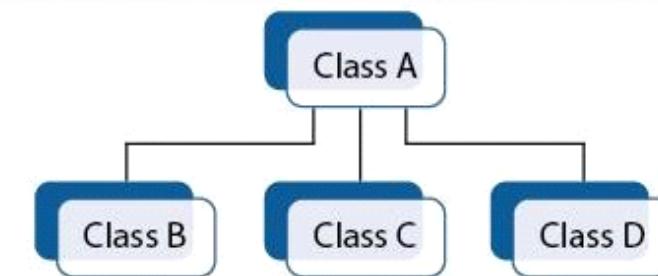
- Inheritance Types



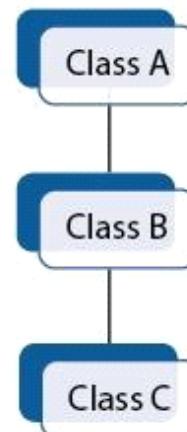
Single Inheritance



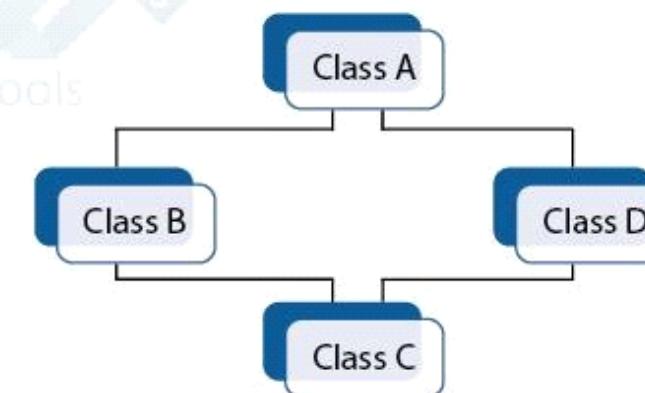
Multiple Inheritance



Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance

# OOPs(Object Oriented Programming Structure)

## Constructor

**What:** If a class name and function name will be similar in that case function is known as constructor. Constructor is special type of method because its name is similar to class name.

- [Refer Example Link:](#)
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

## Destructor

- **What:** The Destructor method will be called as soon as there are no other references to a particular object, or in any order during the shutdown sequence.
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

| CONSTRUCTOR                                                                                       | DESTRUCTOR                                                                                                                |
|---------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| It allocates the memory to an object.<br><code>class_name( arguments if any ){ };</code>          | It deallocates the memory of an object.<br><code>~ class_name( no arguments ){ };</code>                                  |
| Constructor accepts argument<br>Constructor is called automatically, while the object is created. | Destructor does not accept any argument.<br>Destructor is called automatically, as block is exited or program terminates. |
| Constructor allows object to initialize some of its values before it is used.                     | Destructor allows object to execute some code at the time of its destruction.                                             |
| Constructors are called in successive order.                                                      | Destructors are called in reverse order of constructors.                                                                  |
| Destructors are called in reverse order of constructors.                                          | But there is always a single destructor in the class.                                                                     |
| Copy constructor allows a constructor to declare and initialize a object from another object.     | No such concept.                                                                                                          |
| Constructors can be overloaded.                                                                   | Destructors can not be overloaded.                                                                                        |

# OOPs(Object Oriented Programming Structure)

- **Scope Resolution Operator (::)**
- **What :** Scope resolution operator (:) in C++ programming language is used to define a function outside a class or when we want to use a global variable but also has a local variable with the same name.
- **Where:** The most common use for scope resolution is with the pseudo-class "parent". For example, if you want a child object to call its parent's `__construct()` function, you would use `parent::__construct()`.

Scope resolution is complicated, and not used all that often, however it is important you know about it. Furthermore, there's no need to understand it fully at this point - it is explained in full in the Objects chapter.

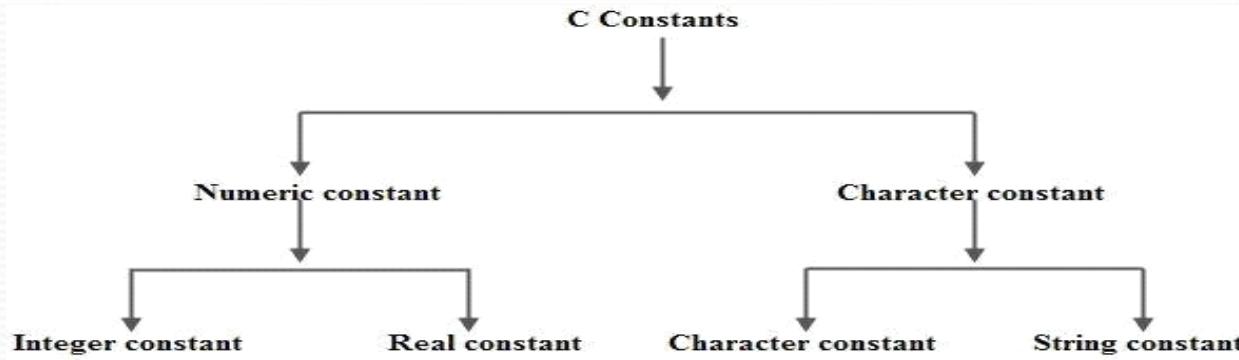
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **Static Keyword**
- **What:** Declaring class members or methods as static makes them accessible without needing an instantiation of the class.
- **How:** In order to define methods and properties as **static**, we use the reserved keyword static.
- **Note:** Static properties cannot be accessed through the object using the arrow operator ->.
- **When:** The main cases in which we consider the use of static methods and properties are when we need them as counters and for utility classes.
- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **Class Constants**
- **What:** Class constants provide a mechanism for holding fixed values in a program. That is, they provide a way of giving a name (and associated compile-time checking) to a value like 3.14 or "Apple". Class constants can only be defined with the const keyword - the define function cannot be used in this context.



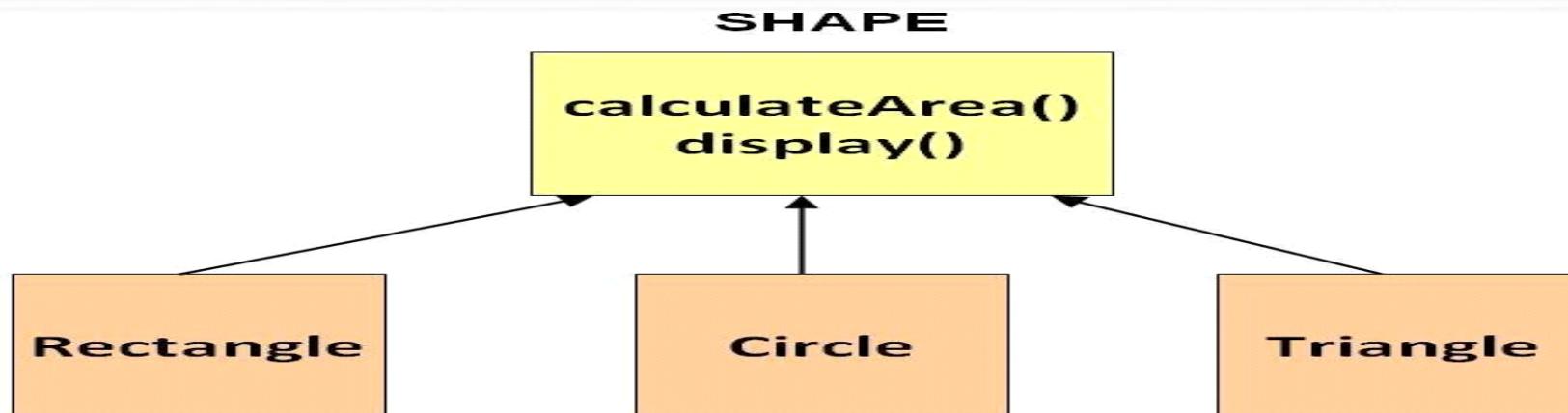
# OOPs(Object Oriented Programming Structure)

## Class Constants

- **Refer Example Link:**
- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **Class Abstraction**
- **What:** PHP introduces abstract classes and methods. It is not allowed to create an instance of a class that has been defined as abstract. Any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature they cannot define the implementation.



# OOPs(Object Oriented Programming Structure)

## Class Abstraction

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **Object Interfaces**
- **What:**Object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are handled

# OOPs(Object Oriented Programming Structure)

- **Implements**

To implement an interface, the *implements* operator is used. All methods in the interface must be implemented within a class; failure to do so will result in a fatal error. Classes may implement more than one interface if desired by separating each interface with a comma.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

## Overloading:

- **What:** Overloading in PHP provides means to dynamically "create" properties and methods. These dynamic entities are processed via magic methods one can establish in a class for various action types. In other terms creating

# OOPs(Object Oriented Programming Structure)

- **Overloading**

Both method calls and member accesses can be overloaded via the \_\_call, \_\_get and \_\_set methods. These methods will only be triggered when your object or inherited object doesn't contain the member or method you're trying to access. All overloading methods must not be defined as static. All overloading methods must be defined as public.

# OOPs(Object Oriented Programming Structure)

- **Member overloading**

`void __set ( string $name, mixed $value ) mixed __get ( string $name )`

Class members can be overloaded to run custom code defined in your class by defining these specially named methods.

The `__set()` method's

`$name` parameter used is the name of the variable that should be set or retrieved.

`$value` parameter specifies the value that the object should set the `$name`.

Note: The `__set()` method cannot take arguments by reference.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **Autoloading Objects**
- **What:** Many developers writing object-oriented applications create one PHP source file per-class definition. One of the biggest annoyances is having to write a long list of needed includes at the beginning of each script (one for each class).

# OOPs(Object Oriented Programming Structure)

- **Object Iteration**

**What:** PHP provides a way for objects to be defined so it is possible to iterate through a list of items, with, for Example a foreach statement. By default, all visible properties will be used for the iteration.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/5.7OOPS/5.7.10AutoloadClass.php>

# OOPs(Object Oriented Programming Structure)

- **Final Keyword**

**What:** PHP introduces the final keyword, which prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final then it cannot be extended.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **Type Hinting**

**What:** PHP introduces Type Hinting. Functions are now able to force parameters to be objects (by specifying the name of the class in the function prototype) or arrays.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- Magic Functions
- **What:** Magic functions (in general) allow you to define class functionality without needing to duplicate code

## PHP Magic Methods

|                         |                            |
|-------------------------|----------------------------|
| <code>__set()</code>    | <code>__toString()</code>  |
| <code>__isset()</code>  | <code>__invoke()</code>    |
| <code>__unset()</code>  | <code>__set_state()</code> |
| <code>__sleep()</code>  | <code>__clone()</code>     |
| <code>__wakeup()</code> | <code>__debugInfo()</code> |

# OOPs(Object Oriented Programming Structure)

## Magic Functions

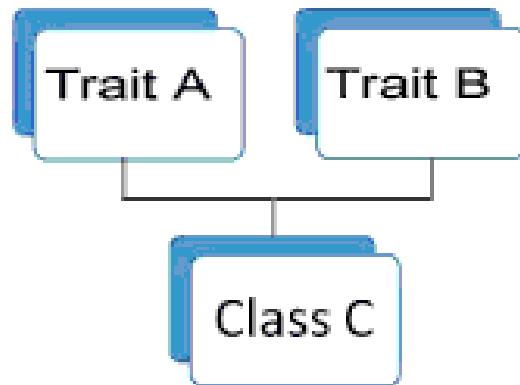
- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

## Trait

- **What:** In php 5.4,Traits to remove the limitation of the multiple inheritance in single inheritance language php



# OOPs(Object Oriented Programming Structure)

- Trait
- is a mechanism for code reuse in single inheritance languages such as PHP. A Trait is intended to reduce some limitations of single inheritance by enabling a developer to reuse sets of methods freely in several independent classes living in different class hierarchies
- **Syntax:**

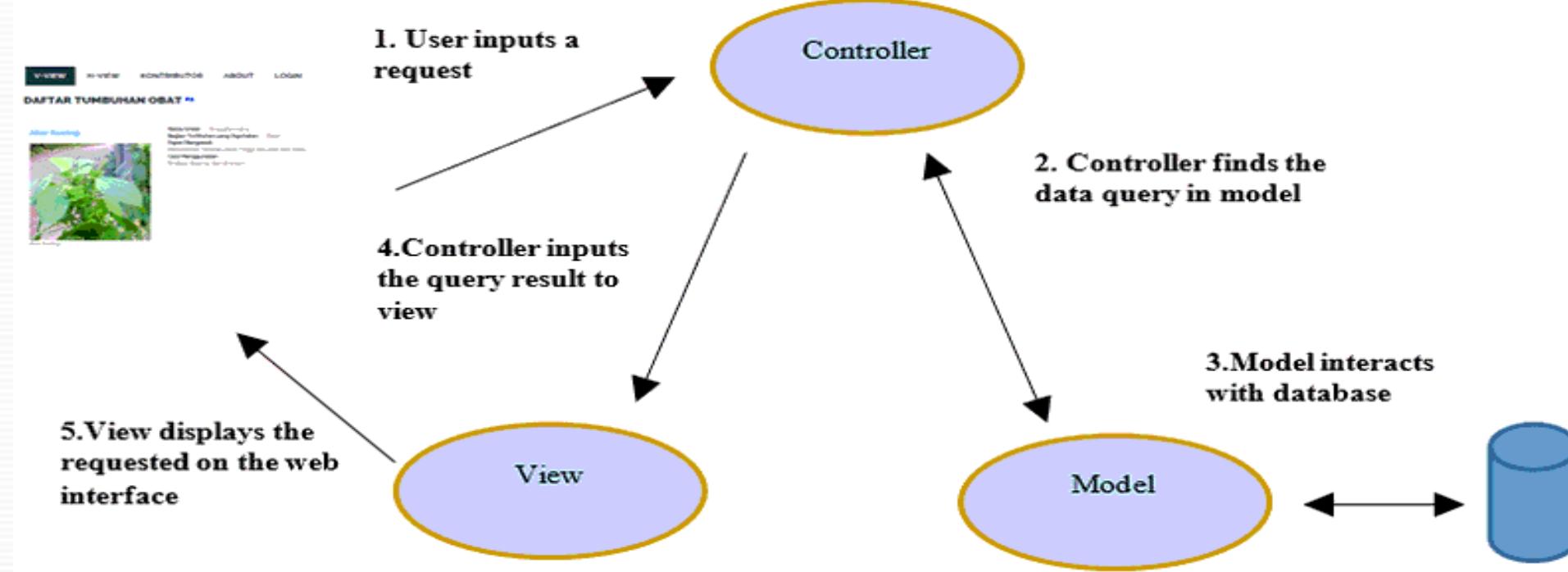
```
<?php
// A sample trait in PHP
trait namethis {
 function ReturnType() {} function ReturnDescription() {}
}?
```

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- Understanding MVC



# OOPs(Object Oriented Programming Structure)

- **Model**

The MVC structure is meant for reasonably-sized applications, using object-oriented coding. The Model part, in a PHP app, would usually be a class (or multiple classes).

Fairly often, the class is a representation of a table you store in the database — member variables are the data columns, and member methods are operations that can be done.

As an example, you might have a User class, having variables such as username, password, email address, and other things. Some of its methods might be a new user creation function, a login function, an authentication function, and a logout function.

- **Refer Example Link:**

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **Controller**
- A controller offers facilities to change the state of the model. The controller interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
- A controller accepts input from the user and instructs the model and view to perform actions based on that input. In effect, the controller is responsible for mapping end-user action to application response.
- The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application they appear as HTTP GET and POST requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

<https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **View**

A view is some form of visualisation of the state of the model.

The view manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application. Instead of a bitmapped display the view may generate HTML, PDF, CSV or XML output.

The view renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented.

The view is responsible for mapping graphics onto a device. A view typically has a one to one correspondence with a display surface and knows how to render to it. A view attaches to a model and renders its contents to the display surface.

- **Refer Example Link:**

- <https://github.com/TopsCode/PHP/blob/master/Module5/>

# OOPs(Object Oriented Programming Structure)

- **Advance Features**
- **What:** **Web services** are client and server applications that communicate over the World Wide Web's (WWW) Hypertext Transfer Protocol (HTTP).  
Web services are application components  
Web services communicate using open protocols  
Web services are self-contained and self-describing  
Web services can be discovered using UDDI  
Web services can be used by other applications  
XML is the basis for Web services

# OOPs(Object Oriented Programming Structure)

- **Advance Features**
- **How:**

The basic Web services platform is XML + HTTP.

XML provides a language which can be used between different platforms and programming languages and still express complex messages and functions.

- **Web services platform elements are:**  
SOAP (Simple Object Access Protocol)
- **Refer Example Link:**



<https://github.com/TopsCode/PHP/blob/master/Module6/>

# Module 3

Introduction of Laravel PHP Framework

# Directory Structure

- [Introduction](#)
- [The Root Directory](#)
  - [The app Directory](#)
  - [The bootstrap Directory](#)
  - [The config Directory](#)
  - [The database Directory](#)
  - [The public Directory](#)
  - [The resources Directory](#)
  - [The routes Directory](#)
  - [The storage Directory](#)
  - [The tests Directory](#)
  - [The vendor Directory](#)
- [The App Directory](#)
  - [The Broadcasting Directory](#)
  - [The Console Directory](#)
  - [The Events Directory](#)
  - [The Exceptions Directory](#)
  - [The Http Directory](#)
  - [The Jobs Directory](#)
  - [The Listeners Directory](#)
  - [The Mail Directory](#)
  - [The Models Directory](#)
  - [The Notifications Directory](#)
  - [The Policies Directory](#)
  - [The Providers Directory](#)
  - [The Rules Directory](#)

# Directory Structure

- [Introduction](#)

The default Laravel application structure is intended to provide a great starting point for both large and small applications. But you are free to organize your application however you like. Laravel imposes almost no restrictions on where any given class is located - as long as Composer can autoload the class.

- [The Root Directory](#)
- [The App Directory](#)

The `app` directory contains the core code of your application. We'll explore this directory in more detail soon; however, almost all of the classes in your application will be in this directory.

- [The Bootstrap Directory](#)

The `bootstrap` directory contains the `app.php` file which bootstraps the framework. This directory also houses a `cache` directory which contains framework generated files for performance optimization such as the route and services cache files. You should not typically need to modify any files within this directory.

# Directory Structure

## . The Config Directory

The `config` directory, as the name implies, contains all of your application's configuration files. It's a great idea to read through all of these files and familiarize yourself with all of the options available to you.

The `database` directory contains your database migrations, model factories, and seeds. If you wish, you may also use this [The Database Directory](#) directory to hold an SQLite database.

# Directory Structure

- **The Public Directory**
- The `public` directory contains the `index.php` file, which is the entry point for all requests entering your application and configures autoloading. This directory also houses your assets such as images, JavaScript, and CSS.
- **The Resources Directory**
- The `resources` directory contains your [views](#) as well as your raw, uncompiled assets such as CSS or JavaScript. This directory also houses all of your language files.

# Directory Structure

- **The Routes Directory**
- The `routes` directory contains all of the route definitions for your application. By default, several route files are included with Laravel: `web.php`, `api.php`, `console.php`, and `channels.php`.
- The `web.php` file contains routes that the `RouteServiceProvider` places in the `web` middleware group, which provides session state, CSRF protection, and cookie encryption. If your application does not offer a stateless, RESTful API then it is likely that all of your routes will most likely be defined in the `web.php` file.
- The `api.php` file contains routes that the `RouteServiceProvider` places in the `api` middleware group. These routes are intended to be stateless, so requests entering the application through these routes are intended to be authenticated [via tokens](#) and will not have access to session state.

# Directory Structure

- The `console.php` file is where you may define all of your closure based console commands. Each closure is bound to a command instance allowing a simple approach to interacting with each command's IO methods. Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application.
- The `channels.php` file is where you may register all of the [event broadcasting](#) channels that your application supports.

- **The Storage Directory**
- The `storage` directory contains your logs, compiled Blade templates, file based sessions, file caches, and other files generated by the framework. This directory is segregated into `app`, `framework`, and `logs` directories. The `app` directory may be used to store any files generated by your application. The `framework` directory is used to store framework generated files and caches. Finally, the `logs` directory contains your application's log files.
- The `storage/app/public` directory may be used to store user-generated files, such as profile avatars, that should be publicly accessible. You should create a symbolic link at `public/storage` which points to this directory. You may create the link using the `php artisan storage:link` Artisan command.

# The App Directory

- . The majority of your application is housed in the `app` directory.

By default, this directory is namespaced under `App` and is

autoloaded by Composer using the [PSR-4 autoloading](#)

[standard](#).

# Directory Structure

- **The Tests Directory**
- The `tests` directory contains your automated tests. Example [PHPUnit](#) unit tests and feature tests are provided out of the box. Each test class should be suffixed with the word `Test`. You may run your tests using the `phpunit` or `php vendor/bin/phpunit` commands. Or, if you would like a more detailed and beautiful representation of your test results, you may run your tests using the `php artisan test` Artisan command.
- **The Vendor Directory**
- The `vendor` directory contains your [Composer](#) dependencies.

# Directory Structure

- The `app` directory contains a variety of additional directories such as `Console`, `Http`, and `Providers`. Think of the `Console` and `Http` directories as providing an API into the core of your application. The HTTP protocol and CLI are both mechanisms to interact with your application, but do not actually contain application logic. In other words, they are two ways of issuing commands to your application. The `Console` directory contains all of your Artisan commands, while the `Http` directory contains your controllers, middleware, and requests.
- **The Broadcasting Directory**
- The `Broadcasting` directory contains all of the broadcast channel classes for your application. These classes are generated using the `make:channel` command. This directory does not exist by default, but will be created for you when you create your first channel. To learn more about channels, check out the documentation on [event broadcasting](#).

# Directory Structure

- **The Console Directory**
- The `Console` directory contains all of the custom Artisan commands for your application. These commands may be generated using the `make:command` command. This directory also houses your console kernel, which is where your custom Artisan commands are registered and your [scheduled tasks](#) are defined.
- **The Events Directory**
- This directory does not exist by default, but will be created for you by the `event:generate` and `make:event` Artisan commands. The `Events` directory houses [event classes](#). Events may be used to alert other parts of your application that a given action has occurred, providing a great deal of flexibility and decoupling.

# The Exceptions Directory

- . The `Exceptions` directory contains your application's exception handler and is also a good place to place any exceptions thrown by your application. If you would like to customize how your exceptions are logged or rendered, you should modify the `Handler` class in this directory.

# Directory Structure

- **The Http Directory**
- The `Http` directory contains your controllers, middleware, and form requests. Almost all of the logic to handle requests entering your application will be placed in this directory.
- **The Jobs Directory**
- This directory does not exist by default, but will be created for you if you execute the `make:job` Artisan command. The `Jobs` directory houses the [queueable jobs](#) for your application. Jobs may be queued by your application or run synchronously within the current request lifecycle. Jobs that run synchronously during the current request are sometimes referred to as "commands" since they are an implementation of the [command pattern](#).

- **The Listeners Directory**
- This directory does not exist by default, but will be created for you if you execute the `event:generate` or `make:listener` Artisan commands. The `Listeners` directory contains the classes that handle your [events](#).
- **The Mail Directory**
- This directory does not exist by default, but will be created for you if you execute the `make:mail` Artisan command. The `Mail` directory contains all of your [classes that represent emails](#) sent by your application. Mail objects allow you to encapsulate all of the logic of building an email in a single, simple class that may be sent using the `Mail::send` method.

# Directory Structure

- **The Models Directory**
- The `Models` directory contains all of your [Eloquent model classes](#). The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.
- **The Notifications Directory**
- This directory does not exist by default, but will be created for you if you execute the `make:notification` Artisan command. The `Notifications` directory contains all of the "transactional" [notifications](#) that are sent by your application, such as simple notifications about events that happen within your application.

# Directory Structure

- **The Policies Directory**
- This directory does not exist by default, but will be created for you if you execute the `make:policy` Artisan command. The `Policies` directory contains the [authorization policy classes](#) for your application. Policies are used to determine if a user can perform a given action against a resource.

- **The Providers Directory**
- The `Providers` directory contains all of the [service providers](#) for your application. Service providers bootstrap your application by binding services in the service container, registering events, or performing any other tasks to prepare your application for incoming requests.
- In a fresh Laravel application, this directory will already contain several providers. You are free to add your own providers to this directory as needed.
- **The Rules Directory**
- This directory does not exist by default, but will be created for you if you execute the `make:rule` Artisan command

# Artisan command to generate boilerplate code for a controller

Laravel has awesome set of artisan commands, probably the most often used are **make:xxx** – like **make:model** or **make:migration** etc. But do you know all 21 of them? And, moreover, do you know their parameters which may help to make the code even quicker?

## . 1. **make:controller**

- This command creates a new controller file in **app/Http/Controllers** folder.
- **Example usage:**
- `php artisan make:controller UserController`

# Artisan command to generate boilerplate code for a controller

## 2. make:model

Create a new Eloquent model class.

**Example usage:**

```
php artisan make:model Photo
```

Or even shorter:

```
php artisan make:model Project -mcr
```

# Artisan command to generate boilerplate code for a controller

## 3. make:migration

Create a new migration file.

**Example usage:**

```
php artisan make:migration create_projects_table
```

**Parameters:**

--create=Table

The table to be created.

--table=Table

Artisan command to generate boilerplate code for a controller

## 4. make:seeder

Create a new database seeder class.

**Example usage:**

```
php artisan make:seeder BooksTableSeeder
```

Artisan command to generate boilerplate code for a controller

## 5. make:request

Create a new form request class in **app/Http/Requests** folder.

**Example usage:**

```
php artisan make:request StoreBlogPost
```

Artisan command to generate boilerplate code for a controller

## 6. make:middleware

Create a new middleware class.

**Example usage:**

```
php artisan make:middleware CheckAge
```

Web Application Development in PHP with Laravel 2021



## Artisan command to generate boilerplate code for a controller

### 7. make:policy

Create a new policy class.

#### Example usage:

```
php artisan make:policy PostPolicy
```

#### Parameters:

--model=Photo

The model that the policy applies to.

Artisan command to generate boilerplate code for a controller

## 8. make:command

Create a new Artisan command.

**Example usage:**

```
php artisan make:command SendEmails
```

**Parameters:**

`--command=Command`

The terminal command that should be assigned.

Artisan command to generate boilerplate code for a controller

## 9. make:event

Create a new event class.

**Example usage:**

```
php artisan make:event OrderShipped
```

**Parameters:** none.

## Artisan command to generate boilerplate code for a controller

### 10. make:job

Create a new job class.

#### Example usage:

```
php artisan make:job SendReminderEmail
```

#### Parameters:

--sync

Indicates that job should be synchronous.

# Artisan command to generate boilerplate code for a controller

## 11. make:listener

Create a new event listener class.

### Example usage:

```
php artisan make:listener SendShipmentNotification
```

### Parameters:

--event=Event

The event class being listened for.

--queued

Indicates the event listener should be queued.

# Artisan command to generate boilerplate code for a controller

## 12. make:mail

Create a new email class.

**Example usage:**

```
php artisan make:mail OrderShipped
```

**Parameters:**

**--markdown**

Create a new Markdown template for the mailable.

**--force**

Create the class even if the mailable already exists.

# Artisan command to generate boilerplate code for a controller

## 13. make:notification

Create a new notification class.

**Example usage:**

```
php artisan make:notification InvoicePaid
```

**Parameters:**

**--markdown**

Create a new Markdown template for the notification.

**--force**

Create the class even if the notification already exists.

Artisan command to generate boilerplate code for a controller

- **14. make:provider**

- Create a new service provider class.

- **Example usage:**

- `php artisan make:provider DuskServiceProvider`

- **Parameters:** none.

Artisan command to generate boilerplate code for a controller

## 15. make:test

Create a new test class.

**Example usage:**

```
php artisan make:test UserTest
```

**Parameters:**

--unit

Create a unit (or, otherwise, feature) test.

Artisan command to generate boilerplate code for a controller

## 16. make:channel

Create a new channel class for broadcasting.

**Example usage:**

```
php artisan make:channel OrderChannel
```

**Parameters:** none.

# Artisan command to generate boilerplate code for a controller

## 17. make:exception

Create a new custom exception class.

### Example usage:

```
php artisan make:exception UserNotFoundException
```

### Parameters:

--render

Create the exception with an empty render method.

--report

Create the exception with an empty report method.

Artisan command to generate boilerplate code for a controller

## 18. make:factory

Create a new model factory.

**Example usage:**

```
php artisan make:factory PostFactory --model=Post
```

**Parameters:**

--model=Post

The name of the model.

## Artisan command to generate boilerplate code for a controller

### 19. make:observer

Create a new observer class.

**Example usage:**

```
php artisan make:observer PostObserver --model=Post
```

**Parameters:**

--model=Post

The model that the observer applies to.

Artisan command to generate boilerplate code for a controller

## 20. make:rule

Create a new validation rule.

**Example usage:**

```
php artisan make:rule Uppercase
```

**Parameters:** none.

Artisan command to generate boilerplate code for a controller

## 21. make:resource

Create a new API resource.

**Example usage:**

```
php artisan make:resource PostResource
```

**Parameters:**

--collection=Post

Create a ResourceCollection instead of individual Resource class.

# Routing

- **Basic Routing**
- The most basic Laravel routes accept a URI and a closure, providing a very simple and expressive method of defining routes and behavior without complicated routing configuration files:
- `use Illuminate\Support\Facades\Route;`
- `Route::get('/greeting', function () {  
 return 'Hello World';  
});`
- **The Default Route Files**

## The Default Route Files

- All Laravel routes are defined in your route files, which are located in the `routes` directory. These files are automatically loaded by your application's `App\Providers\RouteServiceProvider`. The `routes/web.php` file defines routes that are for your web interface. These routes are assigned the `web` middleware group, which provides features like session state and CSRF protection. The routes in `routes/api.php` are stateless and are assigned the `api` middleware group.
- For most applications, you will begin by defining routes in your `routes/web.php` file. The routes defined in `routes/web.php` may be accessed by entering the defined route's URL in your browser. For example, you may access the following route by navigating to `http://example.com/user` in your browser:
  - `use App\Http\Controllers\UserController;`
  - `Route::get('/user', [UserController::class, 'index']);`

# Available Router Methods

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

- . Sometimes you may need to register a route that responds to multiple HTTP verbs. You may do so using the `match` method. Or, you may even register a route that responds to all HTTP verbs using the `any` method:

```
Route::match(['get', 'post'], '/', function () { });
Route::any('/', function () { // })
```

# Route Parameters

- Required Parameters

```
Route::get('/user/{id}', function ($id) {
 return 'User ' . $id;
});
```

- Optional Parameters

```
Route::get('/user/{name?}', function ($name = null) { return $name;
});

Route::get('/user/{name?}', function ($name = 'John') { return $name;
});
```

# Named Routes

```
Route::middleware(['first', 'second'])->group(function () {
 Route::get('/', function () {
 // Uses first & second middleware...
 });
}

Route::get('/user/profile', function () {
 // Uses first & second middleware...
});
});
```

# Named Routes

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route by chaining the `name` method onto the route definition:

```
Route::get('/user/profile', function () {
})->name('profile');
```

# Route Groups

Route groups allow you to share route attributes, such as middleware, across a large number of routes without needing to define those attributes on each individual route.

## Middleware

To assign [middleware](#) to all routes within a group, you may use the `middleware` method before defining the group. Middleware are executed in the order they are listed in the array:

# Controllers

- Instead of defining all of your request handling logic as closures in your route files, you may wish to organize this behavior using "controller" classes. Controllers can group related request handling logic into a single class. For example, a `UserController` class might handle all incoming requests related to users, including showing, creating, updating, and deleting users. By default, controllers are stored in the `app/Http/Controllers` directory.

## Basic Controllers

- Let's take a look at an example of a basic controller. Note that the controller extends the base controller class included with Laravel: `App\Http\Controllers\Controller`:

## Single Action Controllers

- If a controller action is particularly complex, you might find it convenient to dedicate an entire controller class to that single action. To accomplish this, you may define a single `_invoke` method within the controller:

# Controller Middleware

[Middleware](#) may be assigned to the controller's routes in your route files:

```
Route::get('profile', [UserController::class, 'show'])->middleware('auth');
```

## . Resource Controllers

If you think of each Eloquent model in your application as a "resource", it is typical to perform the same sets of actions against each resource in your application. For example, imagine your application contains a `Photo` model and a `Movie` model. It is likely that users can create, read, update, or delete these resources.

```
php artisan make:controller PhotoController --resource
use App\Http\Controllers\PhotoController;
```

```
Route::resource('photos', PhotoController::class)
```

# Actions Handled By Resource Controller

| <b>Verb</b> | <b>URI</b>           | <b>Action</b> | <b>Route Name</b> |
|-------------|----------------------|---------------|-------------------|
| GET         | /photos              | index         | photos.index      |
| GET         | /photos/create       | create        | photos.create     |
| POST        | /photos              | store         | photos.store      |
| GET         | /photos/{photo}      | show          | photos.show       |
| GET         | /photos/{photo}/edit | edit          | photos.edit       |
| PUT/PATCH   | /photos/{photo}      | update        | photos.update     |
| DELETE      | /photos/{photo}      | destroy       | photos.destroy    |

# Middleware

Middleware provide a convenient mechanism for inspecting and filtering HTTP requests entering your application. For example, Laravel includes a middleware that verifies the user of your application is authenticated. If the user is not authenticated, the middleware will redirect the user to your application's login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application

## Defining Middleware

```
php artisan make:middleware EnsureTokenIsValid
```

This command will place a new `EnsureTokenIsValid` class within your `app/Http/Middleware` directory. In this middleware, we will only allow access to the route if the supplied `token` input matches a specified value. Otherwise, we will redirect the users back to the `home` URI

# Middleware

## Registering Middleware

### Global Middleware

If you want a middleware to run during every HTTP request to your application, list the middleware class in the `$middleware` property of your `app/Http/Kernel.php` class.

### Assigning Middleware To Routes

If you would like to assign middleware to specific routes, you should first assign the middleware a key in your application's `app/Http/Kernel.php` file. By default, the `$routeMiddleware` property of this class contains entries for the middleware included with Laravel. You may add your own middleware to this list and assign it a key of your choosing:

```
Route::get('/profile', function () {
})->middleware('auth');
```

# Middleware

## Middleware Groups

Sometimes you may want to group several middleware under a single key to make them easier to assign to routes. You may accomplish this using the `$middlewareGroups` property of your HTTP kernel.

```
protected $middlewareGroups = [
 'web' => [
 \App\Http\Middleware\VerifyCsrfToken::class,
],
 'api' => [
 'throttle:api',
 \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

# HTTP Requests

## Retrieving An Input Value

Using a few simple methods, you may access all of the user input from your `Illuminate\Http\Request` instance without worrying about which HTTP verb was used for the request. Regardless of the HTTP verb, the `input` method may be used to retrieve user input:

```
$name = $request->input('name');
```

# HTTP Requests

## Determining If Input Is Present

You may use the `has` method to determine if a value is present on the request. The `has` method returns `true` if the value is present on the request:

```
if ($request->has('name')) {
}
```

# HTTP Requests

Laravel's `Illuminate\Http\Request` class provides an object-oriented way to interact with the current HTTP request being handled by your application as well as retrieve the input, cookies, and files that were submitted with the request.

## Accessing The Request

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the `Illuminate\Http\Request` class on your route closure or controller method. The incoming request instance will automatically be injected by the Laravel [service container](#):

## Input

### Retrieving Input

#### Retrieving All Input Data

```
$input = $request->all();
```

# HTTP Requests

## Cookies

### Retrieving Cookies From Requests

All cookies created by the Laravel framework are encrypted and signed with an authentication code, meaning they will be considered invalid if they have been changed by the client. To retrieve a cookie value from the request, use the `cookie` method on an `Illuminate\Http\Request` instance:

```
$value = $request->cookie('name');
```

### File Paths & Extensions:

```
$path = $request->photo->path();
```

The `UploadedFile` class also contains methods for accessing the file's fully-qualified path and its extension.

```
$extension = $request->photo->extension();
```

# HTTP Requests

## Storing Uploaded Files

To store an uploaded file, you will typically use one of your configured [filesystems](#). The [UploadedFile](#) class has a `store` method that will move an uploaded file to one of your disks, which may be a location on your local filesystem or a cloud storage location like Amazon S3.

```
$path = $request->photo->store('images');
$path = $request->photo->store('images', 's3');
```

If you do not want a filename to be automatically generated, you may use the `storeAs` method, which accepts the path, filename, and disk name as its arguments:

```
$path = $request->photo->storeAs('images', 'filename.jpg');
$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

# Blade Templates

Blade is the simple, yet powerful templating engine that is included with Laravel. Unlike some PHP templating engines, Blade does not restrict you from using plain PHP code in your templates. In fact, all Blade templates are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application.

Blade template files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

Blade views may be returned from routes or controller using the global `view` helper. Of course, as mentioned in the documentation on [views](#), data may be passed to the Blade view using the `view` helper's second argument:

```
Route::get('/', function () {
 return view('greeting', ['name' => 'Finn']);
});
```

# Blade Templates

## Blade Directives

In addition to template inheritance and displaying data, Blade also provides convenient shortcuts for common PHP control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with PHP control structures while also remaining familiar to their PHP counterparts.

