

# **Server-Side Web Framework**

Avinash Maskey

# Overview

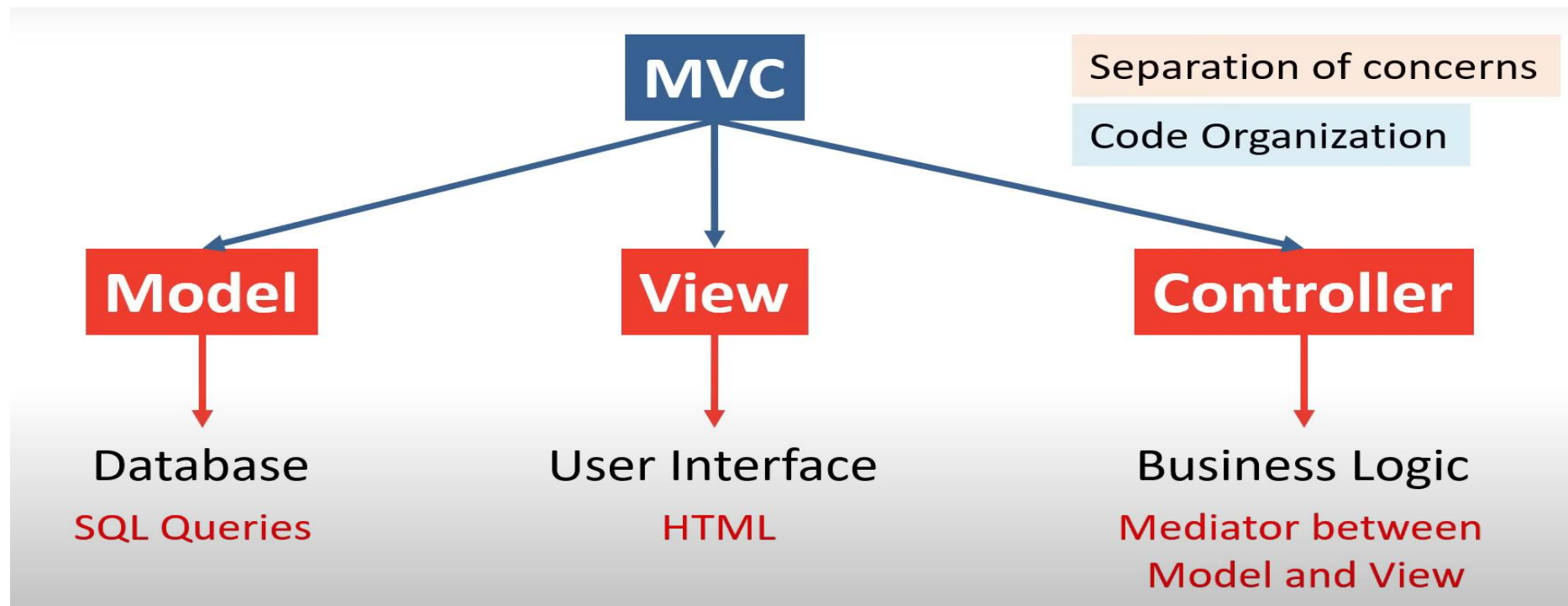
- A **server-side web framework** is a software framework designed to support the development of dynamic websites, web applications, and web services.
- A **framework** in programming is a tool that provides ready-made components or solutions that are customized to speed up development.
- It provides a standard way to build and deploy web applications on the internet. Server-side frameworks handle the HTTP requests from clients, process them on the server, and then send the response back to the client.
- This typically involves database interaction, backend logic, data processing, and rendering the appropriate HTML, CSS, and JavaScript for the client.

# Key Components and Features of Server Side Framework

- **Routing:** Maps incoming requests to the appropriate controller and action/method.
- **Controllers:** Handle incoming requests, process data, and return responses.
- **Models:** Represent the application's data structure, usually mirroring database tables.
- **Views:** Templates for the HTML output that the application sends to browsers.
- **Middleware:** Filters that execute before or after controllers to modify requests or responses.
- **Database Abstraction:** Provides a way to interact with databases using high-level APIs instead of raw SQL.
- **Authentication and Authorization:** Manage user identification and permission levels.
- **Caching:** Temporarily stores content to reduce load times and database access.
- **Session Management:** Tracks user sessions across requests.
- **Error Handling:** Manages errors gracefully, providing useful feedback to users and developers.