## If Statements

You may construct `if` statements using the `@if`, `@elseif`, `@else`, and `@endif` directives. These directives function identically to their PHP counterparts:

# Blade Templates

```
@if (count($records) === 1)
 I have one record!
@elseif (count($records) > 1)
 I have multiple records!
@else
 I don't have any records!
@endif
```

In addition to the conditional directives already discussed, the `@isset` and `@empty` directives may be used as convenient shortcuts for their respective PHP functions:

```
@isset($records)
@endisset
```

# Blade Templates

## Switch Statements

Switch statements can be constructed using the `@switch`, `@case`, `@break`, `@default` and `@endswitch` directives:

```
@switch($i)
 @case(1)
 First case...
 @break
 @default
 Default case...@endswitch
```

# Blade Templates

## Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's loop structures. Again, each of these directives functions identically to their PHP counterparts:

```
@for ($i = 0; $i < 10; $i++)
 The current value is {{ $i }}
@endfor
@foreach ($users as $user)
 <p>This is user {{ $user->id }}</p>
@endforeach
```

# Blade Templates

```
@foreach ($users as $user)
 @if ($user->type == 1)
 @continue
 @endif

 {{ $user->name }}

 @if ($user->number == 5)
 @break
 @endif
@endforeach
```

# Module 4

Database Operation-migration

# Migration

## Introduction

- Migrations are a type of version control for your database.
- They allow a team to modify the database schema and stay up to date on the current schema state.
- Migrations are typically paired with the Schema Builder to easily manage your application's schema.

## Creating Migrations

```
php artisan make:migration create_users_table
```

- The --table and --create options may also be used to indicate the name of the table, and whether the migration will be creating a new table:

```
php artisan make:migration add_votes_to_users_table --table=users
```

```
php artisan make:migration create_users_table --create=users
```

# Migrations

## Running Migrations

### Running All Outstanding Migrations

```
php artisan migrate
```

**Note: If you receive a "class not found" error when running migrations, try running the composer dump-autoload command.**

## b. Forcing Migrations In Production

```
php artisan migrate --force
```

### • Rolling Back Migrations

1. Rollback The Last Migration Operation [php artisan migrate:rollback](#)
2. Rollback all migrations [php artisan migrate:reset](#)
3. Rollback all migrations and run them all again [php artisan migrate:refresh](#)

# Migrations

- Laravel also includes a simple way to seed your database with test data using seed classes.
- All seed classes are stored in database/seeds.
- Seed classes may have any name you wish, but probably should follow some sensible convention, such as UserTableSeeder, etc.
- By default, a DatabaseSeeder class is defined for you. From this class, you may use the call method to run other seed classes, allowing you to control the seeding order.

## • Writing Seeders

- *php artisan make:seeder UserSeeder*
- A seeder class only contains one method by default: run
- This method is called when the db:seed Artisan command is executed.
- Within the run method, you may insert data into your database however you wish.

# Migrations

```
class DatabaseSeeder extends Seeder
{
 public function run()
 {
 DB::table('users')->insert([
 'name' => Str::random(10),
 'email' => Str::random(10).'@gmail.com',
 'password' =>
 Hash::make('password'),
]);
 }
}
```

# Migrations

## Running Seeders

You may execute the db:seed Artisan command to seed your database.

By default, the db:seed command runs the Database\Seeders\DatabaseSeeder class, which may in turn invoke other seed classes.

However, you may use the --class option to specify a specific seeder class to run individually:

[php artisan db:seed](#)

[php artisan db:seed --class=UserSeeder](#)

You may also seed your database using the migrate:fresh command in combination with the --seed option,

[php artisan migrate:fresh --seed](#)

## Forcing Seeders To Run In Production

[php artisan db:seed --force](#)

# Module5

User inputs with Blade Template

# Creating contact us form:

Create contact us page using this link:

Link:

[https://github.com/TopsCode/Laravel/blob/master/Moduel5%26\\_Input/Myproject/resources/views/contactus.blade.php](https://github.com/TopsCode/Laravel/blob/master/Moduel5%26_Input/Myproject/resources/views/contactus.blade.php)

# Sending email:

Sending email doesn't have to be complicated. Laravel provides a clean, simple email API powered by the popular SwiftMailer library. Laravel and SwiftMailer provide drivers for sending email via SMTP, Mailgun, Postmark, Amazon SES, and sendmail, allowing you to quickly get started sending mail through a local or cloud based service of your choice.

## Configuration:

Laravel's email services may be configured via your application's [config/mail.php](#) configuration file. Each mailer configured within this file may have its own unique configuration and even its own unique "transport", allowing your application to use different email services to send certain email messages. For example, your application might use Postmark to send transactional emails while using Amazon SES to send bulk emails.

Within your mail configuration file, you will find a mailers configuration array. This array contains a sample configuration entry for each of the major mail drivers / transports supported by Laravel, while the default configuration value determines which mailer will be used by default when your application needs to send an email message.

# Sending email:

## **Driver / Transport Prerequisites:**

The API based drivers such as Mailgun and Postmark are often simpler and faster than sending mail via SMTP servers. Whenever possible, we recommend that you use one of these drivers. All of the API based drivers require the Guzzle HTTP library, which may be installed via the Composer package manager:

```
composer require guzzlehttp/guzzle
```

# Sending email:

**Env file credential setting:**

use this while using mailtrap

MAIL\_DRIVER=smtp

MAIL\_HOST=smtp.mailtrap.io

MAIL\_PORT=587

MAIL\_USERNAME=YourUserName

MAIL\_PASSWORD=YourPassword

MAIL\_ENCRYPTION=tls

MAIL\_FROM\_ADDRESS=mygoogle@gmail.com

MAIL\_FROM\_NAME="\${APP\_NAME}"

# Sending email:

**Env file credential setting:**

use this while using gmail

MAIL\_DRIVER=smtp

MAIL\_HOST=smtp.googlemail.com

MAIL\_PORT=465

MAIL\_USERNAME= YourEmail

MAIL\_PASSWORD= YourPassword

MAIL\_ENCRYPTION=ssl

# Sending email:

## Generating Mailables:

When building Laravel applications, each type of email sent by your application is represented as a "mailable" class. These classes are stored in the app/Mail directory. Don't worry if you don't see this directory in your application, since it will be generated for you when you create your first mailable class using the make:mail Artisan command:

# Sending email:

## Configuring The Sender:

### Using The from Method

First, let's explore configuring the sender of the email. Or, in other words, who the email is going to be "from". There are two ways to configure the sender. First, you may use the from method within your mailable class' build method:

```
public function build()
{
 return $this->from('example@example.com', 'Example')
 ->view('emails.orders.shipped');
}
```

# Sending email:

## Using A Global from Address

However, if your application uses the same "from" address for all of its emails, it can become cumbersome to call the from method in each mailable class you generate. Instead, you may specify a global "from" address in your config/mail.php configuration file. This address will be used if no other "from" address is specified within the mailable class:

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

In addition, you may define a global "reply\_to" address within your config/mail.php configuration file:

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

# Sending email:

## Configuring The View:

Within a mailable class' build method, you may use the view method to specify which template should be used when rendering the email's contents. Since each email typically uses a Blade template to render its contents, you have the full power and convenience of the Blade templating engine when building your email's HTML:

//Mail class

Link:[https://github.com/TopsCode/Laravel/blob/master/Moduel5\\_Forms %26\\_Input/Myproject/app/Mail/UserMessage.php](https://github.com/TopsCode/Laravel/blob/master/Moduel5_Forms %26_Input/Myproject/app/Mail/UserMessage.php)

//Controller:

[https://github.com/TopsCode/Laravel/blob/master/Moduel5\\_Forms %26\\_Input/Myproject/app/Http/Controllers/ProductController.php](https://github.com/TopsCode/Laravel/blob/master/Moduel5_Forms %26_Input/Myproject/app/Http/Controllers/ProductController.php)

```
public function build()
{
 return $this->view('emails.Email.message');
}
```

# Validation

# Validation

Laravel provides several different approaches to validate your application's incoming data. It is most common to use the `validate` method available on all incoming HTTP requests. However, we will discuss other approaches to validation as well.

Laravel includes a wide variety of convenient validation rules that you may apply to data, even providing the ability to validate if values are unique in a given database table. We'll cover each of these validation rules in detail so that you are familiar with all of Laravel's validation features.

# Validation

## Defining The Routes:

```
Route::resource('/products',ProductController::class);
```

## Creating The controller

### Link:

[https://github.com/TopsCode/Laravel/blob/master/Moduel5\\_Forms%26\\_Input/Myproject/app/Http/Controllers/ProductController.php](https://github.com/TopsCode/Laravel/blob/master/Moduel5_Forms%26_Input/Myproject/app/Http/Controllers/ProductController.php)

# Validation

**Creating a custom error message:**

Creating Form Request

```
php artisan make:request FormValidateRequest
```

The generated form request class will be placed in the app/Http/Requests directory. If this directory does not exist, it will be created when you run the make:request command. Each form request generated by Laravel has two methods: authorize and rules. As you might have guessed, the authorize method is responsible for determining if the currently authenticated user can perform the action represented by the request, while the rules method returns the validation rules that should apply to the request's data:

# Validation

Link:

[https://github.com/TopsCode/Laravel/blob/master/Moduel5\\_Forms%26\\_Input/Myproject/app/Http/Requests/FormValidateRequest.php](https://github.com/TopsCode/Laravel/blob/master/Moduel5_Forms%26_Input/Myproject/app/Http/Requests/FormValidateRequest.php)

# Creating a file uploader

# file uploader

## Creating a file uploader

Laravel's filesystem configuration file is located at config/filesystems.php. Within this file, you may configure all of your filesystem "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver are included in the configuration file so you can modify the configuration to reflect your storage preferences and credentials.

The local driver interacts with files stored locally on the server running the Laravel application while the s3 driver is used to write to Amazon's S3 cloud storage service.

### **The Local Driver:**

When using the local driver, all file operations are relative to the root directory defined in your filesystems configuration file. By default, this value is set to the storage/app directory. Therefore, the following method would write to storage/app/example.txt

```
use Illuminate\Support\Facades\Storage;
Storage::disk('local')->put('example.txt', 'Contents');
```

# file uploader

## The Public Disk:

The public disk included in your application's filesystems configuration file is intended for files that are going to be publicly accessible. By default, the public disk uses the local driver and stores its files in storage/app/public. To make these files accessible from the web, you should create a symbolic link from public/storage to storage/app/public. Utilizing this folder convention will keep your publicly accessible files in one directory that can be easily shared across deployments when using zero down-time deployment systems like Envoyer. To create the symbolic link, you may use the storage:link Artisan command:

```
php artisan storage:link
```

# file uploader

Once a file has been stored and the symbolic link has been created, you can create a URL to the files using the asset helper:

```
echo asset('storage/file.txt');
```

You may configure additional symbolic links in your filesystems configuration file. Each of the configured links will be created when you run the storage:link command:

```
'links' => [
 public_path('storage') => storage_path('app/public'),
 public_path('images') => storage_path('app/images'),
],
```

# file uploader

## Controller Method:

```
$filename=time().".".{$request->img->getClientOriginalExtension();}
```

```
 {$request->img->storeAs('Myimages',$filename);}
```

[https://github.com/TopsCode/Laravel/blob/master/Moduel5 Forms %26 Input/Myproject/app/Http/Controllers/ProductController.php](https://github.com/TopsCode/Laravel/blob/master/Moduel5%20Forms%26Input/Myproject/app/Http/Controllers/ProductController.php)

# Laravel Collective:

Web Application Development in PHP with Laravel 2021



# Laravel Collective:

## Installing Laravel Collective Forms

The first step is to run this in your console making sure you are in your project directory:

```
composer require laravelcollective/html
```

One of the things we had to do in previous versions of Laravel was to add specific providers and aliases which is very easy.

Locate your config folder in your Laravel project and in it, you will see an app.php file. In this file locate providers which should look something like  
'providers' => [

# Laravel Collective:

You can search for it if you want. At the bottom go ahead and paste this:

Collective\Html\HtmlServiceProvider::class,

After providers, look for aliases and add the following lines at its bottom

“Form” => Collective\Html\FormFacade::class,

‘Html’ => Collective\Html\HtmlFacade::class,

# Laravel Collective:

## Opening A Form

```
{!! Form::open(['url' => 'foo/bar']) !!}
//
{!! Form::close() !!}
```

By default, a POST method will be assumed; however, you are free to specify another method:

```
echo Form::open(['url' => 'foo/bar', 'method' => 'put'])
```

Note: Since HTML forms only support POST and GET, PUT and DELETE methods will be spoofed by automatically adding a `_method` hidden field to your form.

You may also open forms that point to named routes or controller actions:

```
echo Form::open(['route' => 'route.name'])
```

# Laravel Collective:

```
echo Form::open(['action' => 'Controller@method'])
```

You may pass in route parameters as well:

```
echo Form::open(['route' => ['route.name', $user->id]])
```

```
echo Form::open(['action' => ['Controller@method', $user->id]])
```

If your form is going to accept file uploads, add a files option to your array:

```
echo Form::open(['url' => 'foo/bar', 'files' => true])
```

# Laravel Collective:

## **Form Model Binding**

### **Opening A Model Form**

Often, you will want to populate a form based on the contents of a model. To do so, use the Form::model method:

```
echo Form::model($user, ['route' => ['user.update', $user->id]])
```

# Laravel Collective:

## Labels

### Generating A Label Element

```
echo Form::label('email', 'E-Mail Address');
```

### Specifying Extra HTML Attributes

```
echo Form::label('email', 'E-Mail Address', ['class' => 'awesome']);
```

**Note:** After creating a label, any form element you create with a name matching the label name will automatically receive an ID matching the label name as well.

# Laravel Collective:

## **Text, Text Area, Password & Hidden Fields**

### **Generating A Text Input**

```
echo Form::text('username');
```

### **Specifying A Default Value**

```
echo Form::text('email', 'example@gmail.com');
```

**Note:** The *hidden* and *textarea* methods have the same signature as the *text* method.

### **Generating A Password Input**

```
echo Form::password('password', ['class' => 'awesome']);
```

### **Generating Other Inputs**

```
echo Form::email($name, $value = null, $attributes = []);
```

```
echo Form::file($name, $attributes = []);
```

# Laravel Collective:

## **Text, Text Area, Password & Hidden Fields**

### **Generating A Text Input**

```
echo Form::text('username');
```

### **Specifying A Default Value**

```
echo Form::text('email', 'example@gmail.com');
```

**Note:** The *hidden* and *textarea* methods have the same signature as the *text* method.

### **Generating A Password Input**

```
echo Form::password('password', ['class' => 'awesome']);
```

### **Generating Other Inputs**

```
echo Form::email($name, $value = null, $attributes = []);
```

```
echo Form::file($name, $attributes = []);
```

# Laravel Collective:

## Checkboxes and Radio Buttons

### Generating A Checkbox Or Radio Input

```
echo Form::checkbox('name', 'value');
```

```
echo Form::radio('name', 'value');
```

### Generating A Checkbox Or Radio Input That Is Checked

```
echo Form::checkbox('name', 'value', true);
```

```
echo Form::radio('name', 'value', true);
```

# Laravel Collective:

## Number

### Generating A Number Input

```
echo Form::number('name', 'value');
```

## Date

### Generating A Date Input

```
echo Form::date('name', \Carbon\Carbon::now());
```

## File Input

### Generating A File Input

```
echo Form::file('image');
```

# Laravel Collective:

## Drop-Down Lists

### Generating A Drop-Down List

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small']);
```

### Generating A Drop-Down List With Selected Default

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small'], 'S');
```

### Generating a Drop-Down List With an Empty Placeholder

This will create an <option> element with no value as the very first option of your drop-down.

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small'], null, ['placeholder' => 'Pick a size...']);
```

# Laravel Collective:

## **Generating A Grouped List**

```
echo Form::select('animal',[
'Cats' => ['leopard' => 'Leopard'],
'Dogs' => ['spaniel' => 'Spaniel'],
]);
```

## **Generating A Drop-Down List With A Range**

```
echo Form::selectRange('number', 10, 20);
```

```
Generating A List With Month Names
echo Form::selectMonth('month');
```

## **Buttons**

### **Generating A Submit Button**

```
echo Form::submit('Click Me!');
```

# Laravel Collective:

## Custom Macros

### Registering A Form Macro

It's easy to define your own custom Form class helpers called "macros". Here's how it works. First, simply register the macro with a given name and a Closure:

```
Form::macro('myField', function()
{
 return '<input type="awesome">';
});
```

Now you can call your macro using its name:

### Calling A Custom Form Macro

```
echo Form::myField();
```

# Laravel Collective:

## Custom Components

### Registering A Custom Component

Custom Components are similar to Custom Macros, however instead of using a closure to generate the resulting HTML, Components utilize [Laravel Blade Templates](#). Components can be incredibly useful for developers who use [Twitter Bootstrap](#), or any other front-end framework, which requires additional markup to properly render forms.

Let's build a Form Component for a simple Bootstrap text input. You might consider registering your Components inside a Service Provider's boot method.

```
Form::component('bsText', 'components.form.text', ['name', 'value', 'attributes']);
<div class="form-group">
{{ Form::label($name, null, ['class' => 'control-label']) }}
{{ Form::text($name, $value, array_merge(['class' => 'form-control'], $attributes)) }}
</div>
```

# Laravel Collective:

## Providing Default Values

When defining your Custom Component's method signature, you can provide default values simply by giving your array items values, like so:

```
Form::component('bsText', 'components.form.text', ['name', 'value' => null, 'attributes' => []]);
```

## Calling A Custom Form Component

Using our example from above (specifically, the one with default values provided), you can call your Custom Component like so:

```
{{ Form::bsText('first_name') }}
```

This would result in something like the following HTML output:

```
<div class="form-group">
<label for="first_name">First Name</label>
<input type="text" name="first_name" value="" class="form-control">
</div>
```

# Laravel Collective:

## Generating URLs

### **link\_to**

Generate a HTML link to the given URL.

```
echo link_to('foo/bar', $title = null, $attributes = [], $secure = null);
```

### **link\_to\_asset**

Generate a HTML link to the given asset.

```
echo link_to_asset('foo/bar.zip', $title = null, $attributes = [], $secure = null);
```

### **link\_to\_route**

Generate a HTML link to the given named route.

```
echo link_to_route('route.name', $title = null, $parameters = [], $attributes = []);
```

### **link\_to\_action**

Generate a HTML link to the given controller action.

```
echo link_to_action('HomeController@getIndex', $title = null, $parameters = [], $attr
```

# Module 6

Practical Implementation with Project

# Module – 6(Authentication)

## Creating a registration & user login form

Using Artisan command to create inbuilt user register and login system – Authenticating Your Application

## Authentication

- [Introduction](#)
- [Database Considerations](#)
- [Authentication Quickstart](#)
  - [Routing](#)
  - [Views](#)
  - [Authenticating](#)
  - [Retrieving The Authenticated User](#)
  - [Protecting Routes](#)
  - [Login Throttling](#)

- [HTTP Basic Authentication](#)
  - [Stateless HTTP Basic Authentication](#)
- [Logging Out](#)
  - [Invalidating Sessions On Other Devices](#)
- [Social Authentication](#)
- [Adding Custom Guards](#)
  - [Closure Request Guards](#)
- [Adding Custom User Providers](#)
  - [The User Provider Contract](#)
  - [The Authenticatable Contract](#)
- [Events](#)

# Module – 6(Authentication)

Laravel makes implementing authentication very simple. In fact, almost everything is configured for you out of the box. The authentication configuration file is located at config/auth.php, which contains several well documented options for tweaking the behavior of the authentication services.

At its core, Laravel's authentication facilities are made up of "guards" and "providers". Guards define how users are authenticated for each request. For example, Laravel ships with a session guard which maintains state using session storage and cookies.

## Database Considerations

By default, Laravel includes an App\User [Eloquent model](#) in your app directory. This model may be used with the default Eloquent authentication driver.

If your application is not using Eloquent, you may use the database authentication driver which uses the Laravel query builder.

When building the database schema for the App\User model, make sure the password column is at least 60 characters in length. Maintaining the default string column length of 255 characters would be a good choice.

# Module – 6(Authentication)

- **Authentication Quickstart**

Laravel ships with several pre-built authentication controllers, which are located in the App\Http\Controllers\Auth namespace. The RegisterController handles new user registration, the LoginController handles authentication, the ForgotPasswordController handles e-mailing links for resetting passwords, and the ResetPasswordController contains the logic to reset passwords. Each of these controllers uses a trait to include their necessary methods. For many applications, you will not need to modify these controllers at all.

## Routing

Laravel provides a quick way to scaffold all of the routes and views you need for authentication using one simple command:

```
php artisan make:auth
```

This command should be used on fresh applications and will install a layout view, registration and login views, as well as routes for all authentication end-points. A HomeController will also be generated to handle post-login requests to your application's dashboard.

## **Creating Applications Including Authentication**

If you are starting a brand new application and would like to include the authentication scaffolding, you may use the --auth directive when creating your application. This command will create a new application with all of the authentication scaffolding compiled and installed:

```
laravel new blog --auth
```

# Module – 6(Authentication)

## Views

As mentioned in the previous section, the `php artisan make:auth` command will create all of the views you need for authentication and place them in the `resources/views/auth` directory.

The `make:auth` command will also create a `resources/views/layouts` directory containing a base layout for your application. All of these views use the Bootstrap CSS framework, but you are free to customize them however you wish

## Authenticating

Now that you have routes and views setup for the included authentication controllers, you are ready to register and authenticate new users for your application! You may access your application in a browser since the authentication controllers already contain the logic (via their traits) to authenticate existing users and store new users in the database.

# Module 6

## Path Customization

When a user is successfully authenticated, they will be redirected to the /home URI. You can customize the post-authentication redirect path using the HOME constant defined in your RouteServiceProvider:

```
public const HOME = '/home';
```

If you need more robust customization of the response returned when a user is authenticated, Laravel provides an empty authenticated(Request \$request, \$user) method within the AuthenticatesUsers trait. This trait is used by the LoginController class that is installed into your application when using the laravel/ui package. Therefore, you can define your own authenticated method within the LoginController class:

```
protected function authenticated(Request $request, $user
{
 return response([
 //
]);
}
```

# Module 6

- **Username Customization**

By default, Laravel uses the email field for authentication. If you would like to customize this, you may define a username method on your LoginController:

```
public function username()
{
 return 'username';
}
```

- **Guard Customization**

You may also customize the "guard" that is used to authenticate and register users. To get started, define a guard method on your LoginController, RegisterController, and ResetPasswordController. The method should return a guard instance:

```
use Illuminate\Support\Facades\Auth;

protected function guard()
{
 return Auth::guard('guard-name');
}
```

# Module 6

- **Validation / Storage Customization**

To modify the form fields that are required when a new user registers with your application, or to customize how new users are stored into your database, you may modify the `RegisterController` class. This class is responsible for validating and creating new users of your application.

The `validator` method of the `RegisterController` contains the validation rules for new users of the application. You are free to modify this method as you wish.

The `create` method of the `RegisterController` is responsible for creating new `App\User` records in your database using the [Eloquent ORM](#). You are free to modify this method according to the needs of your database.

- **Retrieving The Authenticated User**

You may access the authenticated user via the `Auth` facade:

```
use Illuminate\Support\Facades\Auth;
$user = Auth::user();
$id = Auth::id();
```

# Module 6

- Alternatively, once a user is authenticated, you may access the authenticated user via an Illuminate\Http\Request instance. Remember, type-hinted classes will automatically be injected into your controller methods:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ProfileController extends Controller
{
 public function update(Request $request)
 {
 instance of the authenticated user...
 }
}
?
?
```

# Module 6

- **Determining If The Current User Is Authenticated**

To determine if the user is already logged into your application, you may use the check method on the Auth facade, which will return true if the user is authenticated:

```
use Illuminate\Support\Facades\Auth;
```

```
if (Auth::check()) {
 // The user is logged in...
}
```

- **Protecting Routes**

[Route middleware](#) can be used to only allow authenticated users to access a given route. Laravel ships with an auth middleware, which is defined at `Illuminate\Auth\Middleware\Authenticate`. Since this middleware is already registered in your HTTP kernel, all you need to do is attach the middleware to a route definition:

# Module 6

```
Route::get('profile', function () {
 // Only authenticated users may enter...
})->middleware('auth');
```

If you are using [controllers](#), you may call the middleware method from the controller's constructor instead of attaching it in the route definition directly:

```
public function __construct()
{
 $this->middleware('auth');
}
```

- **Redirecting Unauthenticated Users**

When the auth middleware detects an unauthorized user, it will redirect the user to the login [named route](#). You may modify this behavior by updating the redirectTo function in your app/Http/Middleware/Authenticate.php file:

```
protected function redirectTo($request)
{
 return route('login');
}
```

# Module 6

- **Specifying A Guard**

When attaching the auth middleware to a route, you may also specify which guard should be used to authenticate the user. The guard specified should correspond to one of the keys in the guards array of your auth.php configuration file:

```
public function __construct()
{
 $this->middleware('auth:api');
}
```

- **Password Confirmation**

Sometimes, you may wish to require the user to confirm their password before accessing a specific area of your application. For example, you may require this before the user modifies any billing settings within the application.

To accomplish this, Laravel provides a password.confirm middleware. Attaching the password.confirm middleware to a route will redirect users to a screen where they need to confirm their password before they can continue:

# Module 6

```
Route::get('/settings/security', function () {
})->middleware(['auth', 'password.confirm']);
```

After the user has successfully confirmed their password, the user is redirected to the route they originally tried to access. By default, after confirming their password, the user will not have to confirm their password again for three hours. You are free to customize the length of time before the user must re-confirm their password using the auth.password\_timeout configuration option.

## Login Throttling

If you are using Laravel's built-in LoginController class, the Illuminate\Foundation\Auth\ThrottlesLogins trait will already be included in your controller. By default, the user will not be able to login for one minute if they fail to provide the correct credentials after several attempts. The throttling is unique to the user's username / e-mail address and their IP address.

# Module 6

## Manually Authenticating Users

Note that you are not required to use the authentication controllers included with Laravel. If you choose to remove these controllers, you will need to manage user authentication using the Laravel authentication classes directly. Don't worry, it's a cinch!

We will access Laravel's authentication services via the Auth [facade](#), so we'll need to make sure to import the Auth facade at the top of the class. Next, let's check out the attempt method:

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
public function authenticate(Request $request)
{
 $credentials = $request->only('email', 'password');

 if (Auth::attempt($credentials)) {
 return redirect()->intended('dashboard');
 }
}
}
```

# Module 6

The attempt method accepts an array of key / value pairs as its first argument. The values in the array will be used to find the user in your database table. So, in the example above, the user will be retrieved by the value of the email column. If the user is found, the hashed password stored in the database will be compared with the password value passed to the method via the array. You should not hash the password specified as the password value, since the framework will automatically hash the value before comparing it to the hashed password in the database. If the two hashed passwords match an authenticated session will be started for the user.

The attempt method will return true if authentication was successful. Otherwise, false will be returned.

The intended method on the redirector will redirect the user to the URL they were attempting to access before being intercepted by the authentication middleware. A fallback URI may be given to this method in case the intended destination is not available.

# Module 6

## Specifying Additional Conditions

If you wish, you may also add extra conditions to the authentication query in addition to the user's e-mail and password. For example, we may verify that user is marked as "active":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
 // The user is active, not suspended, and exists
}
```

## Accessing Specific Guard Instances

You may specify which guard instance you would like to utilize using the guard method on the Auth facade. This allows you to manage authentication for separate parts of your application using entirely separate authenticatable models or user tables.

The guard name passed to the guard method should correspond to one of the guards configured in your auth.php configuration file:

```
if (Auth::guard('admin')->attempt($credentials)) {
 //
}
```

# Module 6

## Logging Out

To log users out of your application, you may use the `logout` method on the `Auth` facade. This will clear the authentication information in the user's session:

```
Auth::logout();
```

## Remembering Users

If you would like to provide "remember me" functionality in your application, you may pass a boolean value as the second argument to the `attempt` method, which will keep the user authenticated indefinitely, or until they manually logout. Your users table must include the string `remember_token` column, which will be used to store the "remember me" token.

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
 // The user is being remembered...
}
```

If you are using the built-in `LoginController` that is shipped with Laravel, the proper logic to "remember" users is already implemented by the traits used by the controller.

# Module 6

If you are "remembering" users, you may use the `viaRemember` method to determine if the user was authenticated using the "remember me" cookie:

```
if (Auth::viaRemember()) {
 //
}
```

## Other Authentication Methods

### **Authenticate A User Instance**

If you need to log an existing user instance into your application, you may call the `login` method with the user instance. The given object must be an implementation of the `Illuminate\Contracts\Auth\Authenticatable` [contract](#). The `App\User` model included with Laravel already implements this interface:

```
Auth::login($user);

// Login and "remember" the given user...
Auth::login($user, true);
```

You may specify the guard instance you would like to use:

```
Auth::guard('admin')->login($user);
```

# Module 6

## Authenticate A User By ID

To log a user into the application by their ID, you may use the `loginUsingId` method. This method accepts the primary key of the user you wish to authenticate:

```
Auth::loginUsingId(1);
// Login and "remember" the given user...
Auth::loginUsingId(1, true);
```

## Authenticate A User Once

You may use the `once` method to log a user into the application for a single request. No sessions or cookies will be utilized, which means this method may be helpful when building a stateless API:

```
if (Auth::once($credentials)) {
 //
}
```

# Module 6

## HTTP Basic Authentication

[HTTP Basic Authentication](#) provides a quick way to authenticate users of your application without setting up a dedicated "login" page. To get started, attach the auth.basic [middleware](#) to your route. The auth.basic middleware is included with the Laravel framework, so you do not need to define it:

```
Route::get('profile', function () {
 // Only authenticated users may enter...
})->middleware('auth.basic');
```

Once the middleware has been attached to the route, you will automatically be prompted for credentials when accessing the route in your browser. By default, the auth.basic middleware will use the email column on the user record as the "username".

## A Note On FastCGI

If you are using PHP FastCGI, HTTP Basic authentication may not work correctly out of the box. The following lines should be added to your .htaccess file:

```
RewriteCond %{HTTP:Authorization} ^(.+)\$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

# Module 6

## Stateless HTTP Basic Authentication

You may also use HTTP Basic Authentication without setting a user identifier cookie in the session, which is particularly useful for API authentication. To do so, [define a middleware](#) that calls the onceBasic method. If no response is returned by the onceBasic method, the request may be passed further into the application:

```
<?php
namespace App\Http\Middleware;
use Illuminate\Support\Facades\Auth;
class AuthenticateOnceWithBasicAuth
{
 /**
 * Handle an incoming request.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Closure $next
 * @return mixed
 */
 public function handle($request, $next)
 {
 return Auth::onceBasic() ?: $next($request);
 }
}
```

# Module 6

Next, [register the route middleware](#) and attach it to a route:

```
Route::get('api/user', function () {
 // Only authenticated users may enter...
})->middleware('auth.basic.once');
```

## [Logging Out](#)

To manually log users out of your application, you may use the `logout` method on the `Auth` facade. This will clear the authentication information in the user's session:

```
use Illuminate\Support\Facades\Auth;
Auth::logout();
```

## [Invalidating Sessions On Other Devices](#)

Laravel also provides a mechanism for invalidating and "logging out" a user's sessions that are active on other devices without invalidating the session on their current device. This feature is typically utilized when a user is changing or updating their password and you would like to invalidate sessions on other devices while keeping the current device authenticated.

Before getting started, you should make sure that the `Illuminate\Session\Middleware\AuthenticateSession` middleware is present and un-commented in your `app/Http/Kernel.php` class' `web` middleware group:

# Module 6

```
'web' => [
 // ...
 \Illuminate\Session\Middleware\AuthenticateSession::class,
 // ...
],
```

- Then, you may use the `logoutOtherDevices` method on the `Auth` facade. This method requires the user to provide their current password, which your application should accept through an input form:  
`use Illuminate\Support\Facades\Auth;`  
`Auth::logoutOtherDevices($password);`
- When the `logoutOtherDevices` method is invoked, the user's other sessions will be invalidated entirely, meaning they will be "logged out" of all guards they were previously authenticated by. [Adding Custom Guards](#)
- You may define your own authentication guards using the `extend` method on the `Auth` facade. You should place this call to `extend` within a [service provider](#). Since Laravel already ships with an `AuthServiceProvider`, we can place the code in that provider:

- Ref : <https://github.com/TopsCode/Laravel/tree/master/Module6-7>

# Module 6

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;

class AuthServiceProvider extends ServiceProvider
{
 * Register any application authentication / authorization services.
 *
 * @return void
 */
 public function boot()
 {
 $this->registerPolicies();

 Auth::extend('jwt', function ($app, $name, array $config) {
 // Return an instance of Illuminate\Contracts\Auth\Guard...

 return new JwtGuard(Auth::createUserProvider($config['provider']));
 });
 }
}

?>
```

# Module 6

- As you can see in the example above, the callback passed to the extend method should return an implementation of Illuminate\Contracts\Auth\Guard. This interface contains a few methods you will need to implement to define a custom guard. Once your custom guard has been defined, you may use this guard in the guards configuration of your auth.php configuration file:

```
'guards' => [
 'api' => [
 'driver' => 'jwt',
 'provider' => 'users',
],
],
```

- Closure Request Guards**
- The simplest way to implement a custom, HTTP request based authentication system is by using the Auth::viaRequest method. This method allows you to quickly define your authentication process using a single Closure.

# Module 6

- To get started, call the Auth::viaRequest method within the boot method of your AuthServiceProvider. The viaRequest method accepts an authentication driver name as its first argument. This name can be any string that describes your custom guard. The second argument passed to the method should be a Closure that receives the incoming HTTP request and returns a user instance or, if authentication fails, null:

```
use App\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * Register any application authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
 $this->registerPolicies();

 Auth::viaRequest('custom-token', function ($request) {
 return User::where('token', $request->token)->first();
 });
}
```

# Module 6

Once your custom authentication driver has been defined, you use it as a driver within guards configuration of your auth.php configuration file:

```
'guards' => [
 'api' => [
 'driver' => 'custom-token',
],
],
],
```

## [Adding Custom User Providers](#)

If you are not using a traditional relational database to store your users, you will need to extend Laravel with your own authentication user provider. We will use the provider method on the Auth facade to define a custom user provider:

```
<?php
namespace App\Providers;

use App\Extensions\RiakUserProvider;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;
```

# Module 6

```
class AuthServiceProvider extends ServiceProvider
{
 /**
 * Register any application authentication / authorization services.
 *
 * @return void
 */
 public function boot()
 {
 $this->registerPolicies();

 Auth::provider('riak', function ($app, array $config) {
 // Return an instance of Illuminate\Contracts\Auth\UserProvider...
 return new RiakUserProvider($app->make('riak.connection'));
 });
 }
}
```

# Module 6

- After you have registered the provider using the provider method, you may switch to the new user provider in your auth.php configuration file. First, define a provider that uses your new driver:

```
'providers' => [
 'users' => [
 'driver' => 'riak',
],
],
```

- Finally, you may use this provider in your guards configuration:

```
'guards' => [
 'web' => [
 'driver' => 'session',
 'provider' => 'users',
],
],
```

- The User Provider Contract**

- The Illuminate\Contracts\Auth\UserProvider implementations are only responsible for fetching a Illuminate\Contracts\Auth\Authenticatable implementation out of a persistent storage system, such as MySQL, Riak, etc. These two interfaces allow the Laravel authentication mechanisms to continue functioning regardless of how the user data is stored or what type of class is used to represent it.

# Module 6

- Let's take a look at the Illuminate\Contracts\Auth\UserProvider contract:

```
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider
{
 public function retrieveById($identifier);
 public function retrieveByToken($identifier, $token);
 public function updateRememberToken(Authenticatable $user, $token);
 public function retrieveByCredentials(array $credentials);
 public function validateCredentials(Authenticatable $user, array $credentials);
}
```

- The retrieveById function typically receives a key representing the user, such as an auto-incrementing ID from a MySQL database. The Authenticatable implementation matching the ID should be retrieved and returned by the method.
- The retrieveByToken function retrieves a user by their unique \$identifier and "remember me" \$token, stored in a field remember\_token. As with the previous method, the Authenticatable implementation should be returned.

# Module 6

- The updateRememberToken method updates the \$user field remember\_token with the new \$token. A fresh token is assigned on a successful "remember me" login attempt or when the user is logging out.
  - The retrieveByCredentials method receives the array of credentials passed to the Auth::attempt method when attempting to sign into an application. The method should then "query" the underlying persistent storage for the user matching those credentials. Typically, this method will run a query with a "where" condition on \$credentials['username']. The method should then return an implementation of Authenticatable. This method should not attempt to do any password validation or authentication.
  - The validateCredentials method should compare the given \$user with the \$credentials to authenticate the user. For example, this method should probably use Hash::check to compare the value of \$user->getAuthPassword() to the value of \$credentials['password']. This method should return true or false indicating on whether the password is valid.
- 
- [The Authenticatable Contract](#)
  - Now that we have explored each of the methods on the UserProvider, let's take a look at the Authenticatable contract. Remember, the provider should return implementations of this interface from the retrieveById, retrieveByToken, and retrieveByCredentials methods:

# Module 6

```
<?php

namespace Illuminate\Contracts\Auth;
interface Authenticatable
{
 public function getAuthIdentifierName();
 public function getAuthIdentifier();
 public function getAuthPassword();
 public function getRememberToken();
 public function setRememberToken($value);
 public function getRememberTokenName();
}

• This interface is simple. The getAuthIdentifierName method should return the name of the "primary key" field of the user and the getAuthIdentifier method should return the "primary key" of the user. In a MySQL back-end, again, this would be the auto-incrementing primary key. The getAuthPassword should return the user's hashed password. This interface allows the authentication system to work with any User class, regardless of what ORM or storage abstraction layer you are using. By default, Laravel includes a User class in the app directory which implements this interface, so you may consult this class for an implementation example.
```

# Module 6

## Events

Laravel raises a variety of [events](#) during the authentication process. You may attach listeners to these events in your EventServiceProvider:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
 'Illuminate\Auth\Events\Registered' => [
 'App\Listeners\LogRegisteredUser',
],
 'Illuminate\Auth\Events\Attempting' => [
 'App\Listeners\LogAuthenticationAttempt',
],
 'Illuminate\Auth\Events\Authenticated' => [
 'App\Listeners\LogAuthenticated',
],
 'Illuminate\Auth\Events>Login' => [
 'App\Listeners\LogSuccessfulLogin',
],
];
```

# Module 6

```
'Illuminate\Auth\Events\Failed' => [
 'App\Listeners\LogFailedLogin',
],

'Illuminate\Auth\Events\Validated' => [
 'App\Listeners\LogValidated',
],

'Illuminate\Auth\Events\Verified' => [
 'App\Listeners\LogVerified',
],

'Illuminate\Auth\Events\Logout' => [
 'App\Listeners\LogSuccessfulLogout',
],

'Illuminate\Auth\Events\CurrentDeviceLogout' => [
 'App\Listeners\LogCurrentDeviceLogout',
],
```

# Module 6

```
'Illuminate\Auth\Events\OtherDeviceLogout' => [
 'App\Listeners\LogOtherDeviceLogout',
],
```

```
'Illuminate\Auth\Events\Lockout' => [
 'App\Listeners\LogLockout',
],
```

```
'Illuminate\Auth\Events\PasswordReset' => [
 'App\Listeners\LogPasswordReset',
],
];
```

# Module 7

Controllers and Routes for URLs and APIs

# Module 7

## Using Controllers and Routes for URLs and APIs

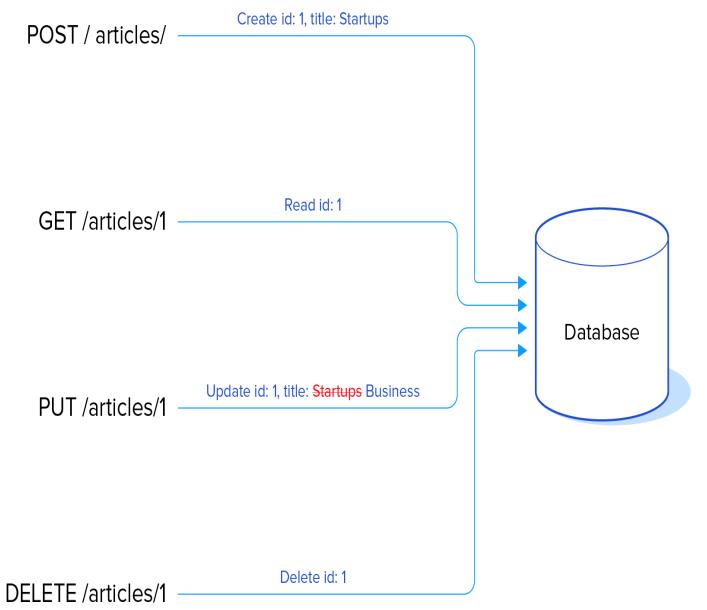
### RESTful APIs

First, we need to understand what exactly is considered a RESTful API. REST stands for *REpresentational State Transfer* and is an architectural style for network communication between applications, which relies on a stateless protocol (usually HTTP) for interaction.

### HTTP Verbs Represent Actions

In RESTful APIs, we use the HTTP verbs as actions, and the endpoints are the resources acted upon. We'll be using the HTTP verbs for their semantic meaning:

- GET: retrieve resources
- POST: create resources
- PUT: update resources
- DELETE: delete resources



# Module 7

## Update Action: PUT vs. POST

RESTful APIs are a matter of much debate and there are plenty of opinions out there on whether it is best to update with POST, PATCH, or PUT, or if the create action is best left to the PUT verb. In this article we'll be using PUT for the update action, as according to the HTTP RFC, PUT means to create/update a resource at a specific location. Another requirement for the PUT verb is idempotence, which in this case basically means you can send that request 1, 2 or 1000 times and the result will be the same: one updated resource in the database.

## Resources

Resources will be the targets of the actions, in our case Articles and Users, and they have their own endpoints:

/articles  
/users

In this laravel api tutorial, the resources will have a 1:1 representation on our data models, but that is not a requirement. You can have resources represented in more than one data model (or not represented at all in the database) and models completely off limits for the user. In the end, you get to decide how to architect resources and models in a way that is fitting to your application.

# Module 7

- **A Note on Consistency**

The greatest advantage of using a set of conventions such as REST is that your API will be much easier to consume and develop around. Some endpoints are pretty straightforward and, as a result, your API will be much more easier to use and maintain as opposed to having endpoints such as GET /get\_article?id\_article=12 and POST /delete\_article?number=40. I've built terrible APIs like that in the past and I still hate myself for it.

However, there will be cases where it will be hard to map to a Create/Retrieve/Update/Delete schema. Remember that the URLs should not contain verbs and that resources are not necessarily rows in a table. Another thing to keep in mind is that you don't have to implement every action for every resource.

## Setting Up a Laravel Web Service Project

As with all modern PHP frameworks, we'll need [Composer](#) to install and handle our dependencies. After you follow the download instructions (and add to your path environment variable), install Laravel using the command:

```
$ composer global require laravel/installer
```

After the installation finishes, you can scaffold a new application like this:

```
$ laravel new myapp
```

# Module 7

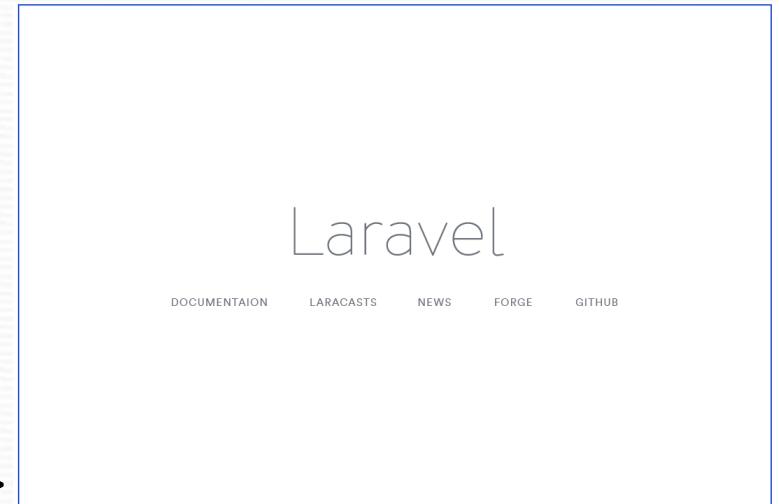
For the above command, you need to have `~/composer/vendor/bin` in your `$PATH`. If you don't want to deal with that, you can also create a new project using Composer:

```
$ composer create-project --prefer-dist laravel/laravel
myapp
```

With Laravel installed, you should be able to start the server and test if everything is working:

```
$ php artisan serve
```

```
Laravel development server started: <http://127.0.0.1:8000>
```



# Module 7

When you open localhost:8000 on your browser, you should see this sample page.

## Migrations and Models

Before actually writing your first migration, make sure you have a database created for this app and add its credentials to the .env file located in the root of the project.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

You can also use Homestead, a Vagrant box specially crafted for Laravel, but that is a bit out of the scope of this article. If you'd like to know more, [refer to the Homestead documentation](#).

Let's get started with our first model and migration—the Article. The article should have a title and a body field, as well as a creation date. Laravel provides several commands through Artisan—Laravel's command line tool—that help us by generating files and putting them in the correct folders. To create the Article model, we can run:

```
$ php artisan make:model Article -m
```

The -m option is short for --migration and it tells Artisan to create one for our model. Here's the generated migration:

# Module 7

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateArticlesTable extends Migration
{
 /**
 * Run the migrations.
 *
 * @return void
 */
 public function up()
 {
 Schema::create('articles', function (Blueprint $table) {
 $table->increments('id');
 $table->timestamps();
 });
 }
}
```

# Module 7

```
/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down() {
 Schema::dropIfExists('articles');
}
}
```

## Let's dissect this for a second:

The up() and down() methods will be run when we migrate and rollback respectively;  
\$table->increments('id') sets up an auto incrementing integer with the name id;

\$table->timestamps() will set up the timestamps for us—created\_at and updated\_at, but don't worry about setting a default, Laravel takes care of updating these fields when needed.

And finally, Schema::dropIfExists() will, of course, drop the table if it exists. With that out of the way, let's add two lines to our up() method:

# Module 7

```
public function up()
{
 Schema::create('articles',
 function (Blueprint $table) {
 $table->increments('id');
 $table->string('title');
 $table->text('body');
 $table->timestamps();
 });
}
```

The string() method creates a VARCHAR equivalent column while text() creates a TEXT equivalent. With that done, let's go ahead and migrate:  
\$ php artisan migrate

You can also use the --step option here, and it will separate each migration into its own batch so that you can roll them back individually if needed. Laravel out of the box comes with two migrations, create\_users\_table and create\_password\_resets\_table. We won't be using the password\_resets table, but having the users table ready for us will be helpful.

Now let's go back to our model and add those attributes to the \$fillable field so that we can use them in our Article::create and Article::update models:

# Module 7

```
class Article extends Model{
 protected $fillable = ['title', 'body'];
}
```

Fields inside the `$fillable` property can be mass assigned using Eloquent's `create()` and `update()` methods. You can also use the `$guarded` property, to allow all but a few properties.

## Database Seeding

Database seeding is the process of filling up our database with dummy data that we can use to test it. Laravel comes with [Faker](#), a great library for generating just the correct format of dummy data for us. So let's create our first seeder:

```
$ php artisan make:seeder ArticlesTableSeeder
```

The seeders will be located in the `/database/seeds` directory. Here's how it looks like after we set it up to create a few articles:

# Module 7

```
class ArticlesTableSeeder extends Seeder
{
 public function run()
 {
 // Let's truncate our existing records to
 // start from scratch.
 Article::truncate();

 $faker = \Faker\Factory::create();
 }
}

// And now, let's create a few articles in our database:
for ($i = 0; $i < 50; $i++) {
 Article::create([
 'title' => $faker->sentence,
 'body' => $faker->paragraph,
]);
}
```

# Module 7

So let's run the seed command:

```
$ php artisan db:seed --class=ArticlesTableSeeder
```

Let's repeat the process to create a Users seeder:

```
class UsersTableSeeder extends Seeder
{
 public function run()
 {
 // Let's clear the users table first
 User::truncate();

 $faker = \Faker\Factory::create();

 // Let's make sure everyone has the same password and
 $password = Hash::make('toptal');

 User::create([
 'name' => 'Administrator',
 'email' => 'admin@test.com',
 'password' => $password,
]);
 }
}
```

# Module 7

```
// And now let's generate a few dozen users for our app:
for ($i = 0; $i < 10; $i++) {
 User::create([
 'name' => $faker->name,
 'email' => $faker->email,
 'password' => $password,
]);
}
}
}
```

- We can make it easier by adding our seeders to the main DatabaseSeeder class inside the database/seeds folder:

```
class DatabaseSeeder extends Seeder
{
 public function run()
 {
 $this->call(ArticlesTableSeeder::class);
 $this->call(UsersTableSeeder::class);
 }
}
```

# Module 7

This way, we can simply run `$ php artisan db:seed` and it will run all the called classes in the `run()` method.

## Routes and Controllers

Let's create the basic endpoints for our application: create, retrieve the list, retrieve a single one, update, and delete. On the `routes/api.php` file, we can simply do this:

```
Use App\Article;

Route::get('articles', function() {
 // If the Content-Type and Accept headers are set to 'application/json',
 // this will return a JSON structure. This will be cleaned up later.
 return Article::all();
});

Route::get('articles/{id}', function($id) {
 return Article::find($id);
});

Route::post('articles', function(Request $request) {
 return Article::create($request->all());
});
```

# Module 7

```
Route::put('articles/{id}', function(Request $request, $id) {
 $article = Article::findOrFail($id);
 $article->update($request->all());

 return $article;
});

Route::delete('articles/{id}', function($id) {
 Article::find($id)->delete();

 return 204;
})
```

The routes inside `api.php` will be prefixed with `/api/` and the API throttling middleware will be automatically applied to these routes (if you want to remove the prefix you can edit the `RouteServiceProvider` class on `/app/Providers/RouteServiceProvider.php`).

Now let's move this code to its own Controller:

```
$ php artisan make:controller ArticleController
```

# Module 7

ArticleController.php:

```
use App\Article;

class ArticleController extends Controller
{
 public function index()
 {
 return Article::all();
 }

 public function show($id)
 {
 return Article::find($id);
 }

 public function store(Request $request)
 {
 return Article::create($request->all());
 }

 public function update(Request $request, $id)
```

# Module 7

```
{
 $article = Article::findOrFail($id);
 $article->update($request->all());

 return $article;
}

public function delete(Request $request, $id)
{
 $article = Article::findOrFail($id);
 $article->delete();

 return 204;
}
}
```

# Module 7

- The routes/api.php file:

```
Route::get('articles', 'ArticleController@index');
Route::get('articles/{id}', 'ArticleController@show');
Route::post('articles', 'ArticleController@store');
Route::put('articles/{id}', 'ArticleController@update');
Route::delete('articles/{id}', 'ArticleController@delete');
```

- We can improve the endpoints by using implicit route model binding. This way, Laravel will inject the Article instance in our methods and automatically return a 404 if it isn't found. We'll have to make changes on the routes file and on the controller:

```
Route::get('articles', 'ArticleController@index');
Route::get('articles/{article}', 'ArticleController@show');
Route::post('articles', 'ArticleController@store');
Route::put('articles/{article}', 'ArticleController@update');
Route::delete('articles/{article}', 'ArticleController@delete');
```

# Module 7

```
class ArticleController extends Controller
{
 public function index()
 {
 return Article::all();
 }

 public function show(Article $article)
 {
 return $article;
 }

 public function store(Request $request)
 {
 $article = Article::create($request->all());

 return response()->json($article, 201);
 }
}
```

# Module 7

```
public function update(Request $request, Article $article)
{
 $article->update($request->all());

 return response()->json($article, 200);
}

public function delete(Article $article)
{
 $article->delete();

 return response()->json(null, 204);
}

}
```

# Module 7

- **A Note on HTTP Status Codes and the Response Format**
- We've also added the `response()->json()` call to our endpoints. This lets us explicitly return JSON data as well as send an HTTP code that can be parsed by the client. The most common codes you'll be returning will be:
  - 200: OK. The standard success code and default option.
  - 201: Object created. Useful for the `store` actions.
  - 204: No content. When an action was executed successfully, but there is no content to return.
  - 206: Partial content. Useful when you have to return a paginated list of resources.
  - 400: Bad request. The standard option for requests that fail to pass validation.
  - 401: Unauthorized. The user needs to be authenticated.
  - 403: Forbidden. The user is authenticated, but does not have the permissions to perform an action.
  - 404: Not found. This will be returned automatically by Laravel when the resource is not found.
  - 500: Internal server error. Ideally you're not going to be explicitly returning this, but if something unexpected breaks, this is what your user is going to receive.

# Module 7

- 503: Service unavailable. Pretty self explanatory, but also another code that is not going to be returned explicitly by the application.
- **Sending a Correct 404 Response**
- If you tried to fetch a non-existent resource, you'll be thrown an exception and you'll receive the whole stacktrace, like this:



TM

- We can fix that by editing our exception handler class, located in app/Exceptions/Handler.php, to return a JSON response:

# Module 7

```
public function render($request, Exception $exception)
{
 // a JSON response.
 if ($exception instanceof ModelNotFoundException) {
 return response()->json([
 'error' => 'Resource not found'
], 404);
 }

 return parent::render($request, $exception);
}
```

Here's an example of the return:

```
{
 data: "Resource not found"
}
```

# Module 7

- If you're using Laravel to serve other pages, you have to edit the code to work with the Accept header, otherwise 404 errors from regular requests will return a JSON as well.
  - ```
public function render($request, Exception $exception)
{
    // This will replace our 404 response with
    // a JSON response.
    if ($exception instanceof ModelNotFoundException &&
        $request->wantsJson())
    {
        return response()->json([
            'data' => 'Resource not found'
        ], 404);
    }
    return parent::render($request, $exception);
}
```
- In this case, the API requests will need the header Accept: application/json.
-

Module 7

Authentication

There are many ways to implement API Authentication in Laravel (one of them being [Passport](#), a great way to implement OAuth2), but in this article, we'll take a very simplified approach.

To get started, we'll need to add an `api_token` field to the `users` table:

```
$ php artisan make:migration --table=users adds_api_token_to_users_table
```

And then implement the migration:

```
public function up()
{
    Schema::table('users', function (Blueprint $table) {
        $table->string('api_token', 60)->unique()->nullable();
    });
}

public function down()
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn(['api_token']);
    });
}
```

Module 7

After that, just run the migration using:

```
$ php artisan migrate
```

Creating the Register Endpoint

We'll make use of the `RegisterController` (in the `Auth` folder) to return the correct response upon registration. Laravel comes with authentication out of the box, but we still need to tweak it a bit to return the response we want.

Done!

Great, your api_token is asda3e23d2dqwd

No, I don't know who you are

Coffee, coming right up John

Hey server, sign me up,
Email:
john.doe@optal.com
Password:
optal

Hey server, log me in,
Email:
john.doe@optal.com
Password:
optal

Hey server,
Make me some coffee

Hey server,
Make me some coffee
Token:
asda3e23d2dqwd



Module 7

- The controller makes use of the trait RegistersUsers to implement the registration. Here's how it works:

```
public function register(Request $request)
{
    // Here the request is validated. The validator method is located
    // inside the RegisterController, and makes sure the name, email
    // password and password_confirmation fields are required.
    $this->validator($request->all())->validate();
    // A Registered event is created and will trigger any relevant
    // observers, such as sending a confirmation email or any
    // code that needs to be run as soon as the user is created.
    event(new Registered($user = $this->create($request->all())));
    // After the user is created, he's logged in.
    $this->guard()->login($user);
    // And finally this is the hook that we want. If there is no
    // registered() method or it returns null, redirect him to
    // some other URL. In our case, we just need to implement
    // that method to return the correct response.
    return $this->registered($request, $user)
        ?: redirect($this->redirectPath());}
```

Module 7

We just need to implement the registered() method in our RegisterController. The method receives the \$request and the \$user, so that's really all we want.

Here's how the method should look like inside the controller:

```
protected function registered(Request $request, $user)
{
    $user->generateToken();

    return response()->json(['data' => $user->toArray()], 201);
}
```

And we can link it on the routes file:

```
Route::post('register', 'Auth\RegisterController@register');
```

In the section above, we used a method on the User model to generate the token. This is useful so that we only have a single way of generating the tokens. Add the following method to your User model:

Module 7

```
class User extends Authenticatable
{
    ...
    public function generateToken()
    {
        $this->api_token = str_random(60);
        $this->save();

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

And that's it. The user is now registered and thanks to Laravel's validation and out of the box authentication, the name, email, password, and password_confirmation fields are required, and the feedback is handled automatically. Checkout the validator() method inside the RegisterController to see how the rules are implemented.

Here's what we get when we hit that endpoint:

```
$ curl -X POST http://localhost:8000/api/register \
-H "Accept: application/json" \
```

```
-H "Content-Type: application/json" \
-d '{"name": "John", "email": "john.doe@toptal.com", "password": "toptal123",
"password_confirmation": "toptal123"}'

{
  "data": {
    "api_token": "OsyHnI0Y9jOlfszq11EC2CBQwCfObmvscrZYo5o2ilZPnohvndH797nDNyAT",
    "created_at": "2017-06-20 21:17:15",
    "email": "john.doe@toptal.com",
    "id": 51,
    "name": "John",
    "updated_at": "2017-06-20 21:17:15"
  }
}
```

Module 7

Creating a Login Endpoint

Just like the registration endpoint, we can edit the LoginController (in the Auth folder) to support our API authentication. The login method of the AuthenticatesUsers trait can be overridden to support our API:

```
public function login(Request $request)
{
    $this->validateLogin($request);

    if ($this->attemptLogin($request)) {
        $user = $this->guard()->user();
        $user->generateToken()
        return response()->json([
            'data' => $user->toArray(),
        ]);
    }

    return $this->sendFailedLoginResponse($request);
}
```

And we can link it on the routes file:

```
Route::post('login', 'Auth\LoginController@login');
```

Module 7

Now, assuming the seeders have been run, here's what we get when we send a POST request to that route:

```
$ curl -X POST localhost:8000/api/login \
-H "Accept: application/json" \
-H "Content-type: application/json" \
-d "{\"email\": \"admin@test.com\", \"password\": \"toptal\" }"
{
  "data": {
    "id":1,
    "name":"Administrator",
    "email":"admin@test.com",
    "created_at":"2017-04-25 01:05:34",
    "updated_at":"2017-04-25 02:50:40",
    "api_token":"JII7q0BSijLOrzaOSm5Dr5hW9cJRZAJKOzvDlxjKCXepwAeZ7JR6YP5zQqnw"
  }
}
```

To send the token in a request, you can do it by sending an attribute `api_token` in the payload or as a bearer token in the request headers in the form of `Authorization`:

Bearer

JII7q0BSijLOrzaOSm5Dr5hW9cJRZAJKOzvDlxjKCXepwAe
Z7JR6YP5zQqnw.

Module 7

Logging Out

With our current strategy, if the token is wrong or missing, the user should receive an unauthenticated response (which we'll implement in the next section). So for a simple logout endpoint, we'll send in the token and it will be removed on the database.

routes/api.php:

```
Route::post('logout', 'Auth\LoginController@logout');
```

Auth\LoginController.php:

```
public function logout(Request $request)
{
```

```
    $user = Auth::guard('api')->user();
```

```
    if ($user) {
```

```
        $user->api_token = null;
        $user->save();
    }
```

```
    return response()->json(['data' => 'User logged out.'], 200);
}
```

Module 7

Using this strategy, whatever token the user has will be invalid, and the API will deny access (using middlewares, as explained in the next section). This needs to be coordinated with the front-end to avoid the user remaining logged without having access to any content.

Using Middlewares to Restrict Access

With the `api_token` created, we can toggle the authentication middleware in the routes file:

```
Route::middleware('auth:api')
    ->get('/user', function (Request $request) {
        return $request->user();
});
```

We can access the current user using the `$request->user()` method or through the Auth facade

```
Auth::guard('api')->user(); // instance of the logged user
Auth::guard('api')->check(); // if a user is authenticated
Auth::guard('api')->id(); // the id of the authenticated user
```

And we get a result like this:

Module 7

This is because we need to edit the current `unauthenticated` method on our Handler class. The current version returns a JSON only if the request has the `Accept: application/json` header, so let's change it:

```
protected function  
unauthenticated($request,  
AuthenticationException $exception)  
{  
    return response()->json(['error' =>  
    'Unauthenticated'], 401);  
}
```

With that fixed, we can go back to the article endpoints to wrap them in the `auth:api` middleware. We can do that by using route groups:

Whoops, looks like something went wrong.

1/1 [InvalidArgumentException](#) in [UrlGenerator.php](#) line 304:
Route [login] not defined.

1. in [UrlGenerator.php](#) line 304
2. at [UrlGenerator->route\('login', array\(\), true\)](#) in [helpers.php](#) line 741
3. at [route\('login'\)](#) in [Handler.php](#) line 74
4. at [Handler->unauthenticated\(object\(Request\), object\(AuthenticationException\)\)](#) in [Handler.php](#) line 114
5. at [Handler->render\(object\(Request\), object\(AuthenticationException\)\)](#) in [Handler.php](#) line 58
6. at [Handler->render\(object\(Request\), object\(AuthenticationException\)\)](#) in [Pipeline.php](#) line 82
7. at [Pipeline->handleException\(object\(Request\), object\(AuthenticationException\)\)](#) in [Pipeline.php](#) line 55
8. at [Pipeline->Illuminate\Routing\{closure}\(object\(Request\)\)](#) in [ThrottleRequests.php](#) line 49
9. at [ThrottleRequests->handle\(object\(Request\), object\(Closure\), '60', '1'\)](#) in [Pipeline.php](#) line 148
10. at [Pipeline->Illuminate\Pipeline\{closure}\(object\(Request\)\)](#) in [Pipeline.php](#) line 53
11. at [Pipeline->Illuminate\Routing\{closure}\(object\(Request\)\)](#) in [Pipeline.php](#) line 102
12. at [Pipeline->then\(object\(Closure\)\)](#) in [Router.php](#) line 581
13. at [Router->runRouteWithinStack\(object\(Route\), object\(Request\)\)](#) in [Router.php](#) line 520
14. at [Router->dispatchToRoute\(object\(Request\)\)](#) in [Router.php](#) line 498



Module 7

```
Route::group(['middleware' => 'auth:api'], function() {
    Route::get('articles', 'ArticleController@index');
    Route::get('articles/{article}', 'ArticleController@show');
    Route::post('articles', 'ArticleController@store');
    Route::put('articles/{article}', 'ArticleController@update');
    Route::delete('articles/{article}', 'ArticleController@delete');
});
```

- This way we don't have to set the middleware for each of the routes. It doesn't save a lot of time right now, but as the project grows it helps to keep the routes DRY.
- **Testing Our Endpoints**
- Laravel includes integration with PHPUnit out of the box with a `phpunit.xml` already set up. The framework also provides us with several helpers and extra assertions that makes our lives much easier, especially for testing APIs.
- There are a number of external tools you can use to test your API; however, testing inside Laravel is a much better alternative—we can have all the benefits of testing an API structure and results while retaining full control of the database. For the list endpoint, for example, we could run a couple of factories and assert the response contains those resources.

Module 7

To get started, we'll need to tweak a few settings to use an in-memory SQLite database. Using that will make our tests run lightning fast, but the trade-off is that some migration commands (constraints, for example) will not work properly in that particular setup. I advise moving away from SQLite in testing when you start getting migration errors or if you prefer a stronger set of tests instead of performant runs.

We'll also run the migrations before each test. This setup will allow us to build the database for each test and then destroy it, avoiding any type of dependency between tests.

In our config/database.php file, we'll need to set up the database field in the sqlite configuration to
:memory::

```
...
'connections' => [
    'sqlite' => [
        'driver' => 'sqlite',
        'database' => ':memory;',
        'prefix' => '',
    ],
    ...
]
```

Module 7

Then enable SQLite in phpunit.xml by adding the environment variable DB_CONNECTION:

```
<php>
  <env name="APP_ENV" value="testing"/>
  <env name="CACHE_DRIVER" value="array"/>
  <env name="SESSION_DRIVER" value="array"/>
  <env name="QUEUE_DRIVER" value="sync"/>
  <env name="DB_CONNECTION" value="sqlite"/>
</php>
```

With that out of the way, all that's left is configuring our base TestCase class to use migrations and seed the database before each test. To do so, we need to add the DatabaseMigrations trait, and then add an Artisan call on our setUp() method. Here's the class after the changes:

Module 7

```
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\TestCase as BaseTestCase;
use Illuminate\Support\Facades\Artisan;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication, DatabaseMigrations;

    public function setUp()
    {
        parent::setUp();
        Artisan::call('db:seed');
    }
}
```

One last thing that I like to do is to add the test command to composer.json:

```
"scripts": {
    "test" : [
        "vendor/bin/phpunit"
    ],
    ...
},
```

The test command will be available like this:

```
$ composer test
```

Module 7

Setting Up Factories for Our Tests

Factories will allow us to quickly create objects with the right data for testing. They're located in the database/factories folder. Laravel comes out of the box with a factory for the User class, so let's add one for the Article class:

```
$factory->define(App\Article::class, function (Faker\Generator $faker) {  
    return [  
        'title' => $faker->sentence,  
        'body' => $faker->paragraph,  
    ];  
});
```

The [Faker](#) library is already injected to help us create the correct format of random data for our models.

Our First Tests

We can use Laravel's assert methods to easily hit an endpoint and evaluate its response. Let's create our first test, the login test, using the following command:

```
$ php artisan make:test Feature/LoginTest
```

Example

<https://github.com/TopsCode/Laravel/tree/master/Module6-7>

Module 7

And here is our test:

```
class LoginTest extends TestCase
{
    public function testRequiresEmailAndLogin()
    {
        $this->json('POST', 'api/login')
            ->assertStatus(422)
            ->assertJson([
                'email' => ['The email field is required.'],
                'password' => ['The password field is required.'],
            ]);
    }
}
```

```
public function
testUserLoginsSuccessfully()
{
    $user = factory(User::class)-
>create([
        'email' => 'testlogin@user.com',
        'password' =>
bcrypt('toptal123'),
    ]);

    $payload = ['email' =>
'testlogin@user.com', 'password' =>
'toxtal123'];
}
```

Module 7

```
$this->json('POST', 'api/login', $payload)
    ->assertStatus(200)
    ->assertJsonStructure([
        'data' => [
            'id',
            'name',
            'email',
            'created_at',
            'updated_at',
            'api_token',
        ],
    ]);
}
}
```

These methods test a couple of simple cases. The json() method hits the endpoint and the other asserts are pretty self explanatory. One detail about assertJson(): this method converts the response into an array searches for the argument, so the order is important. You can chain multiple assertJson() calls in that case.

Now, let's create the register endpoint test and write a couple for that endpoint:

```
$ php artisan make:test RegisterTest
```

Module 7

```
class RegisterTest extends TestCase
{
    public function testsRegistersSuccessfully()
    {
        $payload = [
            'name' => 'John',
            'email' => 'john@toptal.com',
            'password' => 'toptal123',
            'password_confirmation' => 'toptal123',
        ];
    }
}
```

```
$this->json('post', '/api/register', $payload)
    ->assertStatus(201)
    ->assertJsonStructure([
        'data' => [
            'id',
            'name',
            'email',
            'created_at',
            'updated_at',
            'api_token',
        ],
    ]); }
```

Module 7

```
public function testsRequiresPasswordEmailAndName()
{
    $this->json('post', '/api/register')
        ->assertStatus(422)
        ->assertJson([
            'name' => ['The name field is required.'],
            'email' => ['The email field is required.'],
            'password' => ['The password field is required.'],
        ]);
}
```

```
public function testsRequirePasswordConfirmation()
{
    $payload = [
        'name' => 'John',
        'email' => 'john@toptal.com',
        'password' => 'toptal123',
    ];
    $this->json('post', '/api/register', $payload)
        ->assertStatus(422)
        ->assertJson([
            'password' => ['The password confirmation does not
match.']]);
}
```

Module 7

And lastly, the logout endpoint:

```
$ php artisan make:test LogoutTest
```

```
class LogoutTest extends TestCase
{
    public function testUserIsLoggedOutProperly()
    {
        $user = factory(User::class)->create(['email' =>
'user@test.com']);
        $token = $user->generateToken();
        $headers = ['Authorization' => "Bearer $token"];
    }
}
```

```
$this->json('get', '/api/articles', [], $headers)-
>assertStatus(200);
$this->json('post', '/api/logout', [], $headers)-
>assertStatus(200);

$user = User::find($user->id);
$this->assertEquals(null, $user->api_token);
}
```

Module 7

```
public function testUserWithNullToken()
{
    // Simulating login
    $user = factory(User::class)->create(['email' => 'user@test.com']);
    $token = $user->generateToken();
    $headers = ['Authorization' => "Bearer $token"];

    // Simulating logout
    $user->api_token = null;
    $user->save();

    $this->json('get', '/api/articles', [], $headers)->assertStatus(401);
}
```

- It's important to note that, during testing, the Laravel application is not instantiated again on a new request. Which means that when we hit the authentication middleware, it saves the current user inside the TokenGuard instance to avoid hitting the database again. A wise choice, however—in this case, it means have to split the logout test into two, to avoid any issues with the previously cached user.