# What is MVC Framework?

- The **Model-View-Controller (MVC) framework** is a software architectural pattern that separates an application into three main logical components: **Model**, **View**, and **Controller**.
- Each of these components is built to handle specific aspects of the application's development.
- The MVC pattern is widely used in web application development with numerous benefits, including modularity, simplicity in managing large applications, and ease of maintenance.



# Components of MVC (1)

- **Model:**

- The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic related to the data.
- For example, if you're building a book library system, the Model might represent a book with properties like title, author, and ISBN, and it might contain methods to fetch, save, or update the book data in the database.

- **View:**

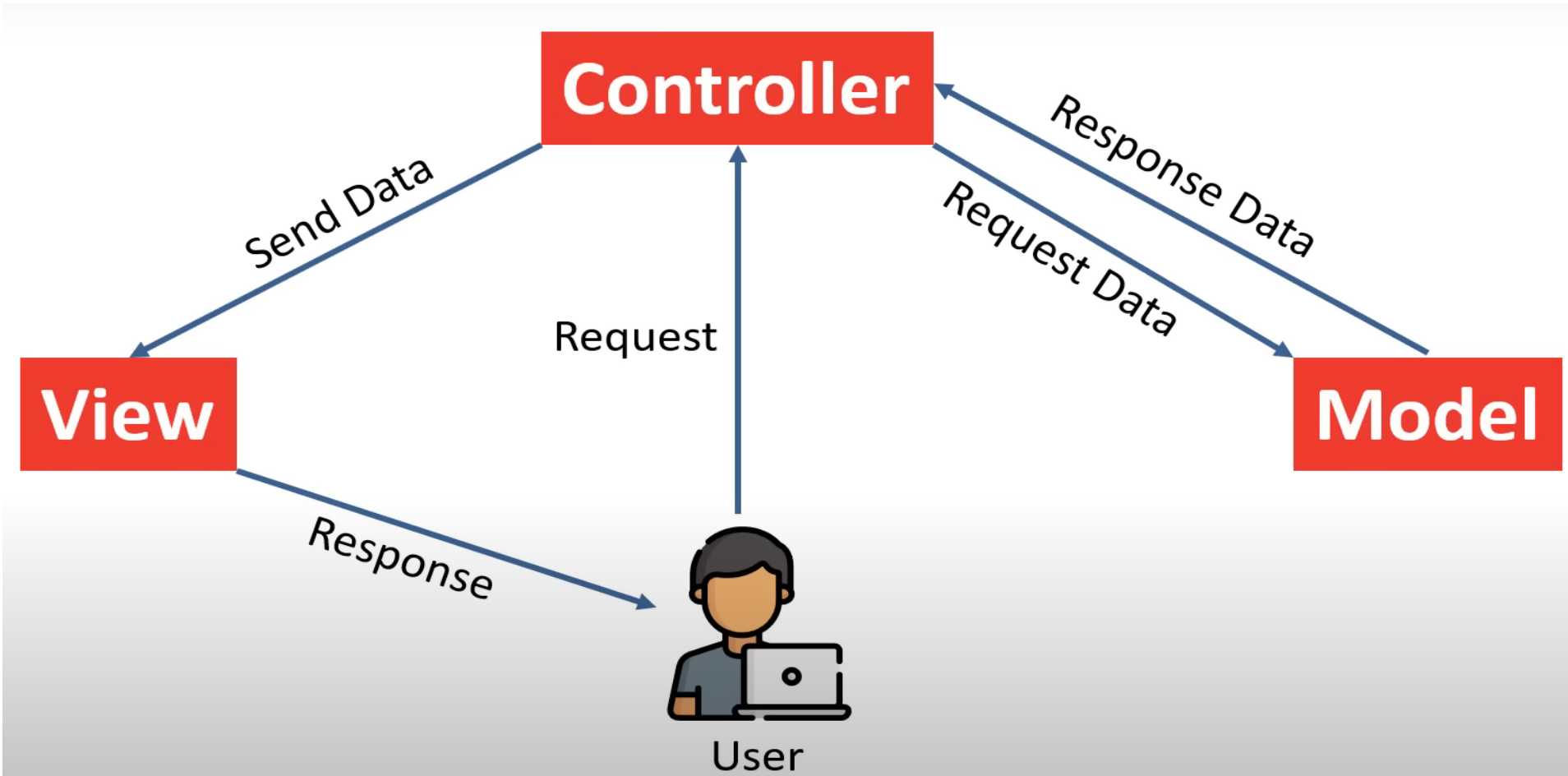
- The View component is used for all the UI logic of the application. For views, you create all the UI components that the users interact with. This includes everything from buttons and input fields to entire layouts and frames.
- Using the book library example, the View could be the pages that show a list of books, a page for adding a new book, or a page for editing an existing book's details.