Module 7

- Testing the Article endpoints is straightforward as well:

```
class ArticleTest extends TestCase
{
    public function testsArticlesAreCreatedCorrectly()
    {
        $user = factory(User::class)->create();
        $token = $user->generateToken();
        $headers = ['Authorization' => "Bearer $token"];
        $payload = [
            'title' => 'Lorem',
            'body' => 'Ipsum',
        ];

        $this->json('POST', '/api/articles', $payload, $headers)
            ->assertStatus(200)
            ->assertJson(['id' => 1, 'title' => 'Lorem', 'body' => 'Ipsum']);
    }

    public function testsArticlesAreUpdatedCorrectly()
    {
        $user = factory(User::class)->create();
        $token = $user->generateToken();
        $headers = ['Authorization' => "Bearer $token"];
```

Module 7

```
$article = factory(Article::class)->create([
    'title' => 'First Article',
    'body' => 'First Body',
]);

$payload = [
    'title' => 'Lorem',
    'body' => 'Ipsum',
];

$response = $this->json('PUT', '/api/articles/' . $article->id, $payload, $headers)
    ->assertStatus(200)
    ->assertJson([
        'id' => 1,
        'title' => 'Lorem',
        'body' => 'Ipsum'
    ]);
}

public function testsArtilcesAreDeletedCorrectly()
{
```

Module 7

```
$user = factory(User::class)->create();
$token = $user->generateToken();
$headers = ['Authorization' => "Bearer $token"];
$article = factory(Article::class)->create([
    'title' => 'First Article',
    'body' => 'First Body',
]);
$this->json('DELETE', '/api/articles/' . $article->id, [], $headers)
    ->assertStatus(204);
}

public function testArticlesAreListedCorrectly()
{
    factory(Article::class)->create([
        'title' => 'First Article',
        'body' => 'First Body'
    ]);
    factory(Article::class)->create([
        'title' => 'Second Article',
        'body' => 'Second Body'
    ]);
}
```

Module 7

- **Events Introduction**
- Laravel's events provide a simple observer pattern implementation, allowing you to subscribe and listen for various events that occur within your application. Event classes are typically stored in the app/Events directory, while their listeners are stored in app/Listeners. Don't worry if you don't see these directories in your application as they will be created for you as you generate events and listeners using Artisan console commands.
- Events serve as a great way to decouple various aspects of your application, since a single event can have multiple listeners that do not depend on each other. For example, you may wish to send a Slack notification to your user each time an order has shipped. Instead of coupling your order processing code to your Slack notification code, you can raise an App\Events\OrderShipped event which a listener can receive and use to dispatch a Slack notification.
- **Registering Events & Listeners**
- The App\Providers\EventServiceProvider included with your Laravel application provides a convenient place to register all of your application's event listeners. The `listen` property contains an array of all events (keys) and their listeners (values). You may add as many events to this array as your application requires. For example, let's add an OrderShipped event:

Module 7

```
use App\Events\OrderShipped;
use App\Listeners\SendShipmentNotification;

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    OrderShipped::class => [
        SendShipmentNotification::class,
    ],
];
```

- The `event:list` command may be used to display a list of all events and listeners registered by your application.
- [Generating Events & Listeners](#)
- Of course, manually creating the files for each event and listener is cumbersome. Instead, add listeners and events to your `EventServiceProvider` and use the `event:generate` Artisan command. This command will generate any events or listeners that are listed in your `EventServiceProvider` that do not already exist:
 - `php artisan event:generate`

Module 7

- Alternatively, you may use the make:event and make:listener Artisan commands to generate individual events and listeners:

- ```
php artisan make:event PodcastProcessed
```

- 

- ```
php artisan make:listener SendPodcastNotification --event=PodcastProcessed
```

- **Manually Registering Events**

- Typically, events should be registered via the EventServiceProvider \$listen array; however, you may also register class or closure based event listeners manually in the boot method of your EventServiceProvider:

```
use App\Events\PodcastProcessed;
use App\Listeners\SendPodcastNotification;
use Illuminate\Support\Facades\Event;
public function boot()
{
    Event::listen(
        PodcastProcessed::class,
        [SendPodcastNotification::class, 'handle']
    );

    Event::listen(function (PodcastProcessed $event) {
    });
}
```

Module 7

- **Queueable Anonymous Event Listeners**
- When registering closure based event listeners manually, you may wrap the listener closure within the `Illuminate\Events\queueable` function to instruct Laravel to execute the listener using the [queue](#):

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;
public function boot()
{
    Event::listen(queueable(function (PodcastProcessed $event) {
        //
    }));
}
```
- Like queued jobs, you may use the `onConnection`, `onQueue`, and `delay` methods to customize the execution of the queued listener:

```
Event::listen(queueable(function (PodcastProcessed $event) {
    //
})->onConnection('redis')->onQueue('podcasts')->delay(now()->addSeconds(10)));
```

Module 7

- If you would like to handle anonymous queued listener failures, you may provide a closure to the catch method while defining the queueable listener. This closure will receive the event instance and the Throwable instance that caused the listener's failure:

```
use App\Events\PodcastProcessed;  
use function Illuminate\Events\queueable;  
use Illuminate\Support\Facades\Event;  
use Throwable;
```

```
Event::listen(queueable(function (PodcastProcessed $event) {  
    //  
}))->catch(function (PodcastProcessed $event, Throwable $e) {  
    // The queued listener failed...  
});
```

- **Wildcard Event Listeners**

- You may even register listeners using the * as a wildcard parameter, allowing you to catch multiple events on the same listener. Wildcard listeners receive the event name as their first argument and the entire event data array as their second argument:

```
Event::listen('event.*', function ($eventName, array $data) {  
    //  
});
```

Module 7

- **Event Discovery**
- Instead of registering events and listeners manually in the `$listen` array of the `EventServiceProvider`, you can enable automatic event discovery. When event discovery is enabled, Laravel will automatically find and register your events and listeners by scanning your application's `Listeners` directory. In addition, any explicitly defined events listed in the `EventServiceProvider` will still be registered.
- Laravel finds event listeners by scanning the listener classes using PHP's reflection services. When Laravel finds any listener class method that begins with `handle`, Laravel will register those methods as event listeners for the event that is type-hinted in the method's signature:

```
use App\Events\PodcastProcessed;
```

```
class SendPodcastNotification
{
    public function handle(PodcastProcessed $event)
    {
        //
    }
}
```

Module 7

- Event discovery is disabled by default, but you can enable it by overriding the `shouldDiscoverEvents` method of your application's `EventServiceProvider`:

```
/**  
 * Determine if events and listeners should be automatically discovered.  
 *  
 * @return bool  
 */  
public function shouldDiscoverEvents()  
{  
    return true;  
}
```

- By default, all listeners within your application's `app/Listeners` directory will be scanned. If you would like to define additional directories to scan, you may override the `discoverEventsWithin` method in your `EventServiceProvider`:

```
protected function discoverEventsWithin()  
{  
    return [  
        $this->app->path('Listeners'),  
    ];  
}
```

Module 7

- [Event Discovery In Production](#)
 - In production, it is not efficient for the framework to scan all of your listeners on every request. Therefore, during your deployment process, you should run the event:cache Artisan command to cache a manifest of all of your application's events and listeners. This manifest will be used by the framework to speed up the event registration process. The event:clear command may be used to destroy the cache.
- [Defining Events](#)
 - An event class is essentially a data container which holds the information related to the event. For example, let's assume an App\Events\OrderShipped event receives an [Eloquent ORM](#) object:

```
<?php  
  
namespace App\Events;  
  
use App\Models\Order;  
use Illuminate\\Broadcasting\\InteractsWithSockets;  
use Illuminate\\Foundation\\Events\\Dispatchable;  
use Illuminate\\Queue\\SerializesModels;
```

Module 7

```
class OrderShipped
{
    use Dispatchable, InteractsWithSockets, SerializesModels;
    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    public $order;

    /**
     * Create a new event instance.
     *
     * @param \App\Models\Order $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }
}
```

Module 7

- As you can see, this event class contains no logic. It is a container for the App\Models\Order instance that was purchased. The SerializesModels trait used by the event will gracefully serialize any Eloquent models if the event object is serialized using PHP's serialize function, such as when utilizing [queued listeners](#).
- **Defining Listeners**
- Next, let's take a look at the listener for our example event. Event listeners receive event instances in their handle method. The event:generate and make:listener Artisan commands will automatically import the proper event class and type-hint the event on the handle method. Within the handle method, you may perform any actions necessary to respond to the event:

```
<?php  
namespace App\Listeners;  
  
use App\Events\OrderShipped;  
  
class SendShipmentNotification  
{  
    /**  
     * Create the event listener.  
     *  
     * @return void  
     */
```

Module 7

```
public function __construct()
{
    //
}
/** 
 * Handle the event.
 *
 * @param \App\Events\OrderShipped $event
 * @return void
 */
public function handle(OrderShipped $event)
{
    // Access the order using $event->order...
}
```

- Your event listeners may also type-hint any dependencies they need on their constructors. All event listeners are resolved via the Laravel [service container](#), so dependencies will be injected automatically.
- [**Stopping The Propagation Of An Event**](#)
- Sometimes, you may wish to stop the propagation of an event to other listeners. You may do so by returning false from your listener's handle method.

Module 7

- **Queued Event Listeners**
- Queueing listeners can be beneficial if your listener is going to perform a slow task such as sending an email or making an HTTP request. Before using queued listeners, make sure to [configure your queue](#) and start a queue worker on your server or local development environment.
- To specify that a listener should be queued, add the ShouldQueue interface to the listener class. Listeners generated by the event:generate and make:listener Artisan commands already have this interface imported into the current namespace so you can use it immediately:

```
<?php  
  
namespace App\Listeners;  
use App\Events\OrderShipped;  
use Illuminate\Contracts\Queue\ShouldQueue;  
  
class SendShipmentNotification implements ShouldQueue  
{  
    //  
}
```

- That's it! Now, when an event handled by this listener is dispatched, the listener will automatically be queued by the event dispatcher using Laravel's [queue system](#). If no exceptions are thrown when the listener is executed by the queue, the queued job will automatically be deleted after it has finished processing.

Module 7

Customizing The Queue Connection & Queue Name

If you would like to customize the queue connection, queue name, or queue delay time of an event listener, you may define the `$connection`, `$queue`, or `$delay` properties on your listener class:

```
<?php  
  
namespace App\Listeners;  
  
use App\Events\OrderShipped;  
use Illuminate\Contracts\Queue\ShouldQueue;  
  
class SendShipmentNotification implements ShouldQueue  
{  
    public $connection = 'sq';  
  
    public $queue = 'listeners';  
  
    public $delay = 60;  
}
```

Module 7

- If you would like to define the listener's queue connection or queue name at runtime, you may define `viaConnection` or `viaQueue` methods on the listener:

```
/**  
 * Get the name of the listener's queue  
 * connection.  
 *  
 * @return string  
 */  
public function viaConnection()  
{  
    return 'sq';  
}
```

```
/**  
 * Get the name of the listener's queue.  
 *  
 * @return string  
 */  
public function viaQueue()  
{  
    return 'listeners';  
}
```

Module 7

- [**Conditionally Queueing Listeners**](#)
- Sometimes, you may need to determine whether a listener should be queued based on some data that are only available at runtime. To accomplish this, a `shouldQueue` method may be added to a listener to determine whether the listener should be queued. If the `shouldQueue` method returns `false`, the listener will not be executed:

```
<?php  
namespace App\Listeners;  
  
use App\Events\OrderCreated;  
use Illuminate\Contracts\Queue\ShouldQueue;  
  
class RewardGiftCard implements ShouldQueue  
{  
    public function handle(OrderCreated $event)  
    {  
        //  
    }  
    public function shouldQueue(OrderCreated $event)  
    {  
        return $event->order->subtotal >= 5000;  
    }  
}
```

Module 7

- [**Manually Interacting With The Queue**](#)
- If you need to manually access the listener's underlying queue job's delete and release methods, you may do so using the Illuminate\Queue\InteractsWithQueue trait. This trait is imported by default on generated listeners and provides access to these methods:

```
<?php  
namespace App\Listeners;  
  
use App\Events\OrderShipped;  
use Illuminate\Contracts\Queue\ShouldQueue;  
use Illuminate\Queue\InteractsWithQueue;  
  
class SendShipmentNotification implements ShouldQueue  
{  
    use InteractsWithQueue;  
    public function handle(OrderShipped $event)  
    {  
        if (true) {  
            $this->release(30);  
        }  
    }  
}
```

-

Module 7

- **Queued Event Listeners & Database Transactions**
- When queued listeners are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your listener depends on these models, unexpected errors can occur when the job that dispatches the queued listener is processed.
- If your queue connection's `after_commit` configuration option is set to false, you may still indicate that a particular queued listener should be dispatched after all open database transactions have been committed by defining an `$afterCommit` property on the listener class:

```
<?php  
  
namespace App\Listeners;  
  
use Illuminate\Contracts\Queue\ShouldQueue;  
use Illuminate\Queue\InteractsWithQueue;  
class SendShipmentNotification implements ShouldQueue  
{  
    use InteractsWithQueue;  
  
    public $afterCommit = true;  
}
```

Module 7

- To learn more about working around these issues, please review the documentation regarding [queued jobs and database transactions](#).
- **Handling Failed Jobs**
- Sometimes your queued event listeners may fail. If queued listener exceeds the maximum number of attempts as defined by your queue worker, the failed method will be called on your listener. The failed method receives the event instance and the Throwable that caused the failure:

```
<?php  
namespace App\Listeners;  
  
use App\Events\OrderShipped;  
use Illuminate\Contracts\Queue\ShouldQueue;  
use Illuminate\Queue\InteractsWithQueue;  
  
class SendShipmentNotification implements ShouldQueue  
{  
    use InteractsWithQueue;  
  
    public function handle(OrderShipped $event)  
    {  
        //  
    }  
}
```

Module 7

```
public function failed(OrderShipped $event, $exception)
{
    //
}
```

- **Specifying Queued Listener Maximum Attempts**
- If one of your queued listeners is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a listener may be attempted.
- You may define `$tries` property on your listener class to specify how many times the listener may be attempted before it is considered to have failed:

```
<?php
namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;
    public $tries = 5;
}
```

Module 7

As an alternative to defining how many times a listener may be attempted before it fails, you may define a time at which the listener should no longer be attempted. This allows a listener to be attempted any number of times within a given time frame. To define the time at which a listener should no longer be attempted, add a `retryUntil` method to your listener class. This method should return a `DateTime` instance:

```
public function retryUntil()
{
    return now()->addMinutes(5);
}
```

Dispatching Events

To dispatch an event, you may call the static `dispatch` method on the event. This method is made available on the event by the `Illuminate\Foundation\Events\Dispatchable` trait. Any arguments passed to the `dispatch` method will be passed to the event's constructor:

```
<?php
namespace App\Http\Controllers;
use App\Events\OrderShipped;
use App\Http\Controllers\Controller;
use App\Models\Order;
use Illuminate\Http\Request;

class OrderShipmentController extends Controller
```

Module 7

- ```
{
 /**
 * Ship the given order.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
 public function store(Request $request)
 {
 $order = Order::findOrFail($request->order_id);

 // Order shipment logic...

 OrderShipped::dispatch($order);
 }
}
```
- When testing, it can be helpful to assert that certain events were dispatched without actually triggering their listeners. Laravel's [built-in testing helpers](#) makes it a cinch.

# Module 7

- Event Subscribers

## Writing Event Subscribers

Event subscribers are classes that may subscribe to multiple events from within the subscriber class itself, allowing you to define several event handlers within a single class. Subscribers should define a subscribe method, which will be passed an event dispatcher instance. You may call the listen method on the given dispatcher to register event listeners:

```
<?php
namespace App\Listeners;
class UserEventSubscriber
{
 public function handleUserLogin($event) {}
 public function handleUserLogout($event) {}
```

```
public function subscribe($events)
{
 $events->listen(
 'Illuminate\Auth\Events\Login',
 [UserEventSubscriber::class, 'handleUserLogin']
);
 $events->listen(
 'Illuminate\Auth\Events\Logout',
 [UserEventSubscriber::class, 'handleUserLogout']); } }
```

# Module 7

- [\*\*Registering Event Subscribers\*\*](#)
- After writing the subscriber, you are ready to register it with the event dispatcher. You may register subscribers using the `$subscribe` property on the `EventServiceProvider`. For example, let's add the `UserEventSubscriber` to the list:

```
<?php

namespace App\Providers;

use App\Listeners\UserEventSubscriber;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
 /**
 * protected $listen = [
 * ...
 *];
 *
 * protected $subscribe = [
 * UserEventSubscriber::class,
 *];
 */

 protected $subscribe = [
 UserEventSubscriber::class,
];
}
```

# Module 7

- **Mail Introduction**
- Sending email doesn't have to be complicated. Laravel provides a clean, simple email API powered by the popular [SwiftMailer](#) library. Laravel and SwiftMailer provide drivers for sending email via SMTP, Mailgun, Postmark, Amazon SES, and sendmail, allowing you to quickly get started sending mail through a local or cloud based service of your choice.
- **Configuration**
- Laravel's email services may be configured via your application's config/mail.php configuration file. Each mailer configured within this file may have its own unique configuration and even its own unique "transport", allowing your application to use different email services to send certain email messages. For example, your application might use Postmark to send transactional emails while using Amazon SES to send bulk emails.
- Within your mail configuration file, you will find a mailers configuration array. This array contains a sample configuration entry for each of the major mail drivers / transports supported by Laravel, while the default configuration value determines which mailer will be used by default when your application needs to send an email message.

# Module 7

## Driver / Transport Prerequisites

The API based drivers such as Mailgun and Postmark are often simpler and faster than sending mail via SMTP servers. Whenever possible, we recommend that you use one of these drivers. All of the API based drivers require the Guzzle HTTP library, which may be installed via the Composer package manager:

```
composer require guzzlehttp/guzzle
```

## Mailgun Driver

To use the Mailgun driver, first install the Guzzle HTTP library. Then, set the default option in your config/mail.php configuration file to mailgun. Next, verify that your config/services.php configuration file contains the following options:

```
'mailgun' => [
 'domain' => env('MAILGUN_DOMAIN'),
 'secret' => env('MAILGUN_SECRET'),
,
```

If you are not using the United States [Mailgun region](#), you may define your region's endpoint in the services configuration file:

```
'mailgun' => [
 'domain' => env('MAILGUN_DOMAIN'),
 'secret' => env('MAILGUN_SECRET'),
 'endpoint' => env('MAILGUN_ENDPOINT', 'api.eu.mailgun.net'),
,
```

# Module 7

## Postmark Driver

To use the Postmark driver, install Postmark's SwiftMailer transport via Composer:

```
composer require wildbit/swiftmailer-postmark
```

Next, install the Guzzle HTTP library and set the default option in your config/mail.php configuration file to postmark. Finally, verify that your config/services.php configuration file contains the following options:

```
'postmark' => [
 'token' => env('POSTMARK_TOKEN'),
,
```

If you would like to specify the Postmark message stream that should be used by a given mailer, you may add the message\_stream\_id configuration option to the mailer's configuration array. This configuration array can be found in your application's config/mail.php configuration file:

```
'postmark' => [
 'transport' => 'postmark',
 'message_stream_id' => env('POSTMARK_MESSAGE_STREAM_ID'),
,
```

This way you are also able to set up multiple Postmark mailers with different message streams.

## SES Driver

To use the Amazon SES driver you must first install the Amazon AWS SDK for PHP. You may install this library via the Composer package manager:

# Module 7

```
composer require aws/aws-sdk-php
```

- Next, set the default option in your config/mail.php configuration file to ses and verify that your config/services.php configuration file contains the following options:

```
'ses' => [
 'key' => env('AWS_ACCESS_KEY_ID'),
 'secret' => env('AWS_SECRET_ACCESS_KEY'),
 'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
,
```

- If you would like to define [additional options](#) that Laravel should pass to the AWS SDK's SendRawEmail method when sending an email, you may define an options array within your ses configuration:

```
'ses' => [
 'key' => env('AWS_ACCESS_KEY_ID'),
 'secret' => env('AWS_SECRET_ACCESS_KEY'),
 'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
 'options' => [
 'ConfigurationSetName' => 'MyConfigurationSet',
 'Tags' => [
 ['Name' => 'foo', 'Value' => 'bar'],
],
],
,
```

# Module 7

## Generating Mailables

When building Laravel applications, each type of email sent by your application is represented as a "mailable" class. These classes are stored in the app/Mail directory. Don't worry if you don't see this directory in your application, since it will be generated for you when you create your first mailable class using the make:mail Artisan command:

```
php artisan make:mail OrderShipped
```

## Writing Mailables

Once you have generated a mailable class, open it up so we can explore its contents. First, note that all of a mailable class' configuration is done in the build method. Within this method, you may call various methods such as from, subject, view, and attach to configure the email's presentation and delivery.

## Configuring The Sender

### Using The from Method

First, let's explore configuring the sender of the email. Or, in other words, who the email is going to be "from". There are two ways to configure the sender. First, you may use the from method within your mailable class' build method:

# Module 7

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
 return $this->from('example@example.com', 'Example')
 ->view('emails.orders.shipped');
}
```

## Using A Global from Address

However, if your application uses the same "from" address for all of its emails, it can become cumbersome to call the from method in each mailable class you generate. Instead, you may specify a global "from" address in your config/mail.php configuration file. This address will be used if no other "from" address is specified within the mailable class:

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

In addition, you may define a global "reply\_to" address within your config/mail.php configuration file:

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

# Module 7

## Configuring The View

Within a mailable class' build method, you may use the view method to specify which template should be used when rendering the email's contents. Since each email typically uses a [Blade template](#) to render its contents, you have the full power and convenience of the Blade templating engine when building your email's HTML:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
 return $this->view('emails.orders.shipped');
}
```

You may wish to create a resources/views/emails directory to house all of your email templates; however, you are free to place them wherever you wish within your resources/views directory.

# Module 7

- [Plain Text Emails](#)
- If you would like to define a plain-text version of your email, you may use the text method. Like the view method, the text method accepts a template name which will be used to render the contents of the email. You are free to define both an HTML and plain-text version of your message:

```
public function build()
{
 return $this->view('emails.orders.shipped')
 ->text('emails.orders.shipped_plain');
}
```
- [View Data](#)
- [Via Public Properties](#)
- Typically, you will want to pass some data to your view that you can utilize when rendering the email's HTML. There are two ways you may make data available to your view. First, any public property defined on your mailable class will automatically be made available to the view. So, for example, you may pass data into your mailable class' constructor and set that data to public properties defined on the class:

```
<?php
```

```
namespace App\Mail;
```

# Module 7

```
use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
 use Queueable, SerializesModels;

 public $order;
 public function __construct(Order $order)
 {
 $this->order = $order;
 }

 public function build()
 {
 return $this->view('emails.orders.shipped');
 }
}
```

# Module 7

- Once the data has been set to a public property, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

```
<div>
 Price: {{ $order->price }}
</div>
```

- Via The with Method:**

- If you would like to customize the format of your email's data before it is sent to the template, you may manually pass your data to the view via the with method. Typically, you will still pass data via the mailable class' constructor; however, you should set this data to protected or private properties so the data is not automatically made available to the template. Then, when calling the with method, pass an array of data that you wish to make available to the template:

```
<?php
namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
class OrderShipped extends Mailable
{
 use Queueable, SerializesModels;
```

# Module 7

```
protected $order;

public function __construct(Order $order)
{
 $this->order = $order;
}

public function build()
{
 return $this->view('emails.orders.shipped')
 ->with([
 'orderName' => $this->order->name,
 'orderPrice' => $this->order->price,
]);
}
```

- Once the data has been passed to the with method, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

```
<div>
 Price: {{ $orderPrice }}
</div>
```

# Module 7

- **Attachments**

- To add attachments to an email, use the attach method within the mailable class' build method. The attach method accepts the full path to the file as its first argument:

```
public function build()
{
 return $this->view('emails.orders.shipped')
 ->attach('/path/to/file');
}
```

- When attaching files to a message, you may also specify the display name and / or MIME type by passing an array as the second argument to the attach method:

```
public function build()
{
 return $this->view('emails.orders.shipped')
 ->attach('/path/to/file', [
 'as' => 'name.pdf',
 'mime' => 'application/pdf',
]);
}
```

# Module 7

- [\*\*Attaching Files From Disk\*\*](#)
- If you have stored a file on one of your [filesystem disks](#), you may attach it to the email using the attachFromStorage method:
  - public function build()
  - {
  - return \$this->view('emails.orders.shipped')
  - ->attachFromStorage('/path/to/file');
  - }
- If necessary, you may specify the file's attachment name and additional options using the second and third arguments to the attachFromStorage method:
  - public function build()
  - {
  - return \$this->view('emails.orders.shipped')
  - ->attachFromStorage('/path/to/file', 'name.pdf', [
  - 'mime'=> 'application/pdf'
  - ]);
  - }
- The attachFromStorageDisk method may be used if you need to specify a storage disk other than your default disk:

# Module 7

```
public function build()
{
 return $this->view('emails.orders.shipped')
 ->attachFromStorageDisk('s3', '/path/to/file');
}
```

- **Raw Data Attachments**

- The attachData method may be used to attach a raw string of bytes as an attachment. For example, you might use this method if you have generated a PDF in memory and want to attach it to the email without writing it to disk. The attachData method accepts the raw data bytes as its first argument, the name of the file as its second argument, and an array of options as its third argument:

```
public function build()
{
 return $this->view('emails.orders.shipped')
 ->attachData($this->pdf, 'name.pdf', [
 'mime' => 'application/pdf',
]);
}
```

# Module 7

- **Inline Attachments**
- Embedding inline images into your emails is typically cumbersome; however, Laravel provides a convenient way to attach images to your emails. To embed an inline image, use the embed method on the \$message variable within your email template. Laravel automatically makes the \$message variable available to all of your email templates, so you don't need to worry about passing it in manually:

```
<body>
 Here is an image:

</body>
```
- The \$message variable is not available in plain-text message templates since plain-text messages do not utilize inline attachments.

## **Embedding Raw Data Attachments**

- If you already have a raw image data string you wish to embed into an email template, you may call the embedData method on the \$message variable. When calling the embedData method, you will need to provide a filename that should be assigned to the embedded image:

```
<body>
 Here is an image from raw data:
```

```

</body>
```

# Module 7

- [Customizing The SwiftMailer Message](#)
- The `withSwiftMessage` method of the `Mailable` base class allows you to register a closure which will be invoked with the `SwiftMailer` message instance before sending the message. This gives you an opportunity to deeply customize the message before it is delivered

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
 $this->view('emails.orders.shipped');

 $this->withSwiftMessage(function ($message) {
 $message->getHeaders()->addTextHeader(
 'Custom-Header', 'Header Value'
);
 });

 return $this;
}
```

# Module 7

- **Markdown Mailables**
- Markdown mailable messages allow you to take advantage of the pre-built templates and components of [mail notifications](#) in your mailables. Since the messages are written in Markdown, Laravel is able to render beautiful, responsive HTML templates for the messages while also automatically generating a plain-text counterpart.
- **Generating Markdown Mailables**
- To generate a mailable with a corresponding Markdown template, you may use the --markdown option of the make:mail Artisan command:
  - `php artisan make:mail OrderShipped --markdown=emails.orders.shipped`
- Then, when configuring the mailable within its build method, call the markdown method instead of the view method. The markdown method accepts the name of the Markdown template and an optional array of data to make available to the template:

```
public function build()
{
 return $this->from('example@example.com')
 ->markdown('emails.orders.shipped', [
 'url' => $this->orderUrl,
]);
}
```

# Module 7

- Writing Markdown Messages
- Markdown mailables use a combination of Blade components and Markdown syntax which allow you to easily construct mail messages while leveraging Laravel's pre-built email UI components:

```
@component('mail::message')
Order Shipped
```

Your order has been shipped!

```
@component('mail::button', ['url' => $url])
View Order
@endcomponent
```

```
Thanks,

{{ config('app.name') }}
@endcomponent
```

- Do not use excess indentation when writing Markdown emails. Per Markdown standards, Markdown parsers will render indented content as code blocks.

# Module 7

## Button Component

The button component renders a centered button link. The component accepts two arguments, a url and an optional color. Supported colors are primary, success, and error. You may add as many button components to a message as you wish:

```
@component('mail::button', ['url' => $url, 'color' => 'success'])
View Order
@endcomponent
```

## Panel Component

The panel component renders the given block of text in a panel that has a slightly different background color than the rest of the message. This allows you to draw attention to a given block of text:

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

## Table Component

The table component allows you to transform a Markdown table into an HTML table. The component accepts the Markdown table as its content. Table column alignment is supported using the default Markdown table alignment syntax:

# Module 7

```
@component('mail::table')
| Laravel | Table | Example |
| ----- | :-----: | -----:|
| Col 2 is | Centered | $10 |
| Col 3 is | Right-Aligned | $20 |
@endcomponent
```

## Customizing The Components

You may export all of the Markdown mail components to your own application for customization. To export the components, use the vendor:publish Artisan command to publish the laravel-mail asset tag:

```
php artisan vendor:publish --tag=laravel-mail
```

This command will publish the Markdown mail components to the resources/views/vendor/mail directory. The mail directory will contain an html and a text directory, each containing their respective representations of every available component. You are free to customize these components however you like.

## Customizing The CSS

After exporting the components, the resources/views/vendor/mail/html/themes directory will contain a default.css file. You may customize the CSS in this file and your styles will automatically be converted to inline CSS styles within the HTML representations of your Markdown mail messages.

# Module 7

If you would like to build an entirely new theme for Laravel's Markdown components, you may place a CSS file within the `html/themes` directory. After naming and saving your CSS file, update the `theme` option of your application's `config/mail.php` configuration file to match the name of your new theme.

To customize the theme for an individual mailable, you may set the `$theme` property of the mailable class to the name of the theme that should be used when sending that mailable.

## Sending Mail

To send a message, use the `to` method on the [Mail facade](#). The `to` method accepts an email address, a user instance, or a collection of users. If you pass an object or collection of objects, the mailer will automatically use their email and name properties when determining the email's recipients, so make sure these attributes are available on your objects. Once you have specified your recipients, you may pass an instance of your mailable class to the `send` method:

```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Mail\OrderShipped;
use App\Models\Order;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;
class OrderShipmentController extends Controller
{
```

# Module 7

```
public function store(Request $request)
{
 $order = Order::findOrFail($request->order_id);

 // Ship the order...

 Mail::to($request->user())->send(new OrderShipped($order));
}
```

- You are not limited to just specifying the "to" recipients when sending a message. You are free to set "to", "cc", and "bcc" recipients by chaining their respective methods together:

```
Mail::to($request->user())
 ->cc($moreUsers)
 ->bcc($evenMoreUsers)
 ->send(new OrderShipped($order));
```

- **Looping Over Recipients**
- Occasionally, you may need to send a mailable to a list of recipients by iterating over an array of recipients / email addresses. However, since the to method appends email addresses to the mailable's list of recipients, each iteration through the loop will send another email to every previous recipient. Therefore, you should always re-create the mailable instance for each recipient:

# Module 7

```
foreach(['taylor@example.com', 'dries@example.com'] as $recipient) {
 Mail::to($recipient)->send(new OrderShipped($order)); }
```

- [\*\*Sending Mail Via A Specific Mailer\*\*](#)
- By default, Laravel will send email using the mailer configured as the default mailer in your application's mail configuration file. However, you may use the mailer method to send a message using a specific mailer configuration:
  - Mail::mailer('postmark')
  - ->to(\$request->user())
  - ->send(new OrderShipped(\$order));
- [\*\*Queueing Mail\*\*](#)
- [\*\*Queueing A Mail Message\*\*](#)
- Since sending email messages can negatively impact the response time of your application, many developers choose to queue email messages for background sending. Laravel makes this easy using its built-in [unified queue API](#). To queue a mail message, use the queue method on the Mail facade after specifying the message's recipients:

```
Mail::to($request->user())
 ->cc($moreUsers)
 ->bcc($evenMoreUsers)
 ->queue(new OrderShipped($order));
```

# Module 7

- This method will automatically take care of pushing a job onto the queue so the message is sent in the background. You will need to [configure your queues](#) before using this feature.
- **Delayed Message Queueing**
  - If you wish to delay the delivery of a queued email message, you may use the later method. As its first argument, the later method accepts a DateTime instance indicating when the message should be sent:

```
Mail::to($request->user())
->cc($moreUsers)
->bcc($evenMoreUsers)
->later(now()->addMinutes(10), new OrderShipped($order));
```
- **Pushing To Specific Queues**
  - Since all mailable classes generated using the make:mail command make use of the Illuminate\Bus\Queueable trait, you may call the onQueue and onConnection methods on any mailable class instance, allowing you to specify the connection and queue name for the message:

# Module 7

```
$message = (new OrderShipped($order))
 ->onConnection('sq')
 ->onQueue('emails');
```

```
Mail::to($request->user())
 ->cc($moreUsers)
 ->bcc($evenMoreUsers)
 ->queue($message);
```

- **Queueing By Default**

- If you have mailable classes that you want to always be queued, you may implement the ShouldQueue contract on the class. Now, even if you call the send method when mailing, the mailable will still be queued since it implements the contract:

```
use Illuminate\Contracts\Queue\ShouldQueue;
```

```
class OrderShipped extends Mailable implements ShouldQueue
{
 //
}
```

# Module 7

- [\*\*Queued Mailables & Database Transactions\*\*](#)
- When queued mailables are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your mailable depends on these models, unexpected errors can occur when the job that sends the queued mailable is processed.
- If your queue connection's `after_commit` configuration option is set to false, you may still indicate that a particular queued mailable should be dispatched after all open database transactions have been committed by defining an `$afterCommit` property on the mailable class:

use Illuminate\Contracts\Queue\ShouldQueue;

```
class OrderShipped extends Mailable implements ShouldQueue
{
 public $afterCommit = true;
}
```

- To learn more about working around these issues, please review the documentation regarding [queued jobs and database transactions](#).
-

# Module 7

- [Rendering Mailables](#)
- Sometimes you may wish to capture the HTML content of a mailable without sending it. To accomplish this, you may call the render method of the mailable. This method will return the evaluated HTML content of the mailable as a string:

```
use App\Mail\InvoicePaid;
use App\Models\Invoice;
$invoice = Invoice::find(1);

return (new InvoicePaid($invoice))->render();
```

- [Previewing Mailables In The Browser](#)

- When designing a mailable's template, it is convenient to quickly preview the rendered mailable in your browser like a typical Blade template. For this reason, Laravel allows you to return any mailable directly from a route closure or controller. When a mailable is returned, it will be rendered and displayed in the browser, allowing you to quickly preview its design without needing to send it to an actual email address:

```
Route::get('/mailable', function () {
 $invoice = App\Models\Invoice::find(1);

 return new App\Mail\InvoicePaid($invoice);
});
```

# Module 7

- [Inline attachments](#) will not be rendered when a mailable is previewed in your browser. To preview these mailables, you should send them to an email testing application such as [MailHog](#) or [HELO](#).
- [\*\*Localizing Mailables\*\*](#)
  - Laravel allows you to send mailables in a locale other than the request's current locale, and will even remember this locale if the mail is queued.
  - To accomplish this, the Mail facade offers a locale method to set the desired language. The application will change into this locale when the mailable's template is being evaluated and then revert back to the previous locale when evaluation is complete:

```
Mail::to($request->user())->locale('es')->send(
 new OrderShipped($order)
)
```
- [\*\*User Preferred Locales\*\*](#)
  - Sometimes, applications store each user's preferred locale. By implementing the HasLocalePreference contract on one or more of your models, you may instruct Laravel to use this stored locale when sending mail:
  -

# Module 7

```
use Illuminate\Contracts\Translation\HasLocalePreference;
```

```
class User extends Model implements HasLocalePreference
{
 /**
 * Get the user's preferred locale.
 *
 * @return string
 */
 public function preferredLocale()
 {
 return $this->locale;
 }
}
```

- Once you have implemented the interface, Laravel will automatically use the preferred locale when sending mailables and notifications to the model. Therefore, there is no need to call the locale method when using this interface:

```
Mail::to($request->user())->send(new OrderShipped($order));
```

# Module 7

- **Testing Mailables**
- Laravel provides several convenient methods for testing that your mailables contain the content that you expect. These methods are: assertSeeInHtml, assertDontSeeInHtml, assertSeeInText, and assertDontSeeInText.
- As you might expect, the "HTML" assertions assert that the HTML version of your mailable contains a given string, while the "text" assertions assert that the plain-text version of your mailable contains a given string:

```
use App\Mail\InvoicePaid;
use App\Models\User;

public function test_mailable_content()
{
 $user = User::factory()->create();

 $mailable = new InvoicePaid($user);

 $mailable->assertSeeInHtml($user->email);
 $mailable->assertSeeInHtml('Invoice Paid');

 $mailable->assertSeeInText($user->email);
 $mailable->assertSeeInText('Invoice Paid');
}
```

# Module 7

- **Mail & Local Development**
  - When developing an application that sends email, you probably don't want to actually send emails to live email addresses.
  - Laravel provides several ways to "disable" the actual sending of emails during local development.
- 
- **Log Driver**
  - Instead of sending your emails, the log mail driver will write all email messages to your log files for inspection. Typically, this driver would only be used during local development. For more information on configuring your application per environment, check out the [configuration documentation](#).
- 
- **HELO / Mailtrap / MailHog**
  - Finally, you may use a service like [HELO](#) or [Mailtrap](#) and the smtp driver to send your email messages to a "dummy" mailbox where you may view them in a true email client. This approach has the benefit of allowing you to actually inspect the final emails in Mailtrap's message viewer.
  - If you are using [Laravel Sail](#), you may preview your messages using [MailHog](#). When Sail is running, you may access the MailHog interface at: <http://localhost:8025>.

# Module 7

- Events
- Laravel fires two events during the process of sending mail messages. The MessageSending event is fired prior to a message being sent, while the MessageSent event is fired after a message has been sent. Remember, these events are fired when the mail is being *sent*, not when it is queued. You may register event listeners for this event in your App\Providers\EventServiceProvider service provider:

```
protected $listen = [
 'Illuminate\Mail\Events\MessageSending' => [
 'App\Listeners\LogSendingMessage',
],
 'Illuminate\Mail\Events\MessageSent' => [
 'App\Listeners\LogSentMessage',
],
];
```

## Queues

- Introduction
- While building your web application, you may have some tasks, such as parsing and storing an uploaded CSV file, that take too long to perform during a typical web request. Thankfully, Laravel allows you to easily create queued jobs that may be processed in the background. By moving time intensive tasks to a queue, your application can respond to web requests with blazing speed and provide a better user experience to your customers.

# Module 7

- Laravel queues provide a unified queueing API across a variety of different queue backends, such as [Amazon SQS](#), [Redis](#), or even a relational database.
  - Laravel's queue configuration options are stored in your application's config/queue.php configuration file. In this file, you will find connection configurations for each of the queue drivers that are included with the framework, including the database, [Amazon SQS](#), [Redis](#), and [Beanstalkd](#) drivers, as well as a synchronous driver that will execute jobs immediately (for use during local development). A null queue driver is also included which discards queued jobs.
  - Laravel now offers Horizon, a beautiful dashboard and configuration system for your Redis powered queues. Check out the full [Horizon documentation](#) for more information.
- 
- **[Connections Vs. Queues](#)**
  - Before getting started with Laravel queues, it is important to understand the distinction between "connections" and "queues". In your config/queue.php configuration file, there is a connections configuration array. This option defines the connections to backend queue services such as Amazon SQS, Beanstalk, or Redis. However, any given queue connection may have multiple "queues" which may be thought of as different stacks or piles of queued jobs.
  - Note that each connection configuration example in the queue configuration file contains a queue attribute. This is the default queue that jobs will be dispatched to when they are sent to a given connection. In other words, if you dispatch a job without explicitly defining which queue it should be dispatched to, the job will be placed on the queue that is defined in the queue attribute of the connection configuration:

# Module 7

```
use App\Jobs\ProcessPodcast;

// This job is sent to the default connection's default queue...
ProcessPodcast::dispatch();
// This job is sent to the default connection's "emails" queue...
ProcessPodcast::dispatch()->onQueue('emails');
```

- Some applications may not need to ever push jobs onto multiple queues, instead preferring to have one simple queue. However, pushing jobs to multiple queues can be especially useful for applications that wish to prioritize or segment how jobs are processed, since the Laravel queue worker allows you to specify which queues it should process by priority. For example, if you push jobs to a high queue, you may run a worker that gives them higher processing priority:

```
php artisan queue:work --queue=high,default
```

- [Driver Notes & Prerequisites](#)
- [Database](#)
- In order to use the database queue driver, you will need a database table to hold the jobs. To generate a migration that creates this table, run the `queue:table` Artisan command. Once the migration has been created, you may migrate your database using the `migrate` command:

```
php artisan queue:table
```

```
php artisan migrate
```

# Module 7

- **Redis**
- In order to use the redis queue driver, you should configure a Redis database connection in your config/database.php configuration file.
- Redis Cluster
- If your Redis queue connection uses a Redis Cluster, your queue names must contain a [key hash tag](#). This is required in order to ensure all of the Redis keys for a given queue are placed into the same hash slot:

```
'redis' => [
 'driver' => 'redis',
 'connection' => 'default',
 'queue' => '{default}',
 'retry_after' => 90,
],
```
- **Blocking**
- When using the Redis queue, you may use the block\_for configuration option to specify how long the driver should wait for a job to become available before iterating through the worker loop and re-polling the Redis database.
- Adjusting this value based on your queue load can be more efficient than continually polling the Redis database for new jobs. For instance, you may set the value to 5 to indicate that the driver should block for five seconds while waiting for a job to become available:

# Module 7

```
'redis' => [
 'driver' => 'redis',
 'connection' => 'default',
 'queue' => 'default',
 'retry_after' => 90,
 'block_for' => 5,
],
```

- Setting `block_for` to 0 will cause queue workers to block indefinitely until a job is available. This will also prevent signals such as SIGTERM from being handled until the next job has been processed.

- **Other Driver Prerequisites**

- The following dependencies are needed for the listed queue drivers. These dependencies may be installed via the Composer package manager:

Amazon SQS: `aws/aws-sdk-php ~3.0`

Beanstalkd: `pda/pheanstalk ~4.0`

Redis: `predis/predis ~1.0` or `phpredis` PHP extension

# Module 7

## Creating Jobs

### Generating Job Classes

By default, all of the queueable jobs for your application are stored in the app/Jobs directory. If the app/Jobs directory doesn't exist, it will be created when you run the make:job Artisan command:

```
php artisan make:job ProcessPodcast
```

The generated class will implement the Illuminate\Contracts\Queue\ShouldQueue interface, indicating to Laravel that the job should be pushed onto the queue to run asynchronously.

Job stubs may be customized using [stub publishing](#).

## Class Structure

Job classes are very simple, normally containing only a handle method that is invoked when the job is processed by the queue. To get started, let's take a look at an example job class. In this example, we'll pretend we manage a podcast publishing service and need to process the uploaded podcast files before they are published:

# Module 7

```
<?php
namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
class ProcessPodcast implements ShouldQueue
{
 use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;
 protected $podcast;
 public function __construct(Podcast $podcast)
 {
 $this->podcast = $podcast;
 }
 public function handle(AudioProcessor $processor)
 {
 }
}
```

# Module 7

In this example, note that we were able to pass an [Eloquent model](#) directly into the queued job's constructor. Because of the `SerializesModels` trait that the job is using, Eloquent models and their loaded relationships will be gracefully serialized and unserialized when the job is processing.

If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance and its loaded relationships from the database. This approach to model serialization allows for much smaller job payloads to be sent to your queue driver.

## [handle Method Dependency Injection](#)

The `handle` method is invoked when the job is processed by the queue. Note that we are able to type-hint dependencies on the `handle` method of the job. The Laravel [service container](#) automatically injects these dependencies.

If you would like to take total control over how the container injects dependencies into the `handle` method, you may use the container's `bindMethod` method. The `bindMethod` method accepts a callback which receives the job and the container. Within the callback, you are free to invoke the `handle` method however you wish. Typically, you should call this method from the `boot` method of your

`App\Providers\AppServiceProvider` [service provider](#):

# Module 7

```
use App\Jobs\ProcessPodcast;
use App\Services\AudioProcessor;

$this->app->bindMethod([ProcessPodcast::class, 'handle'], function ($job, $app) {
 return $job->handle($app->make(AudioProcessor::class));
});
```

Binary data, such as raw image contents, should be passed through the `base64_encode` function before being passed to a queued job. Otherwise, the job may not properly serialize to JSON when being placed on the queue.

## Handling Relationships

Because loaded relationships also get serialized, the serialized job string can sometimes become quite large. To prevent relations from being serialized, you can call the `withoutRelations` method on the model when setting a property value. This method will return an instance of the model without its loaded relationships:

```
public function __construct(Podcast $podcast)
{
 $this->podcast = $podcast->withoutRelations();
}
```

# Module 7

## Unique Jobs

Unique jobs require a cache driver that supports [locks](#). Currently, the memcached, redis, dynamodb, database, file, and array cache drivers support atomic locks. In addition, unique job constraints do not apply to jobs within batches.

Sometimes, you may want to ensure that only one instance of a specific job is on the queue at any point in time. You may do so by implementing the `ShouldBeUnique` interface on your job class. This interface does not require you to define any additional methods on your class:

```
<?php
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
 ...
}
```

In the example above, the `UpdateSearchIndex` job is unique. So, the job will not be dispatched if another instance of the job is already on the queue and has not finished processing. In certain cases, you may want to define a specific "key" that makes the job unique or you may want to specify a timeout beyond which the job no longer stays unique. To accomplish this, you may define `uniqueId` and `uniqueFor` properties or methods on your job class:

# Module 7

```
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;
class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
 public $product;
 public $uniqueFor = 3600;
 public function uniqueId()
 {
 return $this->product->id;
 }
}
```

In the example above, the `UpdateSearchIndex` job is unique by a product ID. So, any new dispatches of the job with the same product ID will be ignored until the existing job has completed processing. In addition, if the existing job is not processed within one hour, the unique lock will be released and another job with the same unique key can be dispatched to the queue.

# Module 7

- **Keeping Jobs Unique Until Processing Begins**
- By default, unique jobs are "unlocked" after a job completes processing or fails all of its retry attempts. However, there may be situations where you would like your job to unlock immediately before it is processed. To accomplish this, your job should implement the `ShouldBeUniqueUntilProcessing` contract instead of the `ShouldBeUnique` contract:

```
<?php
use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUniqueUntilProcessing
{
 // ...
}
```
- **Unique Job Locks**
- Behind the scenes, when a `ShouldBeUnique` job is dispatched, Laravel attempts to acquire a [lock](#) with the `uniqueId` key. If the lock is not acquired, the job is not dispatched. This lock is released when the job completes processing or fails all of its retry attempts. By default, Laravel will use the default cache driver to obtain this lock. However, if you wish to use another driver for acquiring the lock, you may define a `uniqueVia` method that returns the cache driver that should be used:

# Module 7

```
use Illuminate\Support\Facades\Cache;
```

```
class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
public function uniqueVia()
{
 return Cache::driver('redis');
}
```

- If you only need to limit the concurrent processing of a job, use the [WithoutOverlapping](#) job middleware instead
- [Job Middleware](#)
- Job middleware allow you to wrap custom logic around the execution of queued jobs, reducing boilerplate in the jobs themselves. For example, consider the following handle method which leverages Laravel's Redis rate limiting features to allow only one job to process every five seconds:

# Module 7

```
use Illuminate\Support\Facades\Redis;
public function handle()
{
 Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {
 info('Lock obtained...');

 // Handle job...
 }, function () {
 // Could not obtain lock...

 return $this->release(5);
 });
}
```

- While this code is valid, the implementation of the handle method becomes noisy since it is cluttered with Redis rate limiting logic. In addition, this rate limiting logic must be duplicated for any other jobs that we want to rate limit.
- Instead of rate limiting in the handle method, we could define a job middleware that handles rate limiting. Laravel does not have a default location for job middleware, so you are welcome to place job middleware anywhere in your application. In this example, we will place the middleware in an app/Jobs/Middleware directory:

# Module 7

```
<?php

namespace App\Jobs\Middleware;

use Illuminate\Support\Facades\Redis;

class RateLimited
{
 public function handle($job, $next)
 {
 Redis::throttle('key')
 ->block(0)->allow(1)->every(5)
 ->then(function () use ($job, $next) {
 $next($job);
 }, function () use ($job) {
 // Could not obtain lock...

 $job->release(5);
 });
 }
}
```

# Module 7

As you can see, like [route middleware](#), job middleware receive the job being processed and a callback that should be invoked to continue processing the job.

After creating job middleware, they may be attached to a job by returning them from the job's middleware method. This method does not exist on jobs scaffolded by the make:job Artisan command, so you will need to manually add it to your job class:

```
use App\Jobs\Middleware\RateLimited;
public function middleware()
{
 return [new RateLimited];
}
```

Job middleware can also be assigned to queueable event listeners, mailables, and notifications.

## [Rate Limiting](#)

Although we just demonstrated how to write your own rate limiting job middleware, Laravel actually includes a rate limiting middleware that you may utilize to rate limit jobs. Like [route rate limiters](#), job rate limiters are defined using the RateLimiter facade's for method.

For example, you may wish to allow users to backup their data once per hour while imposing no such limit on premium customers. To accomplish this, you may define a RateLimiter in the boot method of your AppServiceProvider:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;
```

# Module 7

```
public function boot()
{
 RateLimiter::for('backups', function ($job) {
 return $job->user->vipCustomer()
 ? Limit::none()
 : Limit::perHour(1)->by($job->user->id);
 });
}
```

In the example above, we defined an hourly rate limit; however, you may easily define a rate limit based on minutes using the perMinute method. In addition, you may pass any value you wish to the by method of the rate limit; however, this value is most often used to segment rate limits by customer:

```
return Limit::perMinute(50)->by($job->user->id);
```

Once you have defined your rate limit, you may attach the rate limiter to your backup job using the Illuminate\Queue\Middleware\RateLimited middleware. Each time the job exceeds the rate limit, this middleware will release the job back to the queue with an appropriate delay based on the rate limit duration.

```
use Illuminate\Queue\Middleware\RateLimited;
public function middleware()
{
 return [new RateLimited('backups')];
}
```

# Module 7

Releasing a rate limited job back onto the queue will still increment the job's total number of attempts. You may wish to tune your tries and maxExceptions properties on your job class accordingly. Or, you may wish to use the [retryUntil method](#) to define the amount of time until the job should no longer be attempted.

If you are using Redis, you may use the Illuminate\Queue\Middleware\RateLimitedWithRedis middleware, which is fine-tuned for Redis and more efficient than the basic rate limiting middleware.

## Preventing Job Overlaps

Laravel includes an Illuminate\Queue\Middleware\WithoutOverlapping middleware that allows you to prevent job overlaps based on an arbitrary key. This can be helpful when a queued job is modifying a resource that should only be modified by one job at a time.

For example, let's imagine you have a queued job that updates a user's credit score and you want to prevent credit score update job overlaps for the same user ID. To accomplish this, you can return the WithoutOverlapping middleware from your job's middleware method:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;
public function middleware()
{
 return [new WithoutOverlapping($this->user->id)];
}
```

# Module 7

Any overlapping jobs will be released back to the queue. You may also specify the number of seconds that must elapse before the released job will be attempted again:

```
public function middleware()
{
 return [(new WithoutOverlapping($this->order->id))->releaseAfter(60)];
}
```

If you wish to immediately delete any overlapping jobs so that they will not be retried, you may use the `dontRelease` method:

```
public function middleware()
{
 return [(new WithoutOverlapping($this->order->id))->dontRelease()];
}
```

The `WithoutOverlapping` middleware is powered by Laravel's atomic lock feature. Sometimes, your job may unexpectedly fail or timeout in such a way that the lock is not released. Therefore, you may explicitly define a lock expiration time using the `expireAfter` method. For example, the example below will instruct Laravel to release the `WithoutOverlapping` lock three minutes after the job has started processing:

```
public function middleware()
{
 return [(new WithoutOverlapping($this->order->id))->expireAfter(180)];
}
```

# Module 7

The WithoutOverlapping middleware requires a cache driver that supports [locks](#). Currently, the memcached, redis, dynamodb, database, file, and array cache drivers support atomic locks.

## Throttling Exceptions

Laravel includes a `Illuminate\Queue\Middleware\ThrottlesExceptions` middleware that allows you to throttle exceptions. Once the job throws a given number of exceptions, all further attempts to execute the job are delayed until a specified time interval lapses. This middleware is particularly useful for jobs that interact with third-party services that are unstable.

For example, let's imagine a queued job that interacts with a third-party API that begins throwing exceptions. To throttle exceptions, you can return the `ThrottlesExceptions` middleware from your job's middleware method. Typically, this middleware should be paired with a job that implements [time based attempts](#):

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;
public function middleware()
{
 return [new ThrottlesExceptions(10, 5)];
}
public function retryUntil()
{
 return now()->addMinutes(5);
}
```

# Module 7

- The first constructor argument accepted by the middleware is the number of exceptions the job can throw before being throttled, while the second constructor argument is the number of minutes that should elapse before the job is attempted again once it has been throttled. In the code example above, if the job throws 10 exceptions within 5 minutes, we will wait 5 minutes before attempting the job again.
- When a job throws an exception but the exception threshold has not yet been reached, the job will typically be retried immediately. However, you may specify the number of minutes such a job should be delayed by calling the backoff method when attaching the middleware to the job:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;
public function middleware()
{
 return [(new ThrottlesExceptions(10, 5))->backoff(5)];
}
```

- Internally, this middleware uses Laravel's cache system to implement rate limiting, and the job's class name is utilized as the cache "key". You may override this key by calling the by method when attaching the middleware to your job. This may be useful if you have multiple jobs interacting with the same third-party service and you would like them to share a common throttling "bucket":

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;
```

# Module 7

```
public function middleware()
{
 return [(new ThrottlesExceptions(10, 10))->by('key')];
}
```

If you are using Redis, you may use the `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis` middleware, which is fine-tuned for Redis and more efficient than the basic exception throttling middleware.

## Dispatching Jobs

Once you have written your job class, you may dispatch it using the `dispatch` method on the job itself. The arguments passed to the `dispatch` method will be given to the job's constructor:

```
<?php namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;
class PodcastController extends Controller{
```

# Module 7

```
public function store(Request $request)
{
 $podcast = Podcast::create(...);

 // ...

 ProcessPodcast::dispatch($podcast);
}
```

If you would like to conditionally dispatch a job, you may use the `dispatchIf` and `dispatchUnless` methods:

```
ProcessPodcast::dispatchIf($accountActive, $podcast);
```

```
ProcessPodcast::dispatchUnless($accountSuspended, $podcast);
```

## Delayed Dispatching

If you would like to specify that a job should not be immediately available for processing by a queue worker, you may use the `delay` method when dispatching the job. For example, let's specify that a job should not be available for processing until 10 minutes after it has been dispatched:

# Module 7

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;
class PodcastController extends Controller
{
 public function store(Request $request)
 {
 $podcast = Podcast::create(...);
 ProcessPodcast::dispatch($podcast)
 ->delay(now()->addMinutes(10));
 }
}
```

- The Amazon SQS queue service has a maximum delay time of 15 minutes.

# Module 7

## Dispatching After The Response Is Sent To Browser

Alternatively, the `dispatchAfterResponse` method delays dispatching a job until after the HTTP response is sent to the user's browser. This will still allow the user to begin using the application even though a queued job is still executing. This should typically only be used for jobs that take about a second, such as sending an email. Since they are processed within the current HTTP request, jobs dispatched in this fashion do not require a queue worker to be running in order for them to be processed:

```
use App\Jobs\SendNotification;
```

```
SendNotification::dispatchAfterResponse();
```

You may also dispatch a closure and chain the `afterResponse` method onto the `dispatch` helper to execute a closure after the HTTP response has been sent to the browser:

```
use App\Mail\WelcomeMessage;
use Illuminate\Support\Facades\Mail;
dispatch(function () {
 Mail::to('taylor@example.com')->send(new WelcomeMessage);
})->afterResponse();
```

# Module 7

## Synchronous Dispatching

If you would like to dispatch a job immediately (synchronously), you may use the `dispatchSync` method. When using this method, the job will not be queued and will be executed immediately within the current process:

```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
 public function store(Request $request)
 {
 $podcast = Podcast::create(...);

 // Create podcast...

 ProcessPodcast::dispatchSync($podcast);
 }
}
```

# Module 7

## Jobs & Database Transactions

While it is perfectly fine to dispatch jobs within database transactions, you should take special care to ensure that your job will actually be able to execute successfully. When dispatching a job within a transaction, it is possible that the job will be processed by a worker before the transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database.

Thankfully, Laravel provides several methods of working around this problem. First, you may set the `after_commit` connection option in your queue connection's configuration array:

```
'redis' => [
 'driver' => 'redis',
 // ...
 'after_commit' => true,
],
```

When the `after_commit` option is true, you may dispatch jobs within database transactions; however, Laravel will wait until all open database transactions have been committed before actually dispatching the job. Of course, if no database transactions are currently open, the job will be dispatched immediately.

If a transaction is rolled back due to an exception that occurs during the transaction, the dispatched jobs that were dispatched during that transaction will be discarded.

# Module 7

Setting the `after_commit` configuration option to true will also cause any queued event listeners, mailables, notifications, and broadcast events to be dispatched after all open database transactions have been committed.

## Specifying Commit Dispatch Behavior Inline

If you do not set the `after_commit` queue connection configuration option to true, you may still indicate that a specific job should be dispatched after all open database transactions have been committed. To accomplish this, you may chain the `afterCommit` method onto your dispatch operation:

```
use App\Jobs\ProcessPodcast;
ProcessPodcast::dispatch($podcast)->afterCommit();
```

Likewise, if the `after_commit` configuration option is set to true, you may indicate that a specific job should be dispatched immediately without waiting for any open database transactions to commit:

```
ProcessPodcast::dispatch($podcast)->beforeCommit();
```

## Job Chaining

Job chaining allows you to specify a list of queued jobs that should be run in sequence after the primary job has executed successfully. If one job in the sequence fails, the rest of the jobs will not be run. To execute a queued job chain, you may use the `chain` method provided by the Bus facade. Laravel's command bus is a lower level component that queued job dispatching is built on top of:

# Module 7

```
use App\Jobs\OptimizePodcast;
use App\Jobs\ProcessPodcast;
use App\Jobs\ReleasePodcast;
use Illuminate\Support\Facades\Bus;
```

```
Bus::chain([
 new ProcessPodcast,
 new OptimizePodcast,
 new ReleasePodcast,
])->dispatch();
```

- In addition to chaining job class instances, you may also chain closures:

```
Bus::chain([
 new ProcessPodcast,
 new OptimizePodcast,
 function () {
 Podcast::update(...);
 },
])->dispatch();
```

- Deleting jobs using the `$this->delete()` method within the job will not prevent chained jobs from being processed. The chain will only stop executing if a job in the chain fails.

# Module 7

## Chain Connection & Queue

If you would like to specify the connection and queue that should be used for the chained jobs, you may use the `onConnection` and `onQueue` methods. These methods specify the queue connection and queue name that should be used unless the queued job is explicitly assigned a different connection / queue:

```
Bus::chain([
 new ProcessPodcast,
 new OptimizePodcast,
 new ReleasePodcast,
])->onConnection('redis')->onQueue('podcasts')->dispatch();
```

## Chain Failures

When chaining jobs, you may use the `catch` method to specify a closure that should be invoked if a job within the chain fails. The given callback will receive the `Throwable` instance that caused the job failure:

```
use Illuminate\Support\Facades\Bus;
use Throwable;
Bus::chain([
 new ProcessPodcast,
 new OptimizePodcast,
 new ReleasePodcast,
])->catch(function (Throwable $e) {
})->dispatch();
```

# Module 7

## Customizing The Queue & Connection

### Dispatching To A Particular Queue

By pushing jobs to different queues, you may "categorize" your queued jobs and even prioritize how many workers you assign to various queues. Keep in mind, this does not push jobs to different queue "connections" as defined by your queue configuration file, but only to specific queues within a single connection. To specify the queue, use the `onQueue` method when dispatching the job:

```
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
 public function store(Request $request)
 {
 $podcast = Podcast::create(...);
 ProcessPodcast::dispatch($podcast)->onQueue('processing');
 }
}
```

# Module 7

- Alternatively, you may specify the job's queue by calling the `onQueue` method within the job's constructor:

```
<?php
```

```
namespace App\Jobs;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
 use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;
 public function __construct()
 {
 $this->onQueue('processing');
 }
}
```

# Module 7

## Dispatching To A Particular Connection

If your application interacts with multiple queue connections, you may specify which connection to push a job to using the `onConnection` method:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;
class PodcastController extends Controller
{
 public function store(Request $request)
 {
 $podcast = Podcast::create(...);
 ProcessPodcast::dispatch($podcast)->onConnection('sq');
 }
}
```

# Module 7

You may chain the `onConnection` and `onQueue` methods together to specify the connection and the queue for a job:

```
ProcessPodcast::dispatch($podcast)
 ->onConnection('sqS')
 ->onQueue('processing');
```

Alternatively, you may specify the job's connection by calling the `onConnection` method within the job's constructor

```
<?php
```

```
namespace App\Jobs;
```

```
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
```

```
class ProcessPodcast implements ShouldQueue
{
 use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;
```

# Module 7

```
public function __construct()
{
 $this->onConnection('sq');
}
}
```

## Specifying Max Job Attempts / Timeout Values Max Attempts

If one of your queued jobs is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a job may be attempted.

One approach to specifying the maximum number of times a job may be attempted is via the --tries switch on the Artisan command line. This will apply to all jobs processed by the worker unless the job being processed specifies a more specific number of times it may be attempted:

```
php artisan queue:work --tries=3
```

If a job exceeds its maximum number of attempts, it will be considered a "failed" job. For more information on handling failed jobs, consult the [failed job documentation](#).

You may take a more granular approach by defining the maximum number of times a job may be attempted on the job class itself. If the maximum number of attempts is specified on the job, it will take precedence over the --tries value provided on the command line:

# Module 7

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
 public $tries = 5;
}
```

## Time Based Attempts

As an alternative to defining how many times a job may be attempted before it fails, you may define a time at which the job should no longer be attempted. This allows a job to be attempted any number of times within a given time frame. To define the time at which a job should no longer be attempted, add a `retryUntil` method to your job class. This method should return a `DateTime` instance:

```
public function retryUntil()
{
 return now()->addMinutes(10);
}
```

You may also define a `tries` property or `retryUntil` method on your [queued event listeners](#).

# Module 7

## Max Exceptions

Sometimes you may wish to specify that a job may be attempted many times, but should fail if the retries are triggered by a given number of unhandled exceptions (as opposed to being released by the release method directly). To accomplish this, you may define a `maxExceptions` property on your job class:

```
<?php
namespace App\Jobs;

use Illuminate\Support\Facades\Redis;

class ProcessPodcast implements ShouldQueue
{
 public $tries = 25;
 public $maxExceptions = 3;
 public function handle()
 {
 Redis::throttle('key')->allow(10)->every(60)->then(function () {
 }, function () {
 return $this->release(10);
 });
 }
}
```

# Module 7

In this example, the job is released for ten seconds if the application is unable to obtain a Redis lock and will continue to be retried up to 25 times. However, the job will fail if three unhandled exceptions are thrown by the job.

## Timeout

The pcntl PHP extension must be installed in order to specify job timeouts.

Often, you know roughly how long you expect your queued jobs to take. For this reason, Laravel allows you to specify a "timeout" value. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a [process manager configured on your server](#).

The maximum number of seconds that jobs can run may be specified using the --timeout switch on the Artisan command line:

```
php artisan queue:work --timeout=30
```

If the job exceeds its maximum attempts by continually timing out, it will be marked as failed.

You may also define the maximum number of seconds a job should be allowed to run on the job class itself. If the timeout is specified on the job, it will take precedence over any timeout specified on the command line:

# Module 7

```
<?php
namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
 public $timeout = 120;
}
```

Sometimes, IO blocking processes such as sockets or outgoing HTTP connections may not respect your specified timeout. Therefore, when using these features, you should always attempt to specify a timeout using their APIs as well. For example, when using Guzzle, you should always specify a connection and request timeout value.

## Failing On Timeout

If you would like to indicate that a job should be marked as [failed](#) on timeout, you may define the `$failOnTimeout` property on the job class:

```
public $failOnTimeout = true;
```

# Module 7

## Error Handling

If an exception is thrown while the job is being processed, the job will automatically be released back onto the queue so it may be attempted again. The job will continue to be released until it has been attempted the maximum number of times allowed by your application. The maximum number of attempts is defined by the --tries switch used on the queue:work Artisan command. Alternatively, the maximum number of attempts may be defined on the job class itself. More information on running the queue worker [can be found below](#).

## Manually Releasing A Job

Sometimes you may wish to manually release a job back onto the queue so that it can be attempted again at a later time. You may accomplish this by calling the release method:

```
public function handle()
{
 // ...

 $this->release();
}
```

By default, the release method will release the job back onto the queue for immediate processing. However, by passing an integer to the release method you may instruct the queue to not make the job available for processing until a given number of seconds has elapsed:

```
$this->release(10);
```

# Module 7

## Manually Failing A Job

Occasionally you may need to manually mark a job as "failed". To do so, you may call the fail method:

```
public function handle()
{
 // ...

 $this->fail();
}
```

If you would like to mark your job as failed because of an exception that you have caught, you may pass the exception to the fail method:

```
$this->fail($exception);
```

For more information on failed jobs, check out the [documentation on dealing with job failures](#).

## Job Batching

Laravel's job batching feature allows you to easily execute a batch of jobs and then perform some action when the batch of jobs has completed executing. Before getting started, you should create a database migration to build a table to contain meta information about your job batches, such as their completion percentage. This migration may be generated using the `queue:batches-table` Artisan command:

# Module 7

```
php artisan queue:batches-table
```

```
php artisan migrate
```

- **Defining Batchable Jobs**
- To define a batchable job, you should [create a queueable job](#) as normal; however, you should add the Illuminate\Bus\Batchable trait to the job class. This trait provides access to a batch method which may be used to retrieve the current batch that the job is executing within:

```
<?php

namespace App\Jobs;

use Illuminate\Bus\Batchable;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ImportCsv implements ShouldQueue
{
```

# Module 7

```
use Batchable, Dispatchable, InteractsWithQueue, Queueable, SerializesModels;
public function handle()
{
 if ($this->batch()->cancelled()) {
 // Determine if the batch has been cancelled...

 return;
 }

 // Import a portion of the CSV file...
}
}
}
```

- **Dispatching Batches**
- To dispatch a batch of jobs, you should use the batch method of the Bus facade. Of course, batching is primarily useful when combined with completion callbacks. So, you may use the then, catch, and finally methods to define completion callbacks for the batch. Each of these callbacks will receive an Illuminate\Bus\Batch instance when they are invoked. In this example, we will imagine we are queueing a batch of jobs that each process a given number of rows from a CSV file:

# Module 7

```
use App\Jobs\ImportCsv;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;
use Throwable;

$batch = Bus::batch([
 new ImportCsv(1, 100),
 new ImportCsv(101, 200),
 new ImportCsv(201, 300),
 new ImportCsv(301, 400),
 new ImportCsv(401, 500),
])->then(function (Batch $batch) {
 // All jobs completed successfully...
})->catch(function (Batch $batch, Throwable $e) {
})->finally(function (Batch $batch) {
 // The batch has finished executing...
})->dispatch();

return $batch->id;
```

- The batch's ID, which may be accessed via the `$batch->id` property, may be used to [query the Laravel command bus](#) for information about the batch after it has been dispatched.

# Module 7

- **Naming Batches**
- Some tools such as Laravel Horizon and Laravel Telescope may provide more user-friendly debug information for batches if batches are named. To assign an arbitrary name to a batch, you may call the name method while defining the batch:

```
$batch = Bus::batch([
 // ...
])->then(function (Batch $batch) {
 // All jobs completed successfully...
})->name('Import CSV')->dispatch();
```

- **Batch Connection & Queue**

- If you would like to specify the connection and queue that should be used for the batched jobs, you may use the onConnection and onQueue methods. All batched jobs must execute within the same connection and queue:

```
$batch = Bus::batch([
])->then(function (Batch $batch) {
 // All jobs completed successfully...
})->onConnection('redis')->onQueue('imports')->dispatch();
```

# Module 7

- [\*\*Chains Within Batches\*\*](#)
- You may define a set of [\*\*chained jobs\*\*](#) within a batch by placing the chained jobs within an array. For example, we may execute two job chains in parallel and execute a callback when both job chains have finished processing:

```
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;
```

```
Bus::batch([
 [
 new ReleasePodcast(1),
 new SendPodcastReleaseNotification(1),
],
 [
 new ReleasePodcast(2),
 new SendPodcastReleaseNotification(2),
],
])->then(function (Batch $batch) {
 // ...
})->dispatch();
```

# Module 7

- **Adding Jobs To Batches**
- Sometimes it may be useful to add additional jobs to a batch from within a batched job. This pattern can be useful when you need to batch thousands of jobs which may take too long to dispatch during a web request. So, instead, you may wish to dispatch an initial batch of "loader" jobs that hydrate the batch with even more jobs:

```
$batch = Bus::batch([
 new LoadImportBatch,
 new LoadImportBatch,
 new LoadImportBatch,
])->then(function (Batch $batch) {
 // All jobs completed successfully...
})->name('Import Contacts')->dispatch();
```
- In this example, we will use the LoadImportBatch job to hydrate the batch with additional jobs. To accomplish this, we may use the add method on the batch instance that may be accessed via the job's batch method:

```
use App\Jobs\ImportContacts;
use Illuminate\Support\Collection;
public function handle()
{
 if ($this->batch()->cancelled()) {
 return;
 }
```

# Module 7

```
$this->batch()->add(Collection::times(1000, function () {
 return new ImportContacts;
});
})
```

- You may only add jobs to a batch from within a job that belongs to the same batch.
- [\*\*Inspecting Batches\*\*](#)
- The Illuminate\Bus\Batch instance that is provided to batch completion callbacks has a variety of properties and methods to assist you in interacting with and inspecting a given batch of jobs:

```
// The UUID of the batch...
$batch->id;

// The name of the batch (if applicable)...
$batch->name;

// The number of jobs assigned to the batch...
$batch->totalJobs;

// The number of jobs that have not been processed by the queue...
$batch->pendingJobs;

// The number of jobs that have failed...
$batch->failedJobs;
```

# Module 7

```
// The number of jobs that have been processed thus far...
$batch->processedJobs();
```

```
// The completion percentage of the batch (0-100)...
$batch->progress();
```

```
// Indicates if the batch has finished executing...
$batch->finished();
```

```
// Cancel the execution of the batch...
$batch->cancel();
```

```
// Indicates if the batch has been cancelled...
$batch->cancelled();
```

## Returning Batches From Routes

All Illuminate\Bus\Batch instances are JSON serializable, meaning you can return them directly from one of your application's routes to retrieve a JSON payload containing information about the batch, including its completion progress. This makes it convenient to display information about the batch's completion progress in your application's UI.

# Module 7

To retrieve a batch by its ID, you may use the Bus facade's `findBatch` method:

```
use Illuminate\Support\Facades\Bus;
use Illuminate\Support\Facades\Route;

Route::get('/batch/{batchId}', function (string $batchId) {
 return Bus::findBatch($batchId);
});
```

## Cancelling Batches

Sometimes you may need to cancel a given batch's execution. This can be accomplished by calling the `cancel` method on the `Illuminate\Bus\Batch` instance:

```
public function handle()
{
 if ($this->user->exceedsImportLimit()) {
 return $this->batch()->cancel();
 }

 if ($this->batch()->cancelled()) {
 return;
 }
}
```

# Module 7

As you may have noticed in previous examples, batched jobs should typically check to see if the batch has been cancelled at the beginning of their handle method:

```
public function handle()
{
 if ($this->batch()->cancelled()) {
 return;
 }

 // Continue processing...
}
```

## Batch Failures

When a batched job fails, the catch callback (if assigned) will be invoked. This callback is only invoked for the first job that fails within the batch.

## Allowing Failures

When a job within a batch fails, Laravel will automatically mark the batch as "cancelled". If you wish, you may disable this behavior so that a job failure does not automatically mark the batch as cancelled. This may be accomplished by calling the allowFailures method while dispatching the batch:

# Module 7

```
$batch = Bus::batch([
 ...
])->then(function (Batch $batch) {
 // All jobs completed successfully...
})->allowFailures()->dispatch();
```

## Retrying Failed Batch Jobs

For convenience, Laravel provides a `queue:retry-batch` Artisan command that allows you to easily retry all of the failed jobs for a given batch. The `queue:retry-batch` command accepts the UUID of the batch whose failed jobs should be retried:

```
php artisan queue:retry-batch 32dbc76c-4f82-4749-b610-a639fe0099b5
```

## Pruning Batches

Without pruning, the `job_batches` table can accumulate records very quickly. To mitigate this, you should [schedule](#) the `queue:prune-batches` Artisan command to run daily:

```
$schedule->command('queue:prune-batches')->daily();
```

By default, all finished batches that are more than 24 hours old will be pruned. You may use the `hours` option when calling the command to determine how long to retain batch data. For example, the following command will delete all batches that finished over 48 hours ago:

# Module 7

```
$schedule->command('queue:prune-batches --hours=48')->daily();
```

Sometimes, your `jobs_batches` table may accumulate batch records for batches that never completed successfully, such as batches where a job failed and that job was never retried successfully. You may instruct the `queue:prune-batches` command to prune these unfinished batch records using the `unfinished` option:

```
$schedule->command('queue:prune-batches --hours=48 --unfinished=72')->daily();
```

## Queueing Closures

Instead of dispatching a job class to the queue, you may also dispatch a closure. This is great for quick, simple tasks that need to be executed outside of the current request cycle. When dispatching closures to the queue, the closure's code content is cryptographically signed so that it can not be modified in transit:

```
$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {
 $podcast->publish();
});
```

Using the `catch` method, you may provide a closure that should be executed if the queued closure fails to complete successfully after exhausting all of your queue's [configured retry attempts](#):

# Module 7

```
use Throwable;

dispatch(function () use ($podcast) {
 $podcast->publish();
})->catch(function (Throwable $e) {
 // This job has failed...
});
```

## Running The Queue Worker

### The queue:work Command

Laravel includes an Artisan command that will start a queue worker and process new jobs as they are pushed onto the queue. You may run the worker using the `queue:work` Artisan command. Note that once the `queue:work` command has started, it will continue to run until it is manually stopped or you close your terminal:

```
php artisan queue:work
```

To keep the `queue:work` process running permanently in the background, you should use a process monitor such as [Supervisor](#) to ensure that the queue worker does not stop running.

Remember, queue workers, are long-lived processes and store the booted application state in memory. As a result, they will not notice changes in your code base after they have been started. So, during your deployment process, be sure to [restart your queue workers](#). In addition, remember that any static state created or modified by your application will not be automatically reset between jobs.

# Module 7

Alternatively, you may run the `queue:listen` command. When using the `queue:listen` command, you don't have to manually restart the worker when you want to reload your updated code or reset the application state; however, this command is significantly less efficient than the `queue:work` command:

```
php artisan queue:listen
```

## [Running Multiple Queue Workers](#)

To assign multiple workers to a queue and process jobs concurrently, you should simply start multiple `queue:work` processes. This can either be done locally via multiple tabs in your terminal or in production using your process manager's configuration settings. [When using Supervisor](#), you may use the `numprocs` configuration value.

## [Specifying The Connection & Queue](#)

You may also specify which queue connection the worker should utilize. The connection name passed to the `work` command should correspond to one of the connections defined in your `config/queue.php` configuration file:

```
php artisan queue:work redis
```

You may customize your queue worker even further by only processing particular queues for a given connection. For example, if all of your emails are processed in an `emails` queue on your `redis` queue connection, you may issue the following command to start a worker that only processes that queue:

# Module 7

```
php artisan queue:work redis --queue=emails
```

- **Processing A Specified Number Of Jobs**
  - The --once option may be used to instruct the worker to only process a single job from the queue:  
`php artisan queue:work --once`
  - The --max-jobs option may be used to instruct the worker to process the given number of jobs and then exit. This option may be useful when combined with [Supervisor](#) so that your workers are automatically restarted after processing a given number of jobs, releasing any memory they may have accumulated:  
`php artisan queue:work --max-jobs=1000`
- **Processing All Queued Jobs & Then Exiting**
  - The --stop-when-empty option may be used to instruct the worker to process all jobs and then exit gracefully. This option can be useful when processing Laravel queues within a Docker container if you wish to shutdown the container after the queue is empty:  
`php artisan queue:work --stop-when-empty`

# Module 7

## Queue Priorities

Sometimes you may wish to prioritize how your queues are processed. For example, in your config/queue.php configuration file, you may set the default queue for your redis connection to low. However, occasionally you may wish to push a job to a high priority queue like so:

```
dispatch((new Job)->onQueue('high'));
```

To start a worker that verifies that all of the high queue jobs are processed before continuing to any jobs on the low queue, pass a comma-delimited list of queue names to the work command:

```
php artisan queue:work --queue=high,low
```

## Queue Workers & Deployment

Since queue workers are long-lived processes, they will not notice changes to your code without being restarted. So, the simplest way to deploy an application using queue workers is to restart the workers during your deployment process. You may gracefully restart all of the workers by issuing the queue:restart command:

```
php artisan queue:restart
```

This command will instruct all queue workers to gracefully exit after they finish processing their current job so that no existing jobs are lost. Since the queue workers will exit when the queue:restart command is executed, you should be running a process manager such as [Supervisor](#) to automatically restart the queue workers.

# Module 7

The queue uses the [cache](#) to store restart signals, so you should verify that a cache driver is properly configured for your application before using this feature.

## Job Expirations & Timeouts

### Job Expiration

In your config/queue.php configuration file, each queue connection defines a retry\_after option. This option specifies how many seconds the queue connection should wait before retrying a job that is being processed. For example, if the value of retry\_after is set to 90, the job will be released back onto the queue if it has been processing for 90 seconds without being released or deleted. Typically, you should set the retry\_after value to the maximum number of seconds your jobs should reasonably take to complete processing.

The only queue connection which does not contain a retry\_after value is Amazon SQS. SQS will retry the job based on the [Default Visibility Timeout](#) which is managed within the AWS console.

### Worker Timeouts

The queue:work Artisan command exposes a --timeout option. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a [process manager configured on your server](#):

```
php artisan queue:work --timeout=60
```

The retry\_after configuration option and the --timeout CLI option are different, but work together to ensure that jobs are not lost and that jobs are only successfully processed once.

# Module 7

The --timeout value should always be at least several seconds shorter than your retry\_after configuration value. This will ensure that a worker processing a frozen job is always terminated before the job is retried. If your --timeout option is longer than your retry\_after configuration value, your jobs may be processed twice.

## Supervisor Configuration

In production, you need a way to keep your queue:work processes running. A queue:work process may stop running for a variety of reasons, such as an exceeded worker timeout or the execution of the queue:restart command.

For this reason, you need to configure a process monitor that can detect when your queue:work processes exit and automatically restart them. In addition, process monitors can allow you to specify how many queue:work processes you would like to run concurrently. Supervisor is a process monitor commonly used in Linux environments and we will discuss how to configure it in the following documentation.

## Installing Supervisor

Supervisor is a process monitor for the Linux operating system, and will automatically restart your queue:work processes if they fail. To install Supervisor on Ubuntu, you may use the following command:

```
sudo apt-get install supervisor
```

# Module 7

If configuring and managing Supervisor yourself sounds overwhelming, consider using [Laravel Forge](#), which will automatically install and configure Supervisor for your production Laravel projects.