# Components of MVC (2)

- **Controller**

- The Controller acts as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output.
- For instance, when a user wants to edit a book in the library, they interact with the View, which then uses the Controller to retrieve the book's details from the Model, and after the edit, it updates the book information using the Model again.

# MVC Framework Workflow



# Example of MVC in Action

- Let's illustrate this with a simplified example of an MVC application for managing a library:

**Scenario:** A user wants to view a list of all books in the library.

- **User Action:** The user navigates to the library's webpage to see a list of books.
- **Controller:** The web server routes this request to the appropriate controller for handling book lists. The controller processes the request, asking the Model for information.
- **Model:** The Model queries the database for a list of all books available in the library. This list is then returned to the Controller.
- **Controller:** With the list of books from the Model, the Controller selects a View that is designed for displaying a list of books.
- **View:** The Controller passes the list of books to the View. The View renders the HTML UI, which includes the list of books. This rendered View is sent back to the user's web browser, effectively showing the user the list of books in the library.



# Benefits of MVC

- **Separation of Concerns:** By dividing the application into models, views, and controllers, MVC makes it easier to manage code (or organize code) because each part can be developed independently.
- **Support for Parallel Development:** Different developers can work on the Model, View, and Controller simultaneously, which speeds up the development process.
- **Ease of Modification:** Because of the separation, changing the UI does not affect the data handling, and vice versa. For example, redesigning the webpage (View) won't require altering the data models.
- **Improved Support for Test-Driven Development (TDD):** Each component can be tested independently, which makes unit testing and debugging easier.

# Other Architectural Frameworks Used In Server-side Development (1)

- **Model-View-ViewModel (MVVM)**

- Model-View-ViewModel (MVVM) is a structural design pattern that is somewhat similar to MVC but is more commonly used in applications with rich user interfaces, including web applications developed with frameworks like AngularJS or Knockout.js. MVVM facilitates a clear separation between the presentation logic and the business logic of an application.
  - ❖ Model: Represents the data and business logic, similar to MVC.
  - ❖ View: The UI layer that displays the data (the user interface).
  - ❖ ViewModel: Acts as an intermediary between the Model and the View. It handles the view logic, converting the data from the Model into a format that the View can display.
- The key component here is the ViewModel, which is responsible for exposing methods, commands, and other properties that help maintain the state of the View, manipulate the Model as results of actions on the View, and trigger events in the UI.

# Other Architectural Frameworks Used In Server-side Development (2)

- **Model-View-Presenter (MVP)**

- Model-View-Presenter (MVP) is a derivative of the MVC framework that introduces a Presenter between the Model and View to take on the burden of handling all UI logic. MVP is particularly popular in the development of Android applications.
  - ❖ Model: Contains the business logic and data. It notifies the Presenter of any data changes.
  - ❖ View: Displays the data (the UI) and notifies the Presenter of any user actions.
  - ❖ Presenter: Acts as a middleman that retrieves data from the Model and formats it for display in the View. Unlike MVC's Controller, the Presenter also decides what happens when a user interacts with the View.
- In MVP, the View more actively involves the Presenter in decision-making about UI changes, making the Presenter somewhat more heavyweight than the Controller in MVC.