## Configuring Supervisor

Supervisor configuration files are typically stored in the /etc/supervisor/conf.d directory. Within this directory, you may create any number of configuration files that instruct supervisor how your processes should be monitored. For example, let's create a laravel-worker.conf file that starts and monitors queue:work processes:

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3 --max-time=3600
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
stopwaitsecs=3600
```

# Module 7

In this example, the numprocs directive will instruct Supervisor to run eight queue:work processes and monitor all of them, automatically restarting them if they fail. You should change the command directive of the configuration to reflect your desired queue connection and worker options.

You should ensure that the value of stopwaitsecs is greater than the number of seconds consumed by your longest running job. Otherwise, Supervisor may kill the job before it is finished processing.

## Starting Supervisor

Once the configuration file has been created, you may update the Supervisor configuration and start the processes using the following commands:

```
sudo supervisorctl reread
```

```
sudo supervisorctl update
```

```
sudo supervisorctl start laravel-worker:*
```

For more information on Supervisor, consult the [Supervisor documentation](#).

# Module 7

## Dealing With Failed Jobs

Sometimes your queued jobs will fail. Don't worry, things don't always go as planned! Laravel includes a convenient way to [specify the maximum number of times a job should be attempted](#). After a job has exceeded this number of attempts, it will be inserted into the failed\_jobs database table. Of course, we will need to create that table if it does not already exist. To create a migration for the failed\_jobs table, you may use the queue:failed-table command:

```
php artisan queue:failed-table
```

```
php artisan migrate
```

When running a [queue worker](#) process, you may specify the maximum number of times a job should be attempted using the --tries switch on the queue:work command. If you do not specify a value for the --tries option, jobs will only be attempted once or as many times as specified by the job class' \$tries property:

```
php artisan queue:work redis --tries=3
```

Using the --backoff option, you may specify how many seconds Laravel should wait before retrying a job that has encountered an exception. By default, a job is immediately released back onto the queue so that it may be attempted again:

```
php artisan queue:work redis --tries=3 --backoff=3
```

# Module 7

If you would like to configure how many seconds Laravel should wait before retrying a job that has encountered an exception on a per-job basis, you may do so by defining a `backoff` property on your job class:

```
public $backoff = 3;
```

If you require more complex logic for determining the job's backoff time, you may define a `backoff` method on your job class:

```
public function backoff()
{
 return 3;
}
```

You may easily configure "exponential" backoffs by returning an array of backoff values from the `backoff` method. In this example, the retry delay will be 1 second for the first retry, 5 seconds for the second retry, and 10 seconds for the third retry:

```
public function backoff()
{
 return [1, 5, 10];
}
```

# Module 7

## Cleaning Up After Failed Jobs

When a particular job fails, you may want to send an alert to your users or revert any actions that were partially completed by the job. To accomplish this, you may define a failed method on your job class. The Throwable instance that caused the job to fail will be passed to the failed method:

```
<?php
namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
use Throwable;

class ProcessPodcast implements ShouldQueue
{
 use InteractsWithQueue, Queueable, SerializesModels;

 protected $podcast;
```

# Module 7

```
public function __construct(Podcast $podcast)
{
 $this->podcast = $podcast;
}
public function handle(AudioProcessor $processor)
{
 // Process uploaded podcast...
} public function failed(Throwable $exception)
{
 // Send user notification of failure, etc...
}
}
```

A new instance of the job is instantiated before invoking the failed method; therefore, any class property modifications that may have occurred within the handle method will be lost.

## Retrying Failed Jobs

To view all of the failed jobs that have been inserted into your failed\_jobs database table, you may use the queue:failed Artisan command:

```
php artisan queue:failed
```

# Module 7

The `queue:failed` command will list the job ID, connection, queue, failure time, and other information about the job. The job ID may be used to retry the failed job. For instance, to retry a failed job that has an ID of 5, issue the following command:

```
php artisan queue:retry 5
```

If necessary, you may pass multiple IDs or an ID range (when using numeric IDs) to the command:

```
php artisan queue:retry 5 6 7 8 9 10
```

```
php artisan queue:retry --range=5-10
```

You may also retry all of the failed jobs for a particular queue:

```
php artisan queue:retry --queue=name
```

To retry all of your failed jobs, execute the `queue:retry` command and pass all as the ID:

```
php artisan queue:retry all
```

If you would like to delete a failed job, you may use the `queue:forget` command:

```
php artisan queue:forget 5
```

When using [Horizon](#), you should use the `horizon:forget` command to delete a failed job instead of the `queue:forget` command.

To delete all of your failed jobs from the `failed_jobs` table, you may use the `queue:flush` command:

```
php artisan queue:flush
```

# Module 7

## Ignoring Missing Models

When injecting an Eloquent model into a job, the model is automatically serialized before being placed on the queue and re-retrieved from the database when the job is processed. However, if the model has been deleted while the job was waiting to be processed by a worker, your job may fail with a `ModelNotFoundException`.

For convenience, you may choose to automatically delete jobs with missing models by setting your job's `deleteWhenMissingModels` property to true. When this property is set to true, Laravel will quietly discard the job without raising an exception:

```
public $deleteWhenMissingModels = true;
```

## Storing Failed Jobs In DynamoDB

Laravel also provides support for storing your failed job records in [DynamoDB](#) instead of a relational database table. However, you must create a DynamoDB table to store all of the failed job records. Typically, this table should be named `failed_jobs`, but you should name the table based on the value of the `queue.failed.table` configuration value within your application's queue configuration file.

The `failed_jobs` table should have a string primary partition key named `application` and a string primary sort key named `uuid`. The `application` portion of the key will contain your application's name as defined by the `name` configuration value within your application's app configuration file. Since the application name is part of the DynamoDB table's key, you can use the same table to store failed jobs for multiple Laravel applications.

# Module 7

In addition, ensure that you install the AWS SDK so that your Laravel application can communicate with Amazon DynamoDB:

```
composer require aws/aws-sdk-php
```

Next, set the queue.failed.driver configuration option's value to dynamodb. In addition, you should define key, secret, and region configuration options within the failed job configuration array. These options will be used to authenticate with AWS. When using the dynamodb driver, the queue.failed.database configuration option is unnecessary:

```
'failed' => [
 'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
 'key' => env('AWS_ACCESS_KEY_ID'),
 'secret' => env('AWS_SECRET_ACCESS_KEY'),
 'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
 'table' => 'failed_jobs',
],
```

## Pruning Failed Jobs

You may delete all of the records in your application's failed\_jobs table by invoking the queue:prune-failed Artisan command:

```
php artisan queue:prune-failed
```

If you provide the --hours option to the command, only the failed job records that were inserted within the last N number of hours will be retained. For example, the following command will delete all of the failed job records that were inserted more than 48 hours ago:

# Module 7

```
php artisan queue:prune-failed --hours=48
```

## Failed Job Events

If you would like to register an event listener that will be invoked when a job fails, you may use the Queue facade's failing method. For example, we may attach a closure to this event from the boot method of the AppServiceProvider that is included with Laravel:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobFailed;

class AppServiceProvider extends ServiceProvider
{
 public function register()
 {
 //
 }
}
```

# Module 7

```
public function boot()
{
 Queue::failing(function (JobFailed $event) {
 // $event->connectionName
 // $event->job
 // $event->exception
 });
}

}
```

## [Clearing Jobs From Queues](#)

When using [Horizon](#), you should use the horizon:clear command to clear jobs from the queue instead of the queue:clear command.

If you would like to delete all jobs from the default queue of the default connection, you may do so using the queue:clear Artisan command:

```
php artisan queue:clear
```

You may also provide the connection argument and queue option to delete jobs from a specific connection and queue:

```
php artisan queue:clear redis --queue=emails
```

Clearing jobs from queues is only available for the SQS, Redis, and database queue drivers. In addition, the SQS message deletion process takes up to 60 seconds, so jobs sent to the SQS queue up to 60 seconds after you clear the queue might also be deleted.

# Module 7

## Monitoring Your Queues

If your queue receives a sudden influx of jobs, it could become overwhelmed, leading to a long wait time for jobs to complete. If you wish, Laravel can alert you when your queue job count exceeds a specified threshold.

To get started, you should schedule the `queue:monitor` command to [run every minute](#). The command accepts the names of the queues you wish to monitor as well as your desired job count threshold:

```
php artisan queue:monitor redis:default,redis:deployments --max=100
```

Scheduling this command alone is not enough to trigger a notification alerting you of the queue's overwhelmed status. When the command encounters a queue that has a job count exceeding your threshold, an `Illuminate\Queue\Events\QueueBusy` event will be dispatched. You may listen for this event within your application's `EventServiceProvider` in order to send a notification to you or your development team:

```
use App\Notifications\QueueHasLongWaitTime;
use Illuminate\Queue\Events\QueueBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;
public function boot()
{
 Event::listen(function (QueueBusy $event) {
```

# Module 7

```
Notification::route('mail', 'dev@example.com')
 ->notify(new QueueHasLongWaitTime(
 $event->connection,
 $event->queue,
 $event->size
)));
}
}
```

## Job Events

Using the before and after methods on the Queue [facade](#), you may specify callbacks to be executed before or after a queued job is processed. These callbacks are a great opportunity to perform additional logging or increment statistics for a dashboard. Typically, you should call these methods from the boot method of a [service provider](#). For example, we may use the AppServiceProvider that is included with Laravel:

```
<?php
```

```
namespace App\Providers;
use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;
```

# Module 7

```
class AppServiceProvider extends ServiceProvider
{
 public function register()
 {
 }

 public function boot()
 {
 Queue::before(function (JobProcessing $event) {
 // $event->connectionName
 // $event->job
 // $event->job->payload()
 });

 Queue::after(function (JobProcessed $event) {
 });
 }
}
```

Using the looping method on the Queue [facade](#), you may specify callbacks that execute before the worker attempts to fetch a job from a queue. For example, you might register a closure to rollback any transactions that were left open by a previously failed job:

# Module 7

```
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Queue;

Queue::looping(function () {
 while (DB::transactionLevel() > 0) {
 DB::rollBack();
 }
});
```

## Task Scheduling

### Introduction

In the past, you may have written a cron configuration entry for each task you needed to schedule on your server. However, this can quickly become a pain because your task schedule is no longer in source control and you must SSH into your server to view your existing cron entries or add additional entries.

Laravel's command scheduler offers a fresh approach to managing scheduled tasks on your server. The scheduler allows you to fluently and expressively define your command schedule within your Laravel application itself. When using the scheduler, only a single cron entry is needed on your server. Your task schedule is defined in the app\Console\Kernel.php file's schedule method. To help you get started, a simple example is defined within the method.

# Module 7

## Defining Schedules

You may define all of your scheduled tasks in the schedule method of your application's App\Console\Kernel class. To get started, let's take a look at an example. In this example, we will schedule a closure to be called every day at midnight. Within the closure we will execute a database query to clear a table:

```
<?php
namespace App\Console;
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\DB;
class Kernel extends ConsoleKernel
{
 protected $commands = [
 //
];
 protected function schedule(Schedule $schedule)
 {
 $schedule->call(function () {
 DB::table('recent_users')->delete();
 })->daily();
 }
}
```

# Module 7

In addition to scheduling using closures, you may also schedule [invokable objects](#). Invokable objects are simple PHP classes that contain an `__invoke` method:

```
$schedule->call(new DeleteRecentUsers)->daily();
```

If you would like to view an overview of your scheduled tasks and the next time they are scheduled to run, you may use the `schedule:list` Artisan command:

```
php artisan schedule:list
```

## Scheduling Artisan Commands

In addition to scheduling closures, you may also schedule [Artisan commands](#) and system commands. For example, you may use the `command` method to schedule an Artisan command using either the command's name or class.

When scheduling Artisan commands using the command's class name, you may pass an array of additional command-line arguments that should be provided to the command when it is invoked:

```
use App\Console\Commands\SendEmailsCommand;
$schedule->command('emails:send Taylor --force')->daily();
```

```
$schedule->command(SendEmailsCommand::class, ['Taylor', '--force'])->daily();
```

# Module 7

## Scheduling Queued Jobs

The job method may be used to schedule a [queued job](#). This method provides a convenient way to schedule queued jobs without using the call method to define closures to queue the job:

```
use App\Jobs\Heartbeat;
$schedule->job(new Heartbeat)->everyFiveMinutes();
```

Optional second and third arguments may be provided to the job method which specifies the queue name and queue connection that should be used to queue the job:

```
use App\Jobs\Heartbeat;
$schedule->job(new Heartbeat, 'heartbeats', 'sqS')->everyFiveMinutes();
```

## Scheduling Shell Commands

The exec method may be used to issue a command to the operating system:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

## Schedule Frequency Options

We've already seen a few examples of how you may configure a task to run at specified intervals. However, there are many more task schedule frequencies that you may assign to a task:

| Method                                 | Description                            |
|----------------------------------------|----------------------------------------|
| <code>-&gt;cron('* * * * *');</code>   | Run the task on a custom cron schedule |
| <code>-&gt;everyMinute();</code>       | Run the task every minute              |
| <code>-&gt;everyTwoMinutes();</code>   | Run the task every two minutes         |
| <code>-&gt;everyThreeMinutes();</code> | Run the task every three minutes       |
| <code>-&gt;everyFourMinutes();</code>  | Run the task every four minutes        |
| <code>-&gt;everyFiveMinutes();</code>  | Run the task every five minutes        |
| <code>-&gt;everyTenMinutes();</code>   | Run the task every ten minutes         |

|                                          |                                                     |
|------------------------------------------|-----------------------------------------------------|
| <code>-&gt;everyFifteenMinutes();</code> | Run the task every fifteen minutes                  |
| <code>-&gt;everyThirtyMinutes();</code>  | Run the task every thirty minutes                   |
| <code>-&gt;hourly();</code>              | Run the task every hour                             |
| <code>-&gt;hourlyAt(17);</code>          | Run the task every hour at 17 minutes past the hour |
| <code>-&gt;everyTwoHours();</code>       | Run the task every two hours                        |
| <code>-&gt;everyThreeHours();</code>     | Run the task every three hours                      |
| <code>-&gt;everyFourHours();</code>      | Run the task every four hours                       |
| <code>-&gt;everySixHours();</code>       | Run the task every six hours                        |
| <code>-&gt;daily();</code>               | Run the task every day at midnight                  |
| <code>-&gt;dailyAt('13:00');</code>      | Run the task every day at 13:00                     |
| <code>-&gt;twiceDaily(1, 13);</code>     | Run the task daily at 1:00 & 13:00                  |
| <code>-&gt;weekly();</code>              | Run the task every Sunday at 00:00                  |

|                                                 |                                                         |
|-------------------------------------------------|---------------------------------------------------------|
| <code>-&gt;weeklyOn(1, '8:00');</code>          | Run the task every week on Monday at 8:00               |
| <code>-&gt;monthly();</code>                    | Run the task on the first day of every month at 00:00   |
| <code>-&gt;monthlyOn(4, '15:00');</code>        | Run the task every month on the 4th at 15:00            |
| <code>-&gt;twiceMonthly(1, 16, '13:00');</code> | Run the task monthly on the 1st and 16th at 13:00       |
| <code>-&gt;lastDayOfMonth('15:00');</code>      | Run the task on the last day of the month at 15:00      |
| <code>-&gt;quarterly();</code>                  | Run the task on the first day of every quarter at 00:00 |
| <code>-&gt;yearly();</code>                     | Run the task on the first day of every year at 00:00    |
| <code>-&gt;yearlyOn(6, 1, '17:00');</code>      | Run the task every year on June 1st at 17:00            |
| <code>-&gt;timezone('America/New_York');</code> | Set the timezone for the task                           |

# Module 7

- These methods may be combined with additional constraints to create even more finely tuned schedules that only run on certain days of the week. For example, you may schedule a command to run weekly on Monday:

```
// Run once per week on Monday at 1 PM...
```

```
$schedule->call(function () {
```

```
//
```

```
)>weekly()->mondays()->at('13:00');
```

```
// Run hourly from 8 AM to 5 PM on weekdays...
```

```
$schedule->command('foo')
```

```
->weekdays()
```

```
->hourly()
```

```
->timezone('America/Chicago')
```

```
->between('8:00', '17:00');
```

- A list of additional schedule constraints may be found below:

| Method                                   | Description                                           |
|------------------------------------------|-------------------------------------------------------|
| ->weekdays();                            | Limit the task to weekdays                            |
| ->weekends();                            | Limit the task to weekends                            |
| ->sundays();                             | Limit the task to Sunday                              |
| ->mondays();                             | Limit the task to Monday                              |
| ->tuesdays();                            | Limit the task to Tuesday                             |
| ->wednesdays();                          | Limit the task to Wednesday                           |
| ->thursdays();                           | Limit the task to Thursday                            |
| ->fridays();                             | Limit the task to Friday                              |
| ->saturdays();                           | Limit the task to Saturday                            |
| ->days(array mixed);                     | Limit the task to specific days                       |
| ->between(\$startTime, \$endTime);       | Limit the task to run between start and end times     |
| ->unlessBetween(\$startTime, \$endTime); | Limit the task to not run between start and end times |
| ->when(Closure);                         | Limit the task based on a truth test                  |
| ->environments(\$env);                   | Limit the task to specific environments               |

# Module 7

## Day Constraints

The days method may be used to limit the execution of a task to specific days of the week. For example, you may schedule a command to run hourly on Sundays and Wednesdays:

```
$schedule->command('emails:send')
->hourly()
->days([0, 3]);
```

Alternatively, you may use the constants available on the Illuminate\Console\Scheduling\Schedule class when defining the days on which a task should run:

```
use Illuminate\Console\Scheduling\Schedule;

$schedule->command('emails:send')
->hourly()
->days([Schedule::SUNDAY, Schedule::WEDNESDAY]);
```

## Between Time Constraints

The between method may be used to limit the execution of a task based on the time of day:

```
$schedule->command('emails:send')
->hourly()
->between('7:00', '22:00');
```

# Module 7

Similarly, the `unlessBetween` method can be used to exclude the execution of a task for a period of time:

```
$schedule->command('emails:send')
 ->hourly()
 ->unlessBetween('23:00', '4:00');
```

## Truth Test Constraints

The `when` method may be used to limit the execution of a task based on the result of a given truth test. In other words, if the given closure returns true, the task will execute as long as no other constraining conditions prevent the task from running:

```
$schedule->command('emails:send')->daily()->when(function () {
 return true;
});
```

The `skip` method may be seen as the inverse of `when`. If the `skip` method returns true, the scheduled task will not be executed:

```
$schedule->command('emails:send')->daily()->skip(function () {
 return true;
});
```

When using chained `when` methods, the scheduled command will only execute if all `when` conditions return true.

# Module 7

## Environment Constraints

The environments method may be used to execute tasks only on the given environments (as defined by the APP\_ENV [environment variable](#)):

```
$schedule->command('emails:send')
 ->daily()
 ->environments(['staging', 'production']);
```

## Timezones

Using the timezone method, you may specify that a scheduled task's time should be interpreted within a given timezone:

```
$schedule->command('report:generate')
 ->timezone('America/New_York')
 ->at('2:00')
```

If you are repeatedly assigning the same timezone to all of your scheduled tasks, you may wish to define a scheduleTimezone method in your App\Console\Kernel class. This method should return the default timezone that should be assigned to all scheduled tasks:

```
protected function scheduleTimezone()
{
 return 'America/Chicago';
}
```

Remember that some timezones utilize daylight savings time. When daylight saving time changes occur, your scheduled task may run twice or even not run at all. For this reason, we recommend avoiding timezone scheduling when possible.

# Module 7

## Preventing Task Overlaps

By default, scheduled tasks will be run even if the previous instance of the task is still running. To prevent this, you may use the `withoutOverlapping` method:

```
$schedule->command('emails:send')->withoutOverlapping();
```

In this example, the `emails:send` [Artisan command](#) will be run every minute if it is not already running. The `withoutOverlapping` method is especially useful if you have tasks that vary drastically in their execution time, preventing you from predicting exactly how long a given task will take.

If needed, you may specify how many minutes must pass before the "without overlapping" lock expires. By default, the lock will expire after 24 hours:

```
$schedule->command('emails:send')->withoutOverlapping(10);
```

## Running Tasks On One Server

To utilize this feature, your application must be using the database, memcached, dynamodb, or redis cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

If your application's scheduler is running on multiple servers, you may limit a scheduled job to only execute on a single server. For instance, assume you have a scheduled task that generates a new report every Friday night. If the task scheduler is running on three worker servers, the scheduled task will run on all three servers and generate the report three times. Not good!

# Module 7

To indicate that the task should run on only one server, use the `onOneServer` method when defining the scheduled task. The first server to obtain the task will secure an atomic lock on the job to prevent other servers from running the same task at the same time:

```
$schedule->command('report:generate')
 ->fridays()
 ->at('17:00')
 ->onOneServer();
```

## Background Tasks

By default, multiple tasks scheduled at the same time will execute sequentially based on the order they are defined in your `schedule` method. If you have long-running tasks, this may cause subsequent tasks to start much later than anticipated. If you would like to run tasks in the background so that they may all run simultaneously, you may use the `runInBackground` method:

```
$schedule->command('analytics:report')
 ->daily()
 ->runInBackground();
```

The `runInBackground` method may only be used when scheduling tasks via the `command` and `exec` methods.

# Module 7

## Maintenance Mode

Your application's scheduled tasks will not run when the application is in [maintenance mode](#), since we don't want your tasks to interfere with any unfinished maintenance you may be performing on your server. However, if you would like to force a task to run even in maintenance mode, you may call the `evenInMaintenanceMode` method when defining the task:

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

## Running The Scheduler

Now that we have learned how to define scheduled tasks, let's discuss how to actually run them on our server. The `schedule:run` Artisan command will evaluate all of your scheduled tasks and determine if they need to run based on the server's current time.

So, when using Laravel's scheduler, we only need to add a single cron configuration entry to our server that runs the `schedule:run` command every minute. If you do not know how to add cron entries to your server, consider using a service such as [Laravel Forge](#) which can manage the cron entries for you:

```
* * * * * cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1
```

## Running The Scheduler Locally

Typically, you would not add a scheduler cron entry to your local development machine. Instead, you may use the `schedule:work` Artisan command. This command will run in the foreground and invoke the scheduler every minute until you terminate the command:

```
php artisan schedule:work
```

# Module 7

## Task Output

The Laravel scheduler provides several convenient methods for working with the output generated by scheduled tasks. First, using the `sendOutputTo` method, you may send the output to a file for later inspection:

```
$schedule->command('emails:send')
 ->daily()
 ->sendOutputTo($filePath);
```

If you would like to append the output to a given file, you may use the `appendOutputTo` method:

```
$schedule->command('emails:send')
 ->daily()
 ->appendOutputTo($filePath);
```

Using the `emailOutputTo` method, you may email the output to an email address of your choice. Before emailing the output of a task, you should configure Laravel's [email services](#):

```
$schedule->command('report:generate')
 ->daily()
 ->sendOutputTo($filePath)
 ->emailOutputTo('taylor@example.com');
```

If you only want to email the output if the scheduled Artisan or system command terminates with a non-zero exit code, use the `emailOutputOnFailure` method:

```
$schedule->command('report:generate')
```

# Module 7

```
$schedule->command('emails:send')
 ->daily()
 ->emailOutputOnFailure('taylor@example.com');
```

The `emailOutputTo`, `emailOutputOnFailure`, `sendOutputTo`, and `appendOutputTo` methods are exclusive to the command and exec methods.

## Task Hooks

Using the `before` and `after` methods, you may specify code to be executed before and after the scheduled task is executed:

```
$schedule->command('emails:send')
 ->daily()
 ->before(function () {
 // The task is about to execute...
 })
 ->after(function () {
 // The task has executed...
 });
```

The `onSuccess` and `onFailure` methods allow you to specify code to be executed if the scheduled task succeeds or fails. A failure indicates that the scheduled Artisan or system command terminated with a non-zero exit code:

```
$schedule->command('emails:send')
```

# Module 7

```
->daily()
 ->onSuccess(function () {
 // The task succeeded...
 })
 ->onFailure(function () {
 // The task failed...
 });
```

If output is available from your command, you may access it in your after, onSuccess or onFailure hooks by type-hinting an Illuminate\Support\Stringable instance as the \$output argument of your hook's closure definition:

```
use Illuminate\Support\Stringable;

$schedule->command('emails:send')
 ->daily()
 ->onSuccess(function (Stringable $output) {
 // The task succeeded...
 })
 ->onFailure(function (Stringable $output) {
 // The task failed...
 });
```

# Module 7

- **Pinging URLs**

Using the pingBefore and thenPing methods, the scheduler can automatically ping a given URL before or after a task is executed. This method is useful for notifying an external service, such as [Envoyer](#), that your scheduled task is beginning or has finished execution:

```
$schedule->command('emails:send')
 ->daily()
 ->pingBefore($url)
 ->thenPing($url);
```

The pingBeforeIf and thenPingIf methods may be used to ping a given URL only if a given condition is true:

```
$schedule->command('emails:send')
 ->daily()
 ->pingBeforeIf($condition, $url)
 ->thenPingIf($condition, $url);
```

The pingOnSuccess and pingOnFailure methods may be used to ping a given URL only if the task succeeds or fails. A failure indicates that the scheduled Artisan or system command terminated with a non-zero exit code:

```
$schedule->command('emails:send')
 ->daily()
 ->pingOnSuccess($successUrl)
```

# Module 7

```
->pingOnFailure($failureUrl);
```

All of the ping methods require the Guzzle HTTP library. Guzzle is typically installed in all new Laravel projects by default, but, you may manually install Guzzle into your project using the Composer package manager if it has been accidentally removed:

```
composer require guzzlehttp/guzzle
```

## Eloquent: Getting Started

### Introduction

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.

Before getting started, be sure to configure a database connection in your application's config/database.php configuration file. For more information on configuring your database, check out [the database configuration documentation](#).

### Generating Model Classes

To get started, let's create an Eloquent model. Models typically live in the app\Models directory and extend the Illuminate\Database\Eloquent\Model class. You may use the make:model [Artisan command](#) to generate a new model:

# Module 7

```
php artisan make:model Flight
```

If you would like to generate a [database migration](#) when you generate the model, you may use the --migration or -m option:

```
php artisan make:model Flight --migration
```

You may generate various other types of classes when generating a model, such as factories, seeders, and controllers. In addition, these options may be combined to create multiple classes at once:

```
Generate a model and a FlightFactory class...
```

```
php artisan make:model Flight --factory
```

```
php artisan make:model Flight -f
```

```
Generate a model and a FlightSeeder class...
```

```
php artisan make:model Flight --seed
```

```
php artisan make:model Flight -s
```

```
Generate a model and a FlightController class...
```

```
php artisan make:model Flight --controller
```

```
php artisan make:model Flight -c
```

```
Generate a model and a migration, factory, seeder, and controller...
```

```
php artisan make:model Flight -mfsc
```

# Module 7

```
Shortcut to generate a model, migration, factory, seeder, and controller...
php artisan make:model Flight --all
```

```
Generate a pivot model...
php artisan make:model Member --pivot
```

## Eloquent Model Conventions

Models generated by the `make:model` command will be placed in the `app/Models` directory. Let's examine a basic model class and discuss some of Eloquent's key conventions:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
 //
}
```

# Module 7

## Table Names

After glancing at the example above, you may have noticed that we did not tell Eloquent which database table corresponds to our Flight model. By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the Flight model stores records in the flights table, while an AirTrafficController model would store records in an air\_traffic\_controllers table.

If your model's corresponding database table does not fit this convention, you may manually specify the model's table name by defining a table property on the model:

```
<?php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{
 /**
 * The table associated with the model.
 *
 * @var string
 */
 protected $table = 'my_flights';
}
```

# Module 7

## Primary Keys

Eloquent will also assume that each model's corresponding database table has a primary key column named id. If necessary, you may define a protected \$primaryKey property on your model to specify a different column that serves as your model's primary key:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
 /**
 * The primary key associated with the table.
 *
 * @var string
 */
 protected $primaryKey = 'flight_id';
}
```

# Module 7

- In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that Eloquent will automatically cast the primary key to an integer. If you wish to use a non-incrementing or a non-numeric primary key you must define a public `$incrementing` property on your model that is set to false:

```
class Flight extends Model
{
 public $incrementing = false;
}
```

- If your model's primary key is not an integer, you should define a protected `$keyType` property on your model. This property should have a value of string:

```
<?php
class Flight extends Model
{
 /**
 * The data type of the auto-incrementing ID.
 *
 * @var string
 */
 protected $keyType = 'string';
}
```

# Module 7

## "Composite" Primary Keys

Eloquent requires each model to have at least one uniquely identifying "ID" that can serve as its primary key. "Composite" primary keys are not supported by Eloquent models. However, you are free to add additional multi-column, unique indexes to your database tables in addition to the table's uniquely identifying primary key.

## Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your model's corresponding database table. Eloquent will automatically set these column's values when models are created or updated. If you do not want these columns to be automatically managed by Eloquent, you should define a `$timestamps` property on your model with a value of false:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{
 /**
 * Indicates if the model should be timestamped.
 *
 * @var bool
 */
 public $timestamps = false;
}
```

# Module 7

- If you need to customize the format of your model's timestamps, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database as well as their format when the model is serialized to an array or JSON:

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{
 /**
 * The storage format of the model's date columns.
 *
 * @var string
 */
 protected $dateFormat = 'U';
}
```

- If you need to customize the names of the columns used to store the timestamps, you may define `CREATED_AT` and `UPDATED_AT` constants on your model:

```
<?php
```

```
class Flight extends Model
```

# Module 7

```
{
 const CREATED_AT = 'creation_date';
 const UPDATED_AT = 'updated_date';
}
```

- **Database Connections**
- By default, all Eloquent models will use the default database connection that is configured for your application. If you would like to specify a different connection that should be used when interacting with a particular model, you should define a `$connection` property on the model:

```
<?php

namespace App\Models;
use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
 /**
 * The database connection that should be used by the model.
 *
 * @var string
 */
 protected $connection = 'sqlite';
}
```

# Module 7

- **Default Attribute Values**
- By default, a newly instantiated model instance will not contain any attribute values. If you would like to define the default values for some of your model's attributes, you may define an `$attributes` property on your model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
 /**
 * The model's default values for attributes.
 *
 * @var array
 */
 protected $attributes = [
 'delayed' => false,
];
}
```

# Module 7

## Retrieving Models

Once you have created a model and [its associated database table](#), you are ready to start retrieving data from your database. You can think of each Eloquent model as a powerful [query builder](#) allowing you to fluently query the database table associated with the model. The model's all method will retrieve all of the records from the model's associated database table:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
 echo $flight->name;
}
```

## Building Queries

The Eloquent all method will return all of the results in the model's table. However, since each Eloquent model serves as a [query builder](#), you may add additional constraints to queries and then invoke the get method to retrieve the results:

```
$flights = Flight::where('active', 1)
 ->orderBy('name')
 ->take(10)
 ->get();
```

Since Eloquent models are query builders, you should review all of the methods provided by Laravel's [query builder](#). You may use any of these methods when writing your Eloquent queries.

# Module 7

## Refreshing Models

If you already have an instance of an Eloquent model that was retrieved from the database, you can "refresh" the model using the `fresh` and `refresh` methods. The `fresh` method will re-retrieve the model from the database. The existing model instance will not be affected:

```
$flight = Flight::where('number', 'FR 900')->first();
```

```
$freshFlight = $flight->fresh();
```

The `refresh` method will re-hydrate the existing model using fresh data from the database. In addition, all of its loaded relationships will be refreshed as well:

```
$flight = Flight::where('number', 'FR 900')->first();
```

```
$flight->number = 'FR 456';
```

```
$flight->refresh();
```

```
$flight->number; // "FR 900"
```

## Collections

As we have seen, Eloquent methods like `all` and `get` retrieve multiple records from the database. However, these methods don't return a plain PHP array. Instead, an instance of `Illuminate\Database\Eloquent\Collection` is returned.

# Module 7

The Eloquent Collection class extends Laravel's base Illuminate\Support\Collection class, which provides a [variety of helpful methods](#) for interacting with data collections. For example, the reject method may be used to remove models from a collection based on the results of an invoked closure:

```
$flights = Flight::where('destination', 'Paris')->get();
```

```
$flights = $flights->reject(function ($flight) {
 return $flight->cancelled;
});
```

In addition to the methods provided by Laravel's base collection class, the Eloquent collection class provides [a few extra methods](#) that are specifically intended for interacting with collections of Eloquent models.

Since all of Laravel's collections implement PHP's iterable interfaces, you may loop over collections as if they were an array:

```
foreach ($flights as $flight) {
 echo $flight->name;
}
```

# Module 7

## Chunking Results

Your application may run out of memory if you attempt to load tens of thousands of Eloquent records via the all or get methods. Instead of using these methods, the chunk method may be used to process large numbers of models more efficiently.

The chunk method will retrieve a subset of Eloquent models, passing them to a closure for processing. Since only the current chunk of Eloquent models is retrieved at a time, the chunk method will provide significantly reduced memory usage when working with a large number of models:

```
use App\Models\Flight;
Flight::chunk(200, function ($flights) {
 foreach ($flights as $flight) {
 }
});
```

The first argument passed to the chunk method is the number of records you wish to receive per "chunk". The closure passed as the second argument will be invoked for each chunk that is retrieved from the database. A database query will be executed to retrieve each chunk of records passed to the closure.

If you are filtering the results of the chunk method based on a column that you will also be updating while iterating over the results, you should use the chunkById method. Using the chunk method in these scenarios could lead to unexpected and inconsistent results. Internally, the chunkById method will always retrieve models with an id column greater than the last model in the previous chunk:

# Module 7

```
Flight::where('departed', true)
->chunkById(200, function ($flights) {
 $flights->each->update(['departed' => false]);
}, $column = 'id');
```

## Streaming Results Lazily

The lazy method works similarly to [the chunk method](#) in the sense that, behind the scenes, it executes the query in chunks. However, instead of passing each chunk directly into a callback as is, the lazy method returns a flattened [LazyCollection](#) of Eloquent models, which lets you interact with the results as a single stream:

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
 //
}
```

If you are filtering the results of the lazy method based on a column that you will also be updating while iterating over the results, you should use the lazyById method. Internally, the lazyById method will always retrieve models with an id column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
->lazyById(200, $column = 'id')
->each->update(['departed' => false]);
```

# Module 7

## Cursors

Similar to the [lazy](#) method, the cursor method may be used to significantly reduce your application's memory consumption when iterating through tens of thousands of Eloquent model records.

The cursor method will only execute a single database query; however, the individual Eloquent models will not be hydrated until they are actually iterated over. Therefore, only one Eloquent model is kept in memory at any given time while iterating over the cursor.

Since the cursor method only ever holds a single Eloquent model in memory at a time, it cannot eager load relationships. If you need to eager load relationships, consider using [the lazy method](#) instead.

Internally, the cursor method uses PHP [generators](#) to implement this functionality:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
 //
}
```

The cursor returns an `Illuminate\Support\LazyCollection` instance. [Lazy collections](#) allow you to use many of the collection methods available on typical Laravel collections while only loading a single model into memory at a time:

# Module 7

```
use App\Models\User;

$users = User::cursor()->filter(function ($user) {
 return $user->id > 500;
});

foreach ($users as $user) {
 echo $user->id;
}
```

Although the cursor method uses far less memory than a regular query (by only holding a single Eloquent model in memory at a time), it will still eventually run out of memory. This is [due to PHP's PDO driver internally caching all raw query results in its buffer](#). If you're dealing with a very large number of Eloquent records, consider using [the lazy method](#) instead.

## Advanced Subqueries

### Subquery Selects

Eloquent also offers advanced subquery support, which allows you to pull information from related tables in a single query. For example, let's imagine that we have a table of flight destinations and a table of flights to destinations. The flights table contains an `arrived_at` column which indicates when the flight arrived at the destination.

# Module 7

Using the subquery functionality available to the query builder's select and addSelect methods, we can select all of the destinations and the name of the flight that most recently arrived at that destination using a single query:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
 ->whereColumn('destination_id', 'destinations.id')
 ->orderByDesc('arrived_at')
 ->limit(1)
])->get();
```

## Subquery Ordering

In addition, the query builder's orderBy function supports subqueries. Continuing to use our flight example, we may use this functionality to sort all destinations based on when the last flight arrived at that destination. Again, this may be done while executing a single database query:

```
return Destination::orderByDesc(
 Flight::select('arrived_at')
 ->whereColumn('destination_id', 'destinations.id')
 ->orderByDesc('arrived_at')
 ->limit(1)
)->get();
```

# Module 7

## Retrieving Single Models / Aggregates

In addition to retrieving all of the records matching a given query, you may also retrieve single records using the `find`, `first`, or `firstWhere` methods. Instead of returning a collection of models, these methods return a single model instance:

```
use App\Models\Flight;

// Retrieve a model by its primary key...
$flight = Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();

$flight = Flight::firstWhere('active', 1);
```

Sometimes you may wish to retrieve the first result of a query or perform some other action if no results are found. The `firstOr` method will return the first result matching the query or, if no results are found, execute the given closure. The value returned by the closure will be considered the result of the `firstOr` method:

```
$model = Flight::where('legs', '>', 3)->firstOr(function () {
 // ...
});
```

# Module 7

## Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query; however, if no result is found, an `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$flight = Flight::findOrFail(1);
```

```
$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

If the `ModelNotFoundException` is not caught, a 404 HTTP response is automatically sent back to the client:

```
use App\Models\Flight;
```

```
Route::get('/api/flights/{id}', function ($id) {
 return Flight::findOrFail($id);
});
```

## Retrieving Or Creating Models

The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model can not be found in the database, a record will be inserted with the attributes resulting from merging the first array argument with the optional second array argument:

# Module 7

The `firstOrNew` method, like `firstOrCreate`, will attempt to locate a record in the database matching the given attributes. However, if a model is not found, a new model instance will be returned. Note that the model returned by `firstOrNew` has not yet been persisted to the database. You will need to manually call the `save` method to persist it:

```
use App\Models\Flight;
// Retrieve flight by name or create it if it doesn't exist...
$flight = Flight::firstOrCreate([
 'name' => 'London to Paris'
]);
// Retrieve flight by name or create it with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrCreate(
 ['name' => 'London to Paris'],
 ['delayed' => 1, 'arrival_time' => '11:30']
);
// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrNew([
 'name' => 'London to Paris'
]);
// Retrieve flight by name or instantiate with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrNew(
 ['name' => 'Tokyo to Sydney'],
 ['delayed' => 1, 'arrival_time' => '11:30']
);
```

# Module 7

## Retrieving Aggregates

When interacting with Eloquent models, you may also use the count, sum, max, and other [aggregate methods](#) provided by the Laravel [query builder](#). As you might expect, these methods return a scalar value instead of an Eloquent model instance:

```
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

## Inserting & Updating Models

Of course, when using Eloquent, we don't only need to retrieve models from the database. We also need to insert new records. Thankfully, Eloquent makes it simple. To insert a new record into the database, you should instantiate a new model instance and set attributes on the model. Then, call the save method on the model instance:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Flight;
use Illuminate\Http\Request;

class FlightController extends Controller
{
```

# Module 7

```
public function store(Request $request)
{
 // Validate the request...

 $flight = new Flight;

 $flight->name = $request->name;

 $flight->save();
}
```

In this example, we assign the name field from the incoming HTTP request to the name attribute of the App\Models\Flight model instance. When we call the save method, a record will be inserted into the database. The model's created\_at and updated\_at timestamps will automatically be set when the save method is called, so there is no need to set them manually.

Alternatively, you may use the create method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the create method:

```
use App\Models\Flight;

$flight = Flight::create([
 'name' => 'London to Paris',
]);
```

# Module 7

However, before using the `create` method, you will need to specify either a fillable or guarded property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default. To learn more about mass assignment, please consult the [mass assignment documentation](#).

## Updates

The `save` method may also be used to update models that already exist in the database. To update a model, you should retrieve it and set any attributes you wish to update. Then, you should call the model's `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
use App\Models\Flight;
$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();
```

## Mass Updates

Updates can also be performed against models that match a given query. In this example, all flights that are active and have a destination of San Diego will be marked as delayed:

```
Flight::where('active', 1)
 ->where('destination', 'San Diego')
 ->update(['delayed' => 1]);
```

# Module 7

The update method expects an array of column and value pairs representing the columns that should be updated.

When issuing a mass update via Eloquent, the saving, saved, updating, and updated model events will not be fired for the updated models. This is because the models are never actually retrieved when issuing a mass update.

## Examining Attribute Changes

Eloquent provides the isDirty, isClean, and wasChanged methods to examine the internal state of your model and determine how its attributes have changed from when the model was originally retrieved.

The isDirty method determines if any of the model's attributes have been changed since the model was retrieved. You may pass a specific attribute name to the isDirty method to determine if a particular attribute is dirty. The isClean will determine if an attribute has remained unchanged since the model was retrieved. This method also accepts an optional attribute argument:

```
use App\Models\User;

$user = User::create([
 'first_name' => 'Taylor',
 'last_name' => 'Otwell',
 'title' => 'Developer',
]);
```

# Module 7

```
$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

The `wasChanged` method determines if any attributes were changed when the model was last saved within the current request cycle. If needed, you may pass an attribute name to see if a particular attribute was changed:

# Module 7

```
$user = User::create([
 'first_name' => 'Taylor',
 'last_name' => 'Otwell',
 'title' => 'Developer',
]);
$user->title = 'Painter';
$user->save();
$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged('first_name'); // false
```

- The `getOriginal` method returns an array containing the original attributes of the model regardless of any changes to the model since it was retrieved. If needed, you may pass a specific attribute name to get the original value of a particular attribute:

```
$user = User::find(1);
$user->name; // John
$user->email; // john@example.com
```

```
$user->name = "Jack";
$user->name; // Jack
```

```
$user->getOriginal('name'); // John
$user->getOriginal();
```

# Module 7

- **Mass Assignment**
- You may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the method:

```
use App\Models\Flight;

$flight = Flight::create([
 'name' => 'London to Paris',
]);
```
- However, before using the `create` method, you will need to specify either a fillable or guarded property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default.
- A mass assignment vulnerability occurs when a user passes an unexpected HTTP request field and that field changes a column in your database that you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed to your model's `create` method, allowing the user to escalate themselves to an administrator.
- So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the `$fillable` property on the model. For example, let's make the `name` attribute of our `Flight` model mass assignable:

# Module 7

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
 /**
 * The attributes that are mass assignable.
 *
 * @var array
 */
 protected $fillable = ['name'];
}
```

Once you have specified which attributes are mass assignable, you may use the `create` method to insert a new record in the database. The `create` method returns the newly created model instance:

```
$flight = Flight::create(['name' => 'London to Paris']);
```

If you already have a model instance, you may use the `fill` method to populate it with an array of attributes:

# Module 7

```
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

## Mass Assignment & JSON Columns

When assigning JSON columns, each column's mass assignable key must be specified in your model's `$fillable` array. For security, Laravel does not support updating nested JSON attributes when using the `guarded` property:

```
protected $fillable = [
 'options->enabled',
];
```

## Allowing Mass Assignment

If you would like to make all of your attributes mass assignable, you may define your model's `$guarded` property as an empty array. If you choose to unguard your model, you should take special care to always hand-craft the arrays passed to Eloquent's `fill`, `create`, and `update` methods:

```
protected $guarded = [];
```

## Upserts

Occasionally, you may need to update an existing model or create a new model if no matching model exists. Like the `firstOrCreate` method, the `updateOrCreate` method persists the model, so there's no need to manually call the `save` method.

# Module 7

- In the example below, if a flight exists with a departure location of Oakland and a destination location of San Diego, its price and discounted columns will be updated. If no such flight exists, a new flight will be created which has the attributes resulting from merging the first argument array with the second argument array:

```
$flight = Flight::updateOrCreate(
 ['departure' => 'Oakland', 'destination' => 'San Diego'],
 ['price' => 99, 'discounted' => 1]
>;
```

- If you would like to perform multiple "upserts" in a single query, then you should use the upsert method instead. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of the columns that should be updated if a matching record already exists in the database. The upsert method will automatically set the created\_at and updated\_at timestamps if timestamps are enabled on the model

```
Flight::upsert([
 ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
 ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
>, ['departure', 'destination'], ['price']);
```

- All databases systems except SQL Server require the columns in the second argument provided to the upsert method to have a "primary" or "unique" index.

# Module 7

## Deleting Models

To delete a model, you may call the `delete` method on the model instance:

```
use App\Models\Flight;
$flight = Flight::find(1);
$flight->delete();
```

You may call the `truncate` method to delete all of the model's associated database records. The `truncate` operation will also reset any auto-incrementing IDs on the model's associated table:

```
Flight::truncate();
```

## Deleting An Existing Model By Its Primary Key

In the example above, we are retrieving the model from the database before calling the `delete` method. However, if you know the primary key of the model, you may delete the model without explicitly retrieving it by calling the `destroy` method. In addition to accepting the single primary key, the `destroy` method will accept multiple primary keys, an array of primary keys, or a [collection](#) of primary keys:

```
Flight::destroy(1); Flight::destroy(1, 2, 3);
Flight::destroy([1, 2, 3]);
Flight::destroy(collect([1, 2, 3]));
```

# Module 7

The `destroy` method loads each model individually and calls the `delete` method so that the deleting and deleted events are properly dispatched for each model.

## [Deleting Models Using Queries](#)

Of course, you may build an Eloquent query to delete all models matching your query's criteria. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not dispatch model events for the models that are deleted:

```
$deletedRows = Flight::where('active', 0)->delete();
```

When executing a mass delete statement via Eloquent, the deleting and deleted model events will not be dispatched for the deleted models. This is because the models are never actually retrieved when executing the delete statement.

## [Soft Deleting](#)

In addition to actually removing records from your database, Eloquent can also "soft delete" models. When models are soft deleted, they are not actually removed from your database. Instead, a `deleted_at` attribute is set on the model indicating the date and time at which the model was "deleted". To enable soft deletes for a model, add the `\Illuminate\Database\Eloquent\SoftDeletes` trait to the model:

# Module 7

```
<?php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
class Flight extends Model
{
 use SoftDeletes;
}
```

The SoftDeletes trait will automatically cast the deleted\_at attribute to a DateTime / Carbon instance for you.

You should also add the deleted\_at column to your database table. The Laravel [schema builder](#) contains a helper method to create this column:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
 $table->softDeletes();
});
Schema::table('flights', function (Blueprint $table) {
 $table->dropSoftDeletes();
});
```

# Module 7

Now, when you call the delete method on the model, the deleted\_at column will be set to the current date and time. However, the model's database record will be left in the table. When querying a model that uses soft deletes, the soft deleted models will automatically be excluded from all query results.

To determine if a given model instance has been soft deleted, you may use the trashed method:

```
if ($flight->trashed()) {
 //
}
```

## [Restoring Soft Deleted Models](#)

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model, you may call the restore method on a model instance. The restore method will set the model's deleted\_at column to null:

```
$flight->restore();
```

You may also use the restore method in a query to restore multiple models. Again, like other "mass" operations, this will not dispatch any model events for the models that are restored:

```
Flight::withTrashed()
 ->where('airline_id', 1)
 ->restore();
```

The restore method may also be used when building [relationship](#) queries:

```
$flight->history()->restore();
```

# Module 7

Sometimes you may need to truly remove a model from your database. You may use the `forceDelete` method to permanently remove a soft deleted model from the database table:

```
$flight->forceDelete();
```

You may also use the `forceDelete` method when building Eloquent relationship queries:

```
$flight->history()->forceDelete();
```

## [Querying Soft Deleted Models](#)

## [Including Soft Deleted Models](#)

As noted above, soft deleted models will automatically be excluded from query results. However, you may force soft deleted models to be included in a query's results by calling the `withTrashed` method on the query:

```
use App\Models\Flight;
$flights = Flight::withTrashed()
 ->where('account_id', 1)
 ->get();
```

The `withTrashed` method may also be called when building a [relationship](#) query:

```
$flight->history()->withTrashed()->get();
```

## [Retrieving Only Soft Deleted Models](#)

The `onlyTrashed` method will retrieve only soft deleted models:

```
$flights = Flight::onlyTrashed()
 ->where('airline_id', 1)
 ->get();
```

## Pruning Models

Sometimes you may want to periodically delete models that are no longer needed. To accomplish this, you may add the Illuminate\Database\Eloquent\Prunable or Illuminate\Database\Eloquent\MassPrunable trait to the models you would like to periodically prune. After adding one of the traits to the model, implement a prunable method which returns an Eloquent query builder that resolves the models that are no longer needed:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
 use Prunable;
 public function prunable()
 {
 return static::where('created_at', '<=', now()->subMonth());
 }
}
```

When marking models as Prunable, you may also define a pruning method on the model. This method will be called before the model is deleted. This method can be useful for deleting any additional resources associated with the model, such as stored files, before the model is permanently removed from the database:

```
protected function pruning()
{
 //
}
```

After configuring your prunable model, you should schedule the `model:prune` Artisan command in your application's `App\Console\Kernel` class. You are free to choose the appropriate interval at which this command should be run:

```
protected function schedule(Schedule $schedule)
{
 $schedule->command('model:prune')->daily();
}
```

Behind the scenes, the `model:prune` command will automatically detect "Prunable" models within your application's `app/Models` directory. If your models are in a different location, you may use the `--model` option to specify the model class names:

```
$schedule->command('model:prune', [
 '--model' => [Address::class, Flight::class],
])->daily();
```

# Module 7

Soft deleting models will be permanently deleted (`forceDelete`) if they match the prunable query.

## Mass Pruning

When models are marked with the `Illuminate\Database\Eloquent\MassPrunable` trait, models are deleted from the database using mass-deletion queries. Therefore, the pruning method will not be invoked, nor will the deleting and deleted model events be dispatched. This is because the models are never actually retrieved before deletion, thus making the pruning process much more efficient:

```
<?php

namespace App\Models;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
 use MassPrunable;
 public function prunable()
 {
 return static::where('created_at', '<=', now()->subMonth());
 }
}
```

# Module 7

## Replicating Models

You may create an unsaved copy of an existing model instance using the replicate method. This method is particularly useful when you have model instances that share many of the same attributes:

```
use App\Models\Address;

$shipping = Address::create([
 'type' => 'shipping',
 'line_1' => '123 Example Street',
 'city' => 'Victorville',
 'state' => 'CA',
 'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
 'type' => 'billing'
]);

$billing->save();
```

To exclude one or more attributes from being replicated to the new model, you may pass an array to the replicate method:

# Module 7

```
$flight = Flight::create([
 'destination' => 'LAX',
 'origin' => 'LHR',
 'last_flown' => '2020-03-04 11:00:00',
 'last_pilot_id' => 747,
]);

$flight = $flight->replicate([
 'last_flown',
 'last_pilot_id'
]);
```

## [Query Scopes](#)

## [Global Scopes](#)

Global scopes allow you to add constraints to all queries for a given model. Laravel's own [soft delete](#) functionality utilizes global scopes to only retrieve "non-deleted" models from the database. Writing your own global scopes can provide a convenient, easy way to make sure every query for a given model receives certain constraints.

## [Writing Global Scopes](#)

Writing a global scope is simple. First, define a class that implements the Illuminate\Database\Eloquent\Scope interface. Laravel does not have a conventional location that you should place scope classes, so you are free to place this class in any directory that you wish.

# Module 7

The Scope interface requires you to implement one method: `apply`. The `apply` method may add where constraints or other types of clauses to the query as needed:

```
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
 public function apply(Builder $builder, Model $model)
 {
 $builder->where('created_at', '<', now()->subYears(2000));
 }
}
```

If your global scope is adding columns to the select clause of the query, you should use the `addSelect` method instead of `select`. This will prevent the unintentional replacement of the query's existing select clause.

# Module 7

- **Applying Global Scopes**

To assign a global scope to a model, you should override the model's booted method and invoke the model's addGlobalScope method. The addGlobalScope method accepts an instance of your scope as its only argument:

```
<?php

namespace App\Models;

use App\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
protected static function booted()
{
 static::addGlobalScope(new AncientScope);
}
}
```

After adding the scope in the example above to the App\Models\User model, a call to the User::all() method will execute the following SQL query:

```
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

# Module 7

## Anonymous Global Scopes

Eloquent also allows you to define global scopes using closures, which is particularly useful for simple scopes that do not warrant a separate class of their own. When defining a global scope using a closure, you should provide a scope name of your own choosing as the first argument to the `addGlobalScope` method:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
 protected static function booted()
 {
 static::addGlobalScope('ancient', function (Builder $builder) {
 $builder->where('created_at', '<', now()->subYears(2000));
 });
 }
}
```

# Module 7

## Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. This method accepts the class name of the global scope as its only argument:

```
User::withoutGlobalScope(AncientScope::class)->get();
```

Or, if you defined the global scope using a closure, you should pass the string name that you assigned to the global scope:

```
User::withoutGlobalScope('ancient')->get();
```

If you would like to remove several or even all of the query's global scopes, you may use the `withoutGlobalScopes` method:

```
// Remove all of the global scopes...
User::withoutGlobalScopes()->get();
User::withoutGlobalScopes([
 FirstScope::class, SecondScope::class
])->get();
```

## Local Scopes

Local scopes allow you to define common sets of query constraints that you may easily re-use throughout your application. For example, you may need to frequently retrieve all users that are considered "popular". To define a scope, prefix an Eloquent model method with `scope`.

Scopes should always return a query builder instance:

# Module 7

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
 public function scopePopular($query)
 {
 return $query->where('votes', '>', 100);
 }
 public function scopeActive($query)
 {
 return $query->where('active', 1);
 }
}
```

## Utilizing A Local Scope

Once the scope has been defined, you may call the scope methods when querying the model. However, you should not include the scope prefix when calling the method. You can even chain calls to various scopes:

# Module 7

```
use App\Models\User;
$users = User::popular()->active()->orderBy('created_at')->get();
```

Combining multiple Eloquent model scopes via an or query operator may require the use of closures to achieve the correct [logical grouping](#):

```
$users = User::popular()->orWhere(function (Builder $query) {
 $query->active();
})->get();
```

However, since this can be cumbersome, Laravel provides a "higher order" orWhere method that allows you to fluently chain scopes together without the use of closures:

```
$users = App\Models\User::popular()->orWhere->active()->get();
```

## Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. To get started, just add your additional parameters to your scope method's signature. Scope parameters should be defined after the \$query parameter:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{
```

# Module 7

```
public function scopeOfType($query, $type)
{
 return $query->where('type', $type);
}
```

Once the expected arguments have been added to your scope method's signature, you may pass the arguments when calling the scope:

```
$users = User::ofType('admin')->get();
```

## Comparing Models

Sometimes you may need to determine if two models are the "same" or not. The `is` and `isNot` methods may be used to quickly verify two models have the same primary key, table, and database connection or not:

```
if ($post->is($anotherPost)) {
 //
}
```

```
if ($post->isNot($anotherPost)) {
 //
}
```

- The `is` and `isNot` methods are also available when using the `belongsTo`, `hasOne`, `morphTo`, and `morphOne` [relationships](#). This method is particularly helpful when you would like to compare a related model without issuing a query to retrieve that model:

# Module 7

```
if ($post->author()->is($user)) {
 //
}
```

## Events

Want to broadcast your Eloquent events directly to your client-side application? Check out Laravel's [model event broadcasting](#).

Eloquent models dispatch several events, allowing you to hook into the following moments in a model's lifecycle: retrieved, creating, created, updating, updated, saving, saved, deleting, deleted, restoring, restored, and replicating.

The retrieved event will dispatch when an existing model is retrieved from the database. When a new model is saved for the first time, the creating and created events will dispatch. The updating / updated events will dispatch when an existing model is modified and the save method is called. The saving / saved events will dispatch when a model is created or updated - even if the model's attributes have not been changed. Event names ending with -ing are dispatched before any changes to the model are persisted, while events ending with -ed are dispatched after the changes to the model are persisted.

To start listening to model events, define a \$dispatchesEvents property on your Eloquent model. This property maps various points of the Eloquent model's lifecycle to your own [event classes](#). Each model event class should expect to receive an instance of the affected model via its constructor:

# Module 7

```
<?php

namespace App\Models;

use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
 use Notifiable;
 protected $dispatchesEvents = [
 'saved' => UserSaved::class,
 'deleted' => UserDeleted::class,
];
}
```

After defining and mapping your Eloquent events, you may use [event listeners](#) to handle the events.

When issuing a mass update or delete query via Eloquent, the saved, updated, deleting, and deleted model events will not be dispatched for the affected models. This is because the models are never actually retrieved when performing mass updates or deletes.

# Module 7

## Using Closures

Instead of using custom event classes, you may register closures that execute when various model events are dispatched. Typically, you should register these closures in the booted method of your model:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{
 protected static function booted()
 {
 static::created(function ($user) {
 //
 });
 }
}
```

If needed, you may utilize [queueable anonymous event listeners](#) when registering model events. This will instruct Laravel to execute the model event listener in the background using your application's [queue](#):

# Module 7

```
use function Illuminate\Events\queueable;
static::created(queueable(function ($user) {
}));
```

## Observers

### Defining Observers

If you are listening for many events on a given model, you may use observers to group all of your listeners into a single class. Observer classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the affected model as their only argument. The `make:observer` Artisan command is the easiest way to create a new observer class:

```
php artisan make:observer UserObserver --model=User
```

This command will place the new observer in your `App/Observers` directory. If this directory does not exist, Artisan will create it for you. Your fresh observer will look like the following:

```
<?php
namespace App\Observers;
use App\Models\User;
class UserObserver
{
```

# Module 7

```
public function created(User $user)
{
 //
}
public function updated(User $user)
{
 //
}
public function deleted(User $user)
{
 //
}
public function forceDeleted(User $user)
{
 //
}
```

- To register an observer, you need to call the observe method on the model you wish to observe. You may register observers in the boot method of your application's App\Providers\EventServiceProvider service provider:

# Module 7

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Register any events for your application.
 *
 * @return void
 */
public function boot()
{
 User::observe(UserObserver::class);
}
```

There are additional events an observer can listen to, such as saving and retrieved. These events are described within the [events](#) documentation.

## Observers & Database Transactions

When models are being created within a database transaction, you may want to instruct an observer to only execute its event handlers after the database transaction is committed. You may accomplish this by defining an `$afterCommit` property on the observer. If a database transaction is not in progress, the event handlers will execute immediately:

# Module 7

```
namespace App\Observers;
```

```
use App\Models\User;
class UserObserver
{
 public $afterCommit = true;
}
```

```
//
```

```
}
```

```
}
```

- **Muting Events**

- You may occasionally need to temporarily "mute" all events fired by a model. You may achieve this using the `withoutEvents` method. The `withoutEvents` method accepts a closure as its only argument. Any code executed within this closure will not dispatch model events. For example, the following example will fetch and delete an `App\Models\User` instance without dispatching any model events. Any value returned by the closure will be returned by the `withoutEvents` method:

```
use App\Models\User;
$user = User::withoutEvents(function () use () {
 User::findOrFail(1)->delete();
 return User::find(2);
});
```

# Module 7

## Saving A Single Model Without Events

Sometimes you may wish to "save" a given model without dispatching any events. You may accomplish this using the `saveQuietly` method:

```
$user = User::findOrFail(1);
$user->name = 'Victoria Faith';
$user->saveQuietly();
```

## Eloquent: Relationships

### Introduction

Database tables are often related to one another. For example, a blog post may have many comments or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports a variety of common relationships

[One To One](#)  
[One To Many](#)  
[Many To Many](#)  
[Has One Through](#)  
[Has Many Through](#)  
[One To One \(Polymorphic\)](#)  
[One To Many \(Polymorphic\)](#)  
[Many To Many \(Polymorphic\)](#)

# Module 7

## Defining Relationships

Eloquent relationships are defined as methods on your Eloquent model classes. Since relationships also serve as powerful [query builders](#), defining relationships as methods provides powerful method chaining and querying capabilities. For example, we may chain additional query constraints on this posts relationship:

```
$user->posts()->where('active', 1)->get();
```

But, before diving too deep into using relationships, let's learn how to define each type of relationship supported by Eloquent.

### One To One

A one-to-one relationship is a very basic type of database relationship. For example, a User model might be associated with one Phone model. To define this relationship, we will place a phone method on the User model. The phone method should call the hasOne method and return its result. The hasOne method is available to your model via the model's Illuminate\Database\Eloquent\Model base class:

```
<?php
```

```
namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;
```

# Module 7

```
class User extends Model
{
 /**
 * Get the phone associated with the user.
 */
 public function phone()
 {
 return $this->hasOne(Phone::class);
 }
}
```

The first argument passed to the hasOne method is the name of the related model class. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship methods as if they were properties defined on the model:

```
$phone = User::find(1)->phone;
```

Eloquent determines the foreign key of the relationship based on the parent model name. In this case, the Phone model is automatically assumed to have a user\_id foreign key. If you wish to override this convention, you may pass a second argument to the hasOne method:

```
return $this->hasOne(Phone::class, 'foreign_key');
```

# Module 7

## Defining The Inverse Of The Relationship

So, we can access the Phone model from our User model. Next, let's define a relationship on the Phone model that will let us access the user that owns the phone. We can define the inverse of a hasOne relationship using the belongsTo method:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
 /**
 * Get the user that owns the phone.
 */
 public function user()
 {
 return $this->belongsTo(User::class);
 }
}
```

When invoking the user method, Eloquent will attempt to find a User model that has an id which matches the user\_id column on the Phone model.

# Module 7

- Eloquent determines the foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. So, in this case, Eloquent assumes that the Phone model has a `user_id` column. However, if the foreign key on the Phone model is not `user_id`, you may pass a custom key name as the second argument to the `belongsTo` method:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
 return $this->belongsTo(User::class, 'foreign_key');
}
```

- If the parent model does not use `id` as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the `belongsTo` method specifying the parent table's custom key:

```
public function user()
{
 return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

# Module 7

- **One To Many**
- A one-to-many relationship is used to define relationships where a single model is the parent to one or more child models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by defining a method on your Eloquent model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
 /**
 * Get the comments for the blog post.
 */
 public function comments()
 {
 return $this->hasMany(Comment::class);
 }
}
```

# Module 7

Remember, Eloquent will automatically determine the proper foreign key column for the Comment model. By convention, Eloquent will take the "snake case" name of the parent model and suffix it with `_id`. So, in this example, Eloquent will assume the foreign key column on the Comment model is `post_id`.

Once the relationship method has been defined, we can access the [collection](#) of related comments by accessing the `comments` property. Remember, since Eloquent provides "dynamic relationship properties", we can access relationship methods as if they were defined as properties on the model:

```
use App\Models\Post;

$comments = Post::find(1)->comments;
foreach ($comments as $comment) {
 //
}
```

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the `comments` method and continuing to chain conditions onto the query:

```
$comment = Post::find(1)->comments()
 ->where('title', 'foo')
 ->first();
```

Like the `hasOne` method, you may also override the foreign and local keys by passing additional arguments to the `hasMany` method:

# Module 7

```
return $this->hasMany(Comment::class, 'foreign_key');
return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

## One To Many (Inverse) / Belongs To

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a hasMany relationship, define a relationship method on the child model which calls the belongsTo method:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Comment extends Model
{
 public function post()
 {
 return $this->belongsTo(Post::class);
 }
}
```

Once the relationship has been defined, we can retrieve a comment's parent post by accessing the post "dynamic relationship property":

# Module 7

In the example above, Eloquent will attempt to find a Post model that has an id which matches the post\_id column on the Comment model.

Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with a \_ followed by the name of the parent model's primary key column. So, in this example, Eloquent will assume the Post model's foreign key on the comments table is post\_id.

However, if the foreign key for your relationship does not follow these conventions, you may pass a custom foreign key name as the second argument to the belongsTo method:

```
public function post()
{
 return $this->belongsTo(Post::class, 'foreign_key');
}
```

If your parent model does not use id as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the belongsTo method specifying your parent table's custom key:

```
public function post()
{
 return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

# Module 7

## Default Models

The belongsTo, hasOne, hasOneThrough, and morphOne relationships allow you to define a default model that will be returned if the given relationship is null. This pattern is often referred to as the [Null Object pattern](#) and can help remove conditional checks in your code. In the following example, the user relation will return an empty App\Models\User model if no user is attached to the Post model:

```
/**
 * Get the author of the post.
 */
public function user()
{
 return $this->belongsTo(User::class)->withDefault();
}
```

To populate the default model with attributes, you may pass an array or closure to the withDefault method:

```
/**
 * Get the author of the post.
 */
public function user()
{
 return $this->belongsTo(User::class)->withDefault([
```

# Module 7

```
'name' => 'Guest Author',
]);
}

/**
 * Get the author of the post.
 */
public function user()
{
 return $this->belongsTo(User::class)->withDefault(function ($user, $post) {
 $user->name = 'Guest Author';
 });
}
```

- **Has One Of Many**
- Sometimes a model may have many related models, yet you want to easily retrieve the "latest" or "oldest" related model of the relationship. For example, a User model may be related to many Order models, but you want to define a convenient way to interact with the most recent order the user has placed. You may accomplish this using the hasOne relationship type combined with the ofMany methods:

# Module 7

```
public function latestOrder()
{
 return $this->hasOne(Order::class)->latestOfMany();
}
```

Likewise, you may define a method to retrieve the "oldest", or first, related model of a relationship:

```
/**
 * Get the user's oldest order.
 */
public function oldestOrder()
{
 return $this->hasOne(Order::class)->oldestOfMany();
}
```

By default, the `latestOfMany` and `oldestOfMany` methods will retrieve the latest or oldest related model based on the model's primary key, which must be sortable. However, sometimes you may wish to retrieve a single model from a larger relationship using a different sorting criteria.

For example, using the `ofMany` method, you may retrieve the user's most expensive order. The `ofMany` method accepts the sortable column as its first argument and which aggregate function (min or max) to apply when querying for the related model:

# Module 7

```
public function largestOrder()
{
 return $this->hasOne(Order::class)->ofMany('price', 'max');
}
```

Because PostgreSQL does not support executing the MAX function against UUID columns, it is not currently possible to use one-of-many relationships in combination with PostgreSQL UUID columns.

## Advanced Has One Of Many Relationships

It is possible to construct more advanced "has one of many" relationships. For example, A Product model may have many associated Price models that are retained in the system even after new pricing is published. In addition, new pricing data for the product may be able to be published in advance to take effect at a future date via a published\_at column.

So, in summary, we need to retrieve the latest published pricing where the published date is not in the future. In addition, if two prices have the same published date, we will prefer the price with the greatest ID. To accomplish this, we must pass an array to the ofMany method that contains the sortable columns which determine the latest price. In addition, a closure will be provided as the second argument to the ofMany method. This closure will be responsible for adding additional publish date constraints to the relationship query:

# Module 7

```
public function currentPricing()
{
 return $this->hasOne(Price::class)->ofMany([
 'published_at' => 'max',
 'id' => 'max',
], function ($query) {
 $query->where('published_at', '<', now());
 });
}
```

## Has One Through

The "has-one-through" relationship defines a one-to-one relationship with another model. However, this relationship indicates that the declaring model can be matched with one instance of another model by proceeding *through* a third model.

For example, in a vehicle repair shop application, each Mechanic model may be associated with one Car model, and each Car model may be associated with one Owner model. While the mechanic and the owner have no direct relationship within the database, the mechanic can access the owner *through* the Car model. Let's look at the tables necessary to define this relationship:

```
mechanics
 id - integer
 name - string
```

# Module 7

```
cars
 id - integer
 model - string
 mechanic_id - integer
owners
 id - integer
 name - string
car_id - integer
```

Now that we have examined the table structure for the relationship, let's define the relationship on the Mechanic model:

```
<?php

namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Mechanic extends Model
{
 public function carOwner()
 {
 return $this->hasOneThrough(Owner::class, Car::class);
 }
}
```

# Module 7

The first argument passed to the `hasOneThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

## Key Conventions

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasOneThrough` method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model.

```
class Mechanic extends Model
{
 public function carOwner()
 {
 return $this->hasOneThrough(
 Owner::class,
 Car::class,
 'mechanic_id', // Foreign key on the cars table...
 'car_id', // Foreign key on the owners table...
 'id', // Local key on the mechanics table...
 'id' // Local key on the cars table...
);
 }
}
```

# Module 7

## Has Many Through

The "has-many-through" relationship provides a convenient way to access distant relations via an intermediate relation. For example, let's assume we are building a deployment platform like [Laravel Vapor](#). A Project model might access many Deployment models through an intermediate Environment model. Using this example, you could easily gather all deployments for a given project. Let's look at the tables required to define this relationship:

projects

  id - integer

  name - string

environments

  id - integer

  project\_id - integer

  name - string

deployments

  id - integer

  environment\_id - integer

  commit\_hash - string

Now that we have examined the table structure for the relationship, let's define the relationship on the Project model:

# Module 7

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
class Project extends Model
{
 /**
 * Get all of the deployments for the project.
 */
 public function deployments()
 {
 return $this->hasManyThrough(Deployment::class, Environment::class);
 }
}
```

- The first argument passed to the `hasManyThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.
- Though the `Deployment` model's table does not contain a `project_id` column, the `hasManyThrough` relation provides access to a project's deployments via `$project->deployments`. To retrieve these models, Eloquent inspects the `project_id` column on the intermediate `Environment` model's table. After finding the relevant environment IDs, they are used to query the `Deployment` model's table.

# Module 7

## Key Conventions

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasManyThrough` method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```
class Project extends Model
{
 public function deployments()
 {
 return $this->hasManyThrough(
 Deployment::class,
 Environment::class,
 'project_id', // Foreign key on the environments table...
 'environment_id', // Foreign key on the deployments table...
 'id', // Local key on the projects table...
 'id' // Local key on the environments table...
);
 }
}
```

# Module 7

## Many To Many Relationships

Many-to-many relations are slightly more complicated than hasOne and hasMany relationships. An example of a many-to-many relationship is a user that has many roles and those roles are also shared by other users in the application. For example, a user may be assigned the role of "Author" and "Editor"; however, those roles may also be assigned to other users as well. So, a user has many roles and a role has many users.

## Table Structure

To define this relationship, three database tables are needed: users, roles, and role\_user. The role\_user table is derived from the alphabetical order of the related model names and contains user\_id and role\_id columns. This table is used as an intermediate table linking the users and roles.

Remember, since a role can belong to many users, we cannot simply place a user\_id column on the roles table. This would mean that a role could only belong to a single user. In order to provide support for roles being assigned to multiple users, the role\_user table is needed. We can summarize the relationship's table structure like so:

users

id - integer

name - string

# Module 7

roles

id - integer

name - string

role\_user

user\_id - integer

role\_id - integer

## Model Structure

Many-to-many relationships are defined by writing a method that returns the result of the belongsToMany method. The belongsToMany method is provided by the Illuminate\Database\Eloquent\Model base class that is used by all of your application's Eloquent models. For example, let's define a roles method on our User model. The first argument passed to this method is the name of the related model class:

```
<?php
```

```
namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model
```

```
{
```

# Module 7

```
public function roles()
{
 return $this->belongsToMany(Role::class);
}
```

- Once the relationship is defined, you may access the user's roles using the roles dynamic relationship property:

```
use App\Models\User;
```

```
$user = User::find(1);
```

```
foreach ($user->roles as $role) {
 //
}
```

- Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the roles method and continuing to chain conditions onto the query:

```
$roles = User::find(1)->roles()->orderBy('name')->get();
```

# Module 7

To determine the table name of the relationship's intermediate table, Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the belongsToMany method:

```
return $this->belongsToMany(Role::class, 'role_user');
```

In addition to customizing the name of the intermediate table, you may also customize the column names of the keys on the table by passing additional arguments to the belongsToMany method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to:

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

## **Laravel Socialite**Introduction

In addition to typical, form based authentication, Laravel also provides a simple, convenient way to authenticate with OAuth providers using [Laravel Socialite](#). Socialite currently supports authentication with Facebook, Twitter, LinkedIn, Google, GitHub, GitLab, and Bitbucket.

Adapters for other platforms are listed at the community driven [Socialite Providers](#) website.

## Installation

To get started with Socialite, use the Composer package manager to add the package to your project's dependencies:

```
composer require laravel/socialite
```

# Module 7

## Upgrading Socialite

When upgrading to a new major version of Socialite, it's important that you carefully review [the upgrade guide](#).

## Configuration

Before using Socialite, you will need to add credentials for the OAuth providers your application utilizes. These credentials should be placed in your application's config/services.php configuration file, and should use the key facebook, twitter, linkedin, google, github, gitlab, or bitbucket, depending on the providers your application requires:

```
'github' => [
 'client_id' => env('GITHUB_CLIENT_ID'),
 'client_secret' => env('GITHUB_CLIENT_SECRET'),
 'redirect' => 'http://example.com/callback-url',
,
```

If the redirect option contains a relative path, it will automatically be resolved to a fully qualified URL.

# Module 7

## AuthenticationRouting

To authenticate users using an OAuth provider, you will need two routes: one for redirecting the user to the OAuth provider, and another for receiving the callback from the provider after authentication. The example controller below demonstrates the implementation of both routes:

```
use Laravel\\Socialite\\Facades\\Socialite;

Route::get('/auth/redirect', function () {
 return Socialite::driver('github')->redirect();
});

Route::get('/auth/callback', function () {
 $user = Socialite::driver('github')->user();

 // $user->token
});
```

The redirect method provided by the Socialite facade takes care of redirecting the user to the OAuth provider, while the user method will read the incoming request and retrieve the user's information from the provider after they are authenticated.

# Module 7

## Optional Parameters

A number of OAuth providers support optional parameters in the redirect request. To include any optional parameters in the request, call the with method with an associative array:

```
use Laravel\\Socialite\\Facades\\Socialite;

return Socialite::driver('google')
 ->with(['hd' => 'example.com'])
 ->redirect();
```

When using the with method, be careful not to pass any reserved keywords such as state or response\_type.

## Access Scopes

Before redirecting the user, you may also add additional "scopes" to the authentication request using the scopes method. This method will merge all existing scopes with the scopes that you supply:

```
use Laravel\\Socialite\\Facades\\Socialite;

return Socialite::driver('github')
 ->scopes(['read:user', 'public_repo'])
 ->redirect();
```

# Module 7

You can overwrite all existing scopes on the authentication request using the `setScopes` method:

```
return Socialite::driver('github')
 ->setScopes(['read:user', 'public_repo'])
 ->redirect();
```

## Retrieving User Details

After the user is redirected back to your authentication callback route, you may retrieve the user's details using Socialite's `user` method. The `user` object returned by the `user` method provides a variety of properties and methods you may use to store information about the user in your own database. Different properties and methods may be available depending on whether the OAuth provider you are authenticating with supports OAuth 1.0 or OAuth 2.0:

```
Route::get('/auth/callback', function () {
 $user = Socialite::driver('github')->user();
 $token = $user->token;
 $refreshToken = $user->refreshToken;
 $expiresIn = $user->expiresIn;

 // OAuth 1.0 providers...
 $token = $user->token;
 $tokenSecret = $user->tokenSecret;
```

```
// All providers...
$user->getId();
$user->getNickname();
$user->getName();
$user->getEmail();
$user->getAvatar();
});
```

- **Retrieving User Details From A Token (OAuth2)**

- If you already have a valid access token for a user, you can retrieve their details using Socialite's `userFromToken` method:

```
use Laravel\\Socialite\\Facades\\Socialite;
```

```
$user = Socialite::driver('github')->userFromToken($token);
```

- **Retrieving User Details From A Token And Secret (OAuth1)**

- If you already have a valid token and secret for a user, you can retrieve their details using Socialite's `userFromTokenAndSecret` method:

```
use Laravel\\Socialite\\Facades\\Socialite;
```

```
$user = Socialite::driver('twitter')->userFromTokenAndSecret($token, $secret);
```

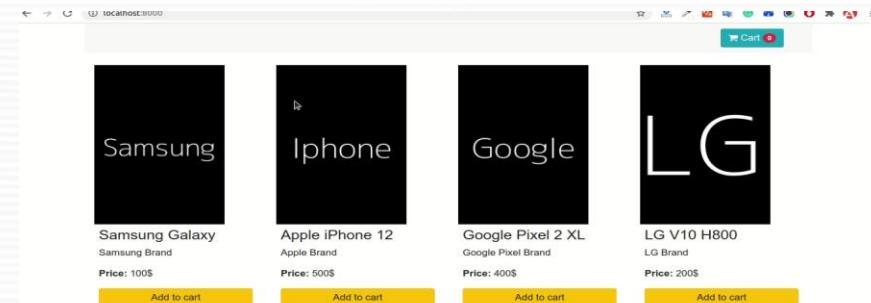
## Stateless Authentication

The stateless method may be used to disable session state verification. This is useful when adding social authentication to an API:

```
use Laravel\\Socialite\\Facades\\Socialite;
```

```
return Socialite::driver('google')->stateless()->user();
```

## Laravel Shopping Add to Cart with Ajax



## **Step 1: Install Laravel**

first of all we need to get fresh Laravel 8 version application using bellow command, So open your terminal OR command prompt and run bellow command:

```
composer create-project --prefer-dist
laravel/laravel blog
```

## **Step 2: Create Table, Model and Seeder**

In this step, we need to create products table, model and add some dummy records with seeder.

### **Create Migration**

```
php artisan make:migration
create_products_table
```

### **Migration**

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
class CreateProductsTable extends Migration
{
 public function up()
 {
 Schema::create('products', function (Blueprint
$table) {
```

```
$table->id();
 $table->string("name", 255)->nullable();
 $table->text("description")->nullable();
 $table->string("image", 255)->nullable();
 $table->decimal("price", 6, 2);
 $table->timestamps();
});
}
public function down()
{
 Schema::dropIfExists('products');
}
}
```

Run migration now:

```
php artisan migrate
```

## Create Model

run bellow command to create Product model:

```
php artisan make:model Product
```

# Module 7

```
app/Models/Products.php
<?php
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Product extends Model
{
 use HasFactory;
 protected $fillable =
 ['name', 'price', 'description', 'image'
];
}
```

## Create Seeder

run bellow command to create Product seeder:

```
php artisan make:seed ProductSeeder
```

# Module 7

```
database/seeders/ProductSeeder.php
namespace Database\Seeders;
use Illuminate\Database\Seeder;
use App\Models\Product;
class ProductSeeder extends Seeder
{
 public function run()
 {
 $products = [
 [
 'name' => 'Samsung Galaxy',
 'description' => 'Samsung Brand',
 'image' => 'https://dummyimage.com/200x300/000/fff&text=Samsung',
 'price' => 100
],
 [
 'name' => 'Apple iPhone 12',
 'description' => 'Apple Brand',
 'image' =>
 'https://dummyimage.com/200x300/000/fff&text=Iphone',
 'price' => 500
],
 [
 'name' => 'Google Pixel 2 XL',
 'description' => 'Google Pixel Brand',
 'image' =>
 'https://dummyimage.com/200x300/000/fff&text=Google',
 'price' => 400
]
];
 }
}
```

# Module 7

```
],
[
 'name' => 'LG V10 H800',
 'description' => 'LG Brand',
 'image' =>
 'https://dummyimage.com/200x300/000/fff&text=LG',
 'price' => 200
]
];
foreach ($products as $key => $value) {
 Product::create($value);
}
}
}
```

## Run seeder now:

```
php artisan db:seed --class=ProductSeeder
```

## Step 3: Create Route

In this step we need to create some routes for add to cart function.

routes/web.php

# Module 7

```
<?php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\ProductController;
/*
• | -----
• | Web Routes
• | -----
• |
• | Here is where you can register web routes for your application. These
• | routes are loaded by the RouteServiceProvider within a group which
• | contains the "web" middleware group. Now create something great!
• |
*/
Route::get('/', [ProductController::class, 'index']);
Route::get('cart', [ProductController::class, 'cart'])->name('cart');
Route::get('add-to-cart/{id}', [ProductController::class, 'addToCart'])->name('add.to.cart');
Route::patch('update-cart', [ProductController::class, 'update'])->name('update.cart');
Route::delete('remove-from-cart', [ProductController::class, 'remove'])->name('remove.from.cart');
```

# Module 7

## Step 4: Create Controller

in this step, we need to create ProductController and add following code on that file:

```
app/Http/Controllers/ProductController.php
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Models\Product;
class ProductController extends Controller
{
 /**
 * Write code on Method
 *
 * @return response()
 */
 public function index()
 {
 $products = Product::all();
 return view('products', compact('products'));
 }
}
```

# Module 7

```
public function cart()
{
 return view('cart');
}

public function addToCart($id)
{
 $product = Product::findOrFail($id);
 $cart = session()->get('cart', []);
 if(isset($cart[$id])) {
 $cart[$id]['quantity]++;
 } else {
 $cart[$id] = [
 "name" => $product->name,
 "quantity" => 1,
 "price" => $product->price,
 "image" => $product->image
];
 }

 session()->put('cart', $cart);
 return redirect()->back()->with('success', 'Product added to cart successfully!');
}
```

# Module 7

```
public function update(Request $request)
{
 if($request->id && $request->quantity){
 $cart = session()->get('cart');
 $cart[$request->id]["quantity"] = $request->quantity;
 session()->put('cart', $cart);
 session()->flash('success', 'Cart updated successfully');
 }
}
/**
 * Write code on Method
 *
 * @return response()
 */
public function remove(Request $request)
{
 if($request->id) {
 $cart = session()->get('cart');
 if(isset($cart[$request->id])) {
 unset($cart[$request->id]);
 session()->put('cart', $cart);
 }
 }
}
```

# Module 7

```
 session()->flash('success', 'Product removed successfully');
}
}
}
```

- **Step 5: Create Blade Files**
- here, we need to create blade files for layout, products and cart page. so let's create one by one files:

resources/views/layout.blade.php

```
<!DOCTYPE html>
<html>
<head>
 <title>Laravel Add To Cart Function - ItSolutionStuff.com</title>
 <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
 <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.2/dist/umd/popper.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>
<link href="{{ asset('css/style.css') }}" rel="stylesheet"></head>
```

# Module 7

```
<body>
<div class="container">
<div class="row">
 <div class="col-lg-12 col-sm-12 col-12 main-section">
 <div class="dropdown">
 <button type="button" class="btn btn-info" data-toggle="dropdown">
 <i class="fa fa-shopping-cart" aria-hidden="true"></i> Cart {{ count((array) session('cart')) }}
 </button>
 <div class="dropdown-menu">
 <div class="row total-header-section">
 <div class="col-lg-6 col-sm-6 col-6">
 <i class="fa fa-shopping-cart" aria-hidden="true"></i> {{ count((array) session('cart')) }}
 </div>
 </div>
 @php $total = 0 @endphp
 @foreach((array) session('cart') as $id => $details)
 @php $total += $details['price'] * $details['quantity'] @endphp
 @endforeach
 <div class="col-lg-6 col-sm-6 col-6 total-section text-right">
 <p>Total : $ {{ $total }}</p>
 </div>
 </div>
</div>
```

# Module 7

```
</div>
</div>
@if(session('cart'))
@foreach(session('cart') as $id => $details)
<div class="row cart-detail">
<div class="col-lg-4 col-sm-4 col-4 cart-detail-img">

</div>
<div class="col-lg-8 col-sm-8 col-8 cart-detail-product">
<p>{{ $details['name'] }}</p>
 ${{ $details['price'] }} Quantity:{{ $details['quantity'] }}
</div>
</div>
@endforeach
@endif
<div class="row">
<div class="col-lg-12 col-sm-12 col-12 text-center checkout">
View all
</div>
</div>
</div>
</div>


```

# Module 7

```
<div class="container">
 @if(session('success'))
 <div class="alert alert-success">
 {{ session('success') }}
 </div>
 @endif
 @yield('content')
</div>
@yield('scripts')

</body>
</html>
```

- **resources/views/products.blade.php**

```
@extends('layout')
@section('content')

<div class="row">
 @foreach($products as $product)
 <div class="col-xs-18 col-sm-6 col-md-3">
 <div class="thumbnail">

 <div class="caption">
```

# Module 7

```
<div class="caption">
 <h4>{{ $product->name }}</h4>
 <p>{{ $product->description }}</p>
 <p>Price: {{ $product->price }}$</p>
 <p class="btn-holder">id) }}" class="btn btn-warning btn-block text-center" role="button">Add to cart </p>
 </div>
 </div>
</div>
@endforeach
</div>
@endsection
```

- **resources/views/cart.blade.php**

```
@extends('layout')
@section('content')
<table id="cart" class="table table-hover table-condensed">
 <thead>
 <tr>
```

# Module 7

```
<th style="width:50%">Product</th>
 <th style="width:10%">Price</th>
 <th style="width:8%">Quantity</th>
 <th style="width:22%" class="text-center">Subtotal</th>
 <th style="width:10%"></th>
</tr>
</thead>
<tbody>
 @php $total = 0 @endphp
 @if(session('cart'))
 @foreach(session('cart') as $id => $details)
 @php $total += $details['price'] * $details['quantity'] @endphp
<tr data-id="{{ $id }}>
 <td data-th="Product">
 <div class="row">
 <div class="col-sm-3 hidden-xs"></div>
 <div class="col-sm-9">
 <h4 class="nomargin">{{ $details['name'] }}</h4>
 </div>
 </div>
 </td>
```

# Module 7

```
</td>

<td data-th="Price">${{ $details['price'] }}</td>

<td data-th="Quantity">
 <input type="number" value="{{ $details['quantity'] }}" class="form-control quantity update-cart" />
</td>

<td data-th="Subtotal" class="text-center">${{ $details['price'] * $details['quantity'] }}</td>

<td class="actions" data-th="">
 <button class="btn btn-danger btn-sm remove-from-cart"><i class="fa fa-trash-o"></i></button>
</td>
</tr>

@endforeach

@endif

</tbody>

<tfoot>

<tr>
 <td colspan="5" class="text-right"><h3>Total ${{ $total }}</h3></td>
</tr>
<tr>
 <td colspan="5" class="text-right">
 <i class="fa fa-angle-left"></i> Continue Shopping
 <button class="btn btn-success">Checkout</button>
 </td>
</tr>
```

# Module 7

```
</tfoot>
</table>
@endsection
@section('scripts')
<script type="text/javascript">
 $(".update-cart").change(function (e) {
 e.preventDefault();
 var ele = $(this);
 $.ajax({
 url: '{{ route('update.cart') }}',
 method: "patch",
 data: {
 _token: '{{ csrf_token() }}',
 id: ele.parents("tr").attr("data-id"),
 quantity: ele.parents("tr").find(".quantity").val()
 },
 success: function (response) {
 window.location.reload();
 }
 });
 });
}</script>
```

# Module 7

```
$(".remove-from-cart").click(function (e) {
 e.preventDefault();
 var ele = $(this);
 if(confirm("Are you sure want to remove?")) {
 $.ajax({
 url: '{{ route('remove.from.cart') }}',
 method: "DELETE",
 data: {
 _token: '{{ csrf_token() }}',
 id: ele.parents("tr").attr("data-id")
 },
 success: function (response) {
 window.location.reload();
 }
 });
 }
});
</script>
@endsection
```

# Module 7

```
public/css/style.css
.thumbnail {
 position: relative;
 padding: 0px;
 margin-bottom: 20px;
}
.thumbnail img {
 width: 80%;
}
.thumbnail .caption{
 margin: 7px;
}
.main-section{
 background-color: #F8F8F8;
}
.dropdown{
 float:right;
 padding-right: 30px;
}
.btn{
 border:0px;
 margin:10px 0px;
 box-shadow:none !important;
}
.dropdown .dropdown-menu{
```

# Module 7

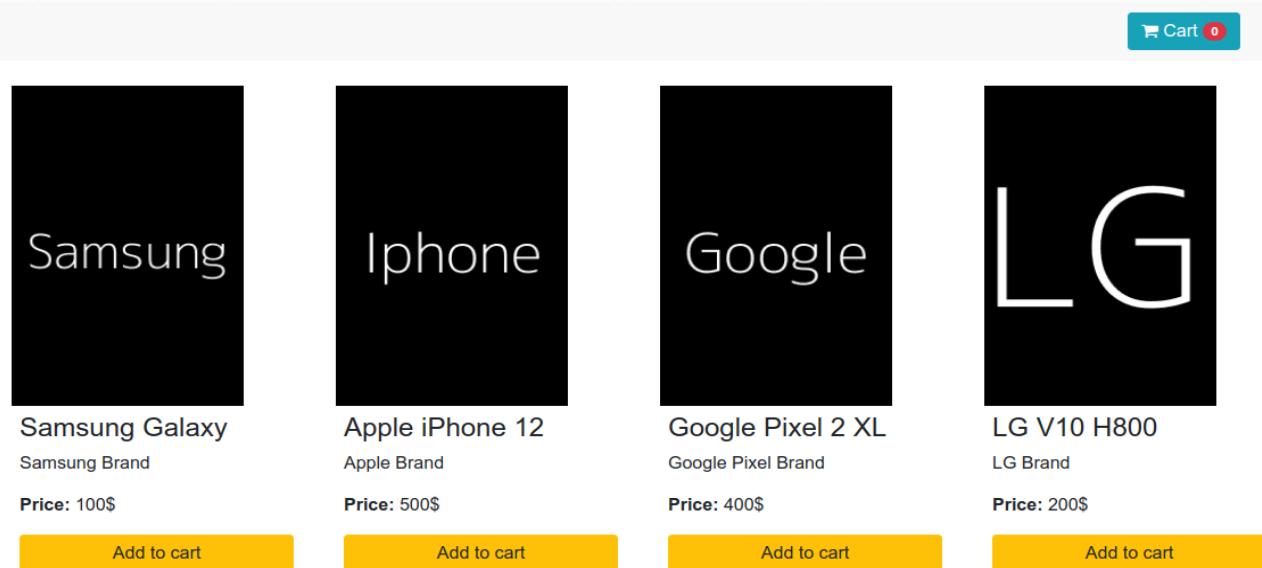
```
padding:20px;
 top:30px !important;
 width:350px !important;
 left:-110px !important;
 box-shadow:0px 5px 30px black;
}
.total-header-section{
 border-bottom:1px solid #d2d2d2;
}
.total-section p{
 margin-bottom:20px;
}
.cart-detail{
 padding:15px 0px;
}
.cart-detail-img img{
 width:100%;
 height:100%;
 padding-left:15px;
}
.cart-detail-product p{
 margin:0px;
 color:#000;
 font-weight:500;
}
```

# Module 7

```
.cart-detail .price{
 font-size:12px;
 margin-right:10px;
 font-weight:500;
}
.cart-detail .count{
 color:#C2C2DC;
}
.checkout{
 border-top:1px solid #d2d2d2;
 padding-top: 15px;
}
.checkout .btn-primary{
 border-radius:50px;
 height:50px;
}
.dropdown-menu:before{
 content: " ";
 position:absolute;
 top:-20px;
 right:50px;
 border:10px solid transparent;
 border-bottom-color:#fff;
}
```

# Module 7

- Now we are ready to run our example so run bellow command so quick run:
  - `php artisan serve`
- Now you can open bellow URL on your browser:
  - **localhost:8000/Products**



# Module 7

## Cart

Cart updated successfully

| Product                                                                                             | Price | Quantity                       | Subtotal |
|-----------------------------------------------------------------------------------------------------|-------|--------------------------------|----------|
|  Apple iPhone 12   | \$500 | <input type="text" value="2"/> | \$1000   |
|  Google Pixel 2 XL | \$400 | <input type="text" value="1"/> | \$400    |

Total \$1400

[Continue Shopping](#) [Checkout](#)

# Module 7

- **Security & Session**
- **HTTP Session**
- [Introduction](#)
- Since HTTP driven applications are stateless, sessions provide a way to store information about the user across multiple requests. That user information is typically placed in a persistent store / backend that can be accessed from subsequent requests.
- Laravel ships with a variety of session backends that are accessed through an expressive, unified API. Support for popular backends such as [Memcached](#), [Redis](#), and databases is included.
- [Configuration](#)
- Your application's session configuration file is stored at config/session.php. Be sure to review the options available to you in this file. By default, Laravel is configured to use the file session driver, which will work well for many applications. If your application will be load balanced across multiple web servers, you should choose a centralized store that all servers can access, such as Redis or a database.
- The session driver configuration option defines where session data will be stored for each request. Laravel ships with several great drivers out of the box:
  - file - sessions are stored in storage/framework/sessions.
  - cookie - sessions are stored in secure, encrypted cookies.
  - database - sessions are stored in a relational database.
  - memcached / redis - sessions are stored in one of these fast, cache based stores.
  - dynamodb - sessions are stored in AWS DynamoDB.
  - array - sessions are stored in a PHP array and will not be persisted.
- The array driver is primarily used during [testing](#) and prevents the data stored in the session from being persisted.

# Module 7

- [Driver Prerequisites](#)
- [Database](#)
- When using the database session driver, you will need to create a table to contain the session records. An example Schema declaration for the table may be found below:

```
Schema::create('sessions', function ($table) {
 $table->string('id')->primary();
 $table->foreignId('user_id')->nullable()->index();
 $table->string('ip_address', 45)->nullable();
 $table->text('user_agent')->nullable();
 $table->text('payload');
 $table->integer('last_activity')->index();
});
```
- You may use the `session:table` Artisan command to generate this migration. To learn more about database migrations, you may consult the complete [migration documentation](#):
  - `php artisan session:table`
  - `php artisan migrate`
- [Redis](#)
- Before using Redis sessions with Laravel, you will need to either install the `PhpRedis` PHP extension via PECL or install the `predis/predis` package (~1.0) via Composer. For more information on configuring Redis, consult Laravel's [Redis documentation](#).
- In the session configuration file, the `connection` option may be used to specify which Redis connection is used by the session.

# Module 7

- [Interacting With The Session](#)
- [Retrieving Data](#)
- There are two primary ways of working with session data in Laravel: the global session helper and via a Request instance. First, let's look at accessing the session via a Request instance, which can be type-hinted on a route closure or controller method. Remember, controller method dependencies are automatically injected via the Laravel [service container](#):

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class UserController extends Controller
{
 public function show(Request $request, $id)
 {
 $value = $request->session()->get('key');
 }
}
```

- When you retrieve an item from the session, you may also pass a default value as the second argument to the get method. This default value will be returned if the specified key does not exist in the session. If you pass a closure as the default value to the get method and the requested key does not exist, the closure will be executed and its result returned:

# Module 7

```
$value = $request->session()->get('key', 'default');
```

```
$value = $request->session()->get('key', function () {
 return 'default';
});
```

- [The Global Session Helper](#)
- You may also use the global session PHP function to retrieve and store data in the session. When the session helper is called with a single, string argument, it will return the value of that session key. When the helper is called with an array of key / value pairs, those values will be stored in the session:

```
Route::get('/home', function () {
 // Retrieve a piece of data from the session...
 $value = session('key');

 // Specifying a default value...
 $value = session('key', 'default');

 // Store a piece of data in the session...
 session(['key' => 'value']);
});
```
- There is little practical difference between using the session via an HTTP request instance versus using the global session helper. Both methods are [testable](#) via the assertSessionHas method which is available in all of your test cases.

# Module 7

- Retrieving All Session Data
  - If you would like to retrieve all the data in the session, you may use the all method:

```
$data = $request->session()->all();
```
- Determining If An Item Exists In The Session
  - To determine if an item is present in the session, you may use the has method. The has method returns true if the item is present and is not null:

```
if ($request->session()->has('users')) {
 //
}
```
  - To determine if an item is present in the session, even if its value is null, you may use the exists method:

```
if ($request->session()->exists('users')) {
 //
}
```
  - To determine if an item is not present in the session, you may use the missing method. The missing method returns true if the item is null or if the item is not present:

```
if ($request->session()->missing('users')) {
 //
}
```
- Storing Data
  - To store data in the session, you will typically use the request instance's put method or the global session helper:

# Module 7

```
// Via a request instance...
$request->session()->put('key', 'value');

// Via the global "session" helper...
session(['key' => 'value']);
```

- [\*\*Pushing To Array Session Values\*\*](#)
  - The push method may be used to push a new value onto a session value that is an array. For example, if the user.teams key contains an array of team names, you may push a new value onto the array like so:

```
$request->session()->push('user.teams', 'developers');
```
- [\*\*Retrieving & Deleting An Item\*\*](#)
  - The pull method will retrieve and delete an item from the session in a single statement:

```
$value = $request->session()->pull('key', 'default');
```
- [\*\*Incrementing & Decrementing Session Values\*\*](#)
  - If your session data contains an integer you wish to increment or decrement, you may use the increment and decrement methods:

```
$request->session()->increment('count');

$request->session()->increment('count', $incrementBy = 2);

$request->session()->decrement('count');

$request->session()->decrement('count', $decrementBy = 2);
```

# Module 7

- **Flash Data**
- Sometimes you may wish to store items in the session for the next request. You may do so using the flash method. Data stored in the session using this method will be available immediately and during the subsequent HTTP request. After the subsequent HTTP request, the flashed data will be deleted. Flash data is primarily useful for short-lived status messages:
  - `$request->session()->flash('status', 'Task was successful!');`
  - If you need to persist your flash data for several requests, you may use the reflash method, which will keep all of the flash data for an additional request. If you only need to keep specific flash data, you may use the keep method:
    - `$request->session()->reflash();`
    - `$request->session()->keep(['username', 'email']);`
  - To persist your flash data only for the current request, you may use the now method:
    - `$request->session()->now('status', 'Task was successful!');`
- **Deleting Data**
- The forget method will remove a piece of data from the session. If you would like to remove all data from the session, you may use the flush method:

```
// Forget a single key...
$request->session()->forget('name');

// Forget multiple keys...
$request->session()->forget(['name', 'status']);

$request->session()->flush();
```

# Module 7

- [Regenerating The Session ID](#)
  - Regenerating the session ID is often done in order to prevent malicious users from exploiting a [session fixation](#) attack on your application.
  - Laravel automatically regenerates the session ID during authentication if you are using one of the Laravel [application starter kits](#) or [Laravel Fortify](#); however, if you need to manually regenerate the session ID, you may use the regenerate method:

```
$request->session()->regenerate();
```
  - If you need to regenerate the session ID and remove all data from the session in a single statement, you may use the invalidate method:

```
$request->session()->invalidate();
```
- [Session Blocking](#)
  - To utilize session blocking, your application must be using a cache driver that supports [atomic locks](#). Currently, those cache drivers include the memcached, dynamodb, redis, and database drivers. In addition, you may not use the cookie session driver.
  - By default, Laravel allows requests using the same session to execute concurrently. So, for example, if you use a JavaScript HTTP library to make two HTTP requests to your application, they will both execute at the same time. For many applications, this is not a problem; however, session data loss can occur in a small subset of applications that make concurrent requests to two different application endpoints which both write data to the session.
  - To mitigate this, Laravel provides functionality that allows you to limit concurrent requests for a given session. To get started, you may simply chain the block method onto your route definition. In this example, an incoming request to the /profile endpoint would acquire a session lock. While this lock is being held, any incoming requests to the /profile or /order endpoints which share the same session ID will wait for the first request to finish executing before continuing their execution:

# Module 7

```
Route::post('/profile', function () {
 //
})->block($lockSeconds = 10, $waitSeconds = 10)
```

```
Route::post('/order', function () {
 //
})->block($lockSeconds = 10, $waitSeconds = 10)
```

- The block method accepts two optional arguments. The first argument accepted by the block method is the maximum number of seconds the session lock should be held for before it is released. Of course, if the request finishes executing before this time the lock will be released earlier.
- The second argument accepted by the block method is the number of seconds a request should wait while attempting to obtain a session lock. An Illuminate\Contracts\Cache\LockTimeoutException will be thrown if the request is unable to obtain a session lock within the given number of seconds.
- If neither of these arguments is passed, the lock will be obtained for a maximum of 10 seconds and requests will wait a maximum of 10 seconds while attempting to obtain a lock:

```
Route::post('/profile', function () {
 //
})->block()
```

# Module 7

- [Adding Custom Session Drivers](#)
- [Implementing The Driver](#)
- If none of the existing session drivers fit your application's needs, Laravel makes it possible to write your own session handler. Your custom session driver should implement PHP's built-in `SessionHandlerInterface`. This interface contains just a few simple methods. A stubbed MongoDB implementation looks like the following:

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface
{
 public function open($savePath, $sessionId) {}
 public function close() {}
 public function read($sessionId) {}
 public function write($sessionId, $data) {}
 public function destroy($sessionId) {}
 public function gc($lifetime) {}
}
```

- Laravel does not ship with a directory to contain your extensions. You are free to place them anywhere you like. In this example, we have created an `Extensions` directory to house the `MongoSessionHandler`.
- Since the purpose of these methods is not readily understandable, let's quickly cover what each of the methods do:

# Module 7

- The open method would typically be used in file based session store systems. Since Laravel ships with a file session driver, you will rarely need to put anything in this method. You can simply leave this method empty.
- The close method, like the open method, can also usually be disregarded. For most drivers, it is not needed.
- The read method should return the string version of the session data associated with the given \$sessionId. There is no need to do any serialization or other encoding when retrieving or storing session data in your driver, as Laravel will perform the serialization for you.
- The write method should write the given \$data string associated with the \$sessionId to some persistent storage system, such as MongoDB or another storage system of your choice. Again, you should not perform any serialization - Laravel will have already handled that for you.
- The destroy method should remove the data associated with the \$sessionId from persistent storage.
- The gc method should destroy all session data that is older than the given \$lifetime, which is a UNIX timestamp. For self-expiring systems like Memcached and Redis, this method may be left empty.

## Registering The Driver

- Once your driver has been implemented, you are ready to register it with Laravel. To add additional drivers to Laravel's session backend, you may use the extend method provided by the [Session facade](#). You should call the extend method from the boot method of a [service provider](#). You may do this from the existing App\Providers\AppServiceProvider or create an entirely new provider:

```
<?php
namespace App\Providers;

use App\Extensions\MongoSessionHandler;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;
```

# Module 7

```
class SessionServiceProvider extends ServiceProvider
{
 /**
 * Register any application services.
 *
 * @return void
 */
 public function register()
 {
 //
 }

 public function boot()
 {
 Session::extend('mongo', function ($app) {
 // Return an implementation of SessionHandlerInterface...
 return new MongoSessionHandler();
 });
 }
}
```

- Once the session driver has been registered, you may use the mongo driver in your config/session.php configuration file.

# Module 7

## Security

### Configuration

Laravel aims to make implementing authentication very simple. In fact, almost everything is configured for you out of the box. The authentication configuration file is located at `app/config/auth.php`, which contains several well-documented options for tweaking the behavior of the authentication facilities.

By default, Laravel includes a User model in your `app/models` directory which may be used with the default Eloquent authentication driver. Please remember when building the Schema for this Model to ensure that the password field is a minimum of 60 characters.

If your application is not using Eloquent, you may use the database authentication driver which uses the Laravel query builder.

Note: Before getting started, make sure that your users (or equivalent) table contains a nullable, string `remember_token` column of 100 characters. This column will be used to store a token for "remember me" sessions being maintained by your application. This can be done by using `$table->rememberToken();` in a migration.

### Storing Passwords

The Laravel Hash class provides secure Bcrypt hashing:

#### **Hashing A Password Using Bcrypt**

```
$password = Hash::make('secret');
```

#### **Verifying A Password Against A Hash**

```
if (Hash::check('secret', $hashedPassword))
{
 // The passwords match...
}
```

# Module 7

## Checking If A Password Needs To Be Rehashed

```
if (Hash::needsRehash($hashed))
{
 $hashed = Hash::make('secret');
}
```

## Authenticating Users

To log a user into your application, you may use the Auth::attempt method.

```
if (Auth::attempt(array('email' => $email, 'password' => $password)))
{
 return Redirect::intended('dashboard');
}
```

Take note that email is not a required option, it is merely used for example. You should use whatever column name corresponds to a "username" in your database. The Redirect::intended function will redirect the user to the URL they were trying to access before being caught by the authentication filter. A fallback URI may be given to this method in case the intended destination is not available.

When the attempt method is called, the auth.attempt [event](#) will be fired. If the authentication attempt is successful and the user is logged in, the auth.login event will be fired as well.

## Determining If A User Is Authenticated

To determine if the user is already logged into your application, you may use the check method:

```
if (Auth::check())
{
 // The user is logged in...
}
```

# Module 7

## Authenticating A User And "Remembering" Them

If you would like to provide "remember me" functionality in your application, you may pass true as the second argument to the attempt method, which will keep the user authenticated indefinitely (or until they manually logout). Of course, your users table must include the string remember\_token column, which will be used to store the "remember me" token.

```
if (Auth::attempt(array('email' => $email, 'password' => $password), true))
{
 // The user is being remembered...
}
```

Note: If the attempt method returns true, the user is considered logged into the application.

## Determining If User Authed Via Remember

If you are "remembering" user logins, you may use the viaRemember method to determine if the user was authenticated using the "remember me" cookie:

```
if (Auth::viaRemember())
{
 //
}
```

## Authenticating A User With Conditions

```
if (Auth::attempt(array('email' => $email, 'password' => $password, 'active' => 1)))
{
 // The user is active, not suspended, and exists.
}
```

Note: For added protection against session fixation, the user's session ID will automatically be regenerated after authenticating.

# Module 7

## Accessing The Logged In User

Once a user is authenticated, you may access the User model / record:

```
$email = Auth::user()->email;
```

To retrieve the authenticated user's ID, you may use the id method:

```
$id = Auth::id();
```

To simply log a user into the application by their ID, use the loginUsingId method:

```
Auth::loginUsingId(1);
```

## Validating User Credentials Without Login

The validate method allows you to validate a user's credentials without actually logging them into the application:

```
if (Auth::validate($credentials))
{
 //
}
```

## Logging A User In For A Single Request

You may also use the once method to log a user into the application for a single request. No sessions or cookies will be utilized.

```
if (Auth::once($credentials))
{
}
```

## Logging A User Out Of The Application

```
Auth::logout();
```

# Module 7

- **Manually Logging In Users**
- If you need to log an existing user instance into your application, you may simply call the login method with the instance:

```
$user = User::find(1);
Auth::login($user);
```
- This is equivalent to logging in a user via credentials using the attempt method.
- **Protecting Routes**
- Route filters may be used to allow only authenticated users to access a given route. Laravel provides the auth filter by default, and it is defined in app/filters.php.
- **Protecting A Route**

```
Route::get('profile', array('before' => 'auth', function()
{
 // Only authenticated users may enter...
}));
```

## CSRF Protection

Laravel provides an easy method of protecting your application from cross-site request forgeries.

### Inserting CSRF Token Into Form

```
<input type="hidden" name="_token"
value="<?php echo csrf_token(); ?>">
```

### Validate The Submitted CSRF Token

```
Route::post('register', array('before' => 'csrf',
function()
{
 return 'You gave a valid CSRF token!';
}));
```

# Module 7

## HTTP Basic Authentication

HTTP Basic Authentication provides a quick way to authenticate users of your application without setting up a dedicated "login" page. To get started, attach the auth.basic filter to your route:

### Protecting A Route With HTTP Basic

```
Route::get('profile', array('before' => 'auth.basic', function()
{
 // Only authenticated users may enter...
}));
```

By default, the basic filter will use the email column on the user record when authenticating. If you wish to use another column you may pass the column name as the first parameter to the basic method in your app/filters.php file:

```
Route::filter('auth.basic', function()
{
 return Auth::basic('username');
});
```

### Setting Up A Stateless HTTP Basic Filter

You may also use HTTP Basic Authentication without setting a user identifier cookie in the session, which is particularly useful for API authentication. To do so, define a filter that returns the onceBasic method:

```
Route::filter('basic.once', function()
{
 return Auth::onceBasic();
});
```

If you are using PHP FastCGI, HTTP Basic authentication will not work correctly by default. The following lines should be added to your .htaccess file:

# Module 7

- RewriteCond %{HTTP:Authorization} ^(.+)\$
- RewriteRule .\* - [E=HTTP\_AUTHORIZATION:%{HTTP:Authorization}]
- **Password Reminders & Reset**
- **Model & Table**
- Most web applications provide a way for users to reset their forgotten passwords. Rather than forcing you to re-implement this on each application, Laravel provides convenient methods for sending password reminders and performing password resets. To get started, verify that your User model implements the Illuminate\Auth\Reminders\RemindableInterface contract. Of course, the User model included with the framework already implements this interface, and uses the Illuminate\Auth\Reminders\RemindableTrait to include the methods needed to implement the interface.
- **Implementing The RemindableInterface**

```
use Illuminate\Auth\Reminders\RemindableTrait;
use Illuminate\Auth\Reminders\RemindableInterface;
class User extends Eloquent implements RemindableInterface {

 use RemindableTrait;

}
```
- **Generating The Reminder Table Migration**
- Next, a table must be created to store the password reset tokens. To generate a migration for this table, simply execute the auth:reminders-table Artisan command:
  - php artisan auth:reminders-table
  - php artisan migrate

# Module 7

## Password Reminder Controller

Now we're ready to generate the password reminder controller. To automatically generate a controller, you may use the auth:reminders-controller Artisan command, which will create a RemindersController.php file in your app/controllers directory.

```
php artisan auth:reminders-controller
```

The generated controller will already have a getRemind method that handles showing your password reminder form. All you need to do is create a password.remind [Template](#) by creating a file remind.blade.php in the app/views/password/ directory. This view should have a basic form with an email field. The form should POST to the RemindersController@postRemind action.

A simple form on the password.remind view might look like this:

```
<form action="{{ action('RemindersController@postRemind') }}" method="POST">
 <input type="email" name="email">
 <input type="submit" value="Send Reminder">
</form>
```

In addition to getRemind, the generated controller will already have a postRemind method that handles sending the password reminder e-mails to your users. This method expects the email field to be present in the POST variables. If the reminder e-mail is successfully sent to the user, a status message will be flashed to the session. If the reminder fails, an error message will be flashed instead.

Within the postRemind controller method you may modify the message instance before it is sent to the user:

```
 Password::remind(Input::only('email'), function($message)
{
 $message->subject('Password Reminder');
});
```

# Module 7

Your user will receive an e-mail with a link that points to the `getReset` method of the controller. The password reminder token, which is used to identify a given password reminder attempt, will also be passed to the controller method. The action is already configured to return a `password.reset` template which you should build. The token will be passed to the view, and you should place this token in a hidden form field named `token`. In addition to the token, your password reset form should contain `email`, `password`, and `password_confirmation` fields. The form should POST to the `RemindersController@postReset` method.

A simple form on the `password.reset` view might look like this:

```
<form action="{{ action('RemindersController@postReset') }}" method="POST">
 <input type="hidden" name="token" value="{{ $token }}>
 <input type="email" name="email">
 <input type="password" name="password">
 <input type="password" name="password_confirmation">
 <input type="submit" value="Reset Password">
</form>
```

Finally, the `postReset` method is responsible for actually changing the password in storage. In this controller action, the Closure passed to the `Password::reset` method sets the `password` attribute on the `User` and calls the `save` method. Of course, this Closure is assuming your `User` model is an [Eloquent model](#); however, you are free to change this Closure as needed to be compatible with your application's database storage system.

If the password is successfully reset, the user will be redirected to the root of your application. Again, you are free to change this redirect URL. If the password reset fails, the user will be redirect back to the reset form, and an error message will be flashed to the session.

# Module 7

## Password Validation

By default, the Password::reset method will verify that the passwords match and are  $\geq$  six characters. You may customize these rules using the Password::validator method, which accepts a Closure. Within this Closure, you may do any password validation you wish. Note that you are not required to verify that the passwords match, as this will be done automatically by the framework.

```
 Password::validator(function($credentials)
{
 return strlen($credentials['password']) >= 6;
});
```

Note: By default, password reset tokens expire after one hour. You may change this via the reminder.expire option of your app/config/auth.php file.

# Module 7

## Encryption

Laravel provides facilities for strong AES encryption via the mcrypt PHP extension:

### **Encrypting A Value**

```
$encrypted = Crypt::encrypt('secret');
```

Note: Be sure to set a 16, 24, or 32 character random string in the key option of the app/config/app.php file. Otherwise, encrypted values will not be secure.

### **Decrypting A Value**

```
$decrypted = Crypt::decrypt($encryptedValue);
```

### **Setting The Cipher & Mode**

You may also set the cipher and mode used by the encrypter:

```
Crypt::setMode('ctr');
```

```
Crypt::setCipher($cipher);
```

# Thank You