# Other Architectural Frameworks Used In Server-side Development (3)

- **Model-View-Adapter (MVA)**

- Model-View-Adapter (MVA) is a variation that introduces an Adapter between the View and the Model to allow for more flexibility in how data is presented to the user. The Adapter transforms data from the Model to a form that is more suitable for the View.
  - ❖ Model: The data layer.
  - ❖ View: The presentation layer (UI).
  - ❖ Adapter: Transforms data from the Model for the View. It also handles user actions and updates the Model accordingly.
- MVA is less common but can be useful in scenarios where the data model and the presentation layer need to be kept at a significant distance from one another, such as in applications requiring the transformation of data into different formats for presentation.

# Other Architectural Frameworks Used In Server-side Development (3)

- **Three-Tier Architecture**

- The Three-Tier Architecture is a broader architectural pattern that divides applications into three logical and physical computing tiers: presentation, application, and data. This is more of an architectural style than a strict framework and can encompass MVC, MVP, or other patterns in its layers.
  - ❖ Presentation Tier: The user interface and user interaction management (could employ MVC, MVP, or MVVM).
  - ❖ Application Tier (Business Logic Layer): Handles all application operations, business logic, and client communications.
  - ❖ Data Tier: Manages data persistence and database operations.
- This architecture is widely used in enterprise applications for its scalability, security, and ability to distribute across multiple platforms.

# Popular MVC Frameworks

- **Laravel**
- Symfony
- CodeIgniter
- Yii
- CakePHP
- Zend Framework

# Introduction to Laravel

- **Laravel** is a free, open-source and one of the more popular PHP web Framework based on MVC architectural pattern.
- It simplifies the development process by providing a clean and elegant syntax and tools needed for large, robust applications.
- Key Features of Laravel:
  - **Eloquent ORM:** An advanced implementation of the active record pattern, providing an expressive way to interact with databases.
  - **Blade Templating Engine:** Allows for expressive and powerful templating, with inheritance and sections.
  - **Artisan Console:** A built-in tool for performing repetitive and complex tasks easily.
  - **Migration System:** Version control for your database, allowing you to modify and share the application's database schema.
  - **Security:** Offers robust security features, including protection against SQL injection, cross-site request forgery, and cross-site scripting.
  - **API Support:** Simplifies the process of building RESTful APIs.

# Creating a Laravel Project - Laravel Installation (1)

- Before creating your first Laravel project, make sure that your local machine has **PHP (Xampp) and Composer installed**. If you are developing on macOS, PHP and Composer can be installed in minutes via Laravel Herd. In addition, we recommend installing Node and NPM.
- After you have installed PHP and Composer, you may create a new Laravel project via Composer's create-project command:

*composer create-project laravel/laravel example-app*

- Or, you may create new Laravel projects by globally installing the Laravel installer via Composer:

*composer global require laravel/installer*

*laravel new example-app*



# Creating a Laravel Project - Laravel Installation (2)


- Once the project has been created, start Laravel's local development server using Laravel Artisan's serve command:

```
cd example-app  
php artisan serve
```

- Once you have started the Artisan development server, your application will be accessible in your web browser at <http://localhost:8000>.
- Next, you're ready to start taking your next steps into the Laravel ecosystem. Of course, you may also want to configure a database.
- In Laravel, environment-based configuration is managed through a `.env` file located at the root of the application. This file allows you to define settings that vary between development and production environments, such as database connections, mail server details, and application keys.
- Since these settings often include sensitive information, the `.env` file should not be committed to source control to prevent exposure of credentials.

# Databases and Migrations (1)

- Now that you have created your Laravel application, you probably want to store some data in a database. By default, your application's .env configuration file specifies that Laravel will be interacting with a MySQL database and will access the database at 127.0.0.1.
- If you do not want to install MySQL or Postgres on your local machine, you can always use a SQLite database. SQLite is a small, fast, self-contained database engine.
- To get started, update your .env configuration file to use Laravel's sqlite database driver. You may remove the other database configuration options:

A screenshot of a code editor showing the configuration for a SQLite database in a .env file. The first line, 'DB\_CONNECTION=sqlite', is highlighted in green. The subsequent lines, 'DB\_CONNECTION=mysql', 'DB\_HOST=127.0.0.1', 'DB\_PORT=3306', 'DB\_DATABASE=laravel', 'DB\_USERNAME=root', and 'DB\_PASSWORD=', are in red. The editor has a dark background with a small trash icon in the top right corner.

```
DB_CONNECTION=sqlite
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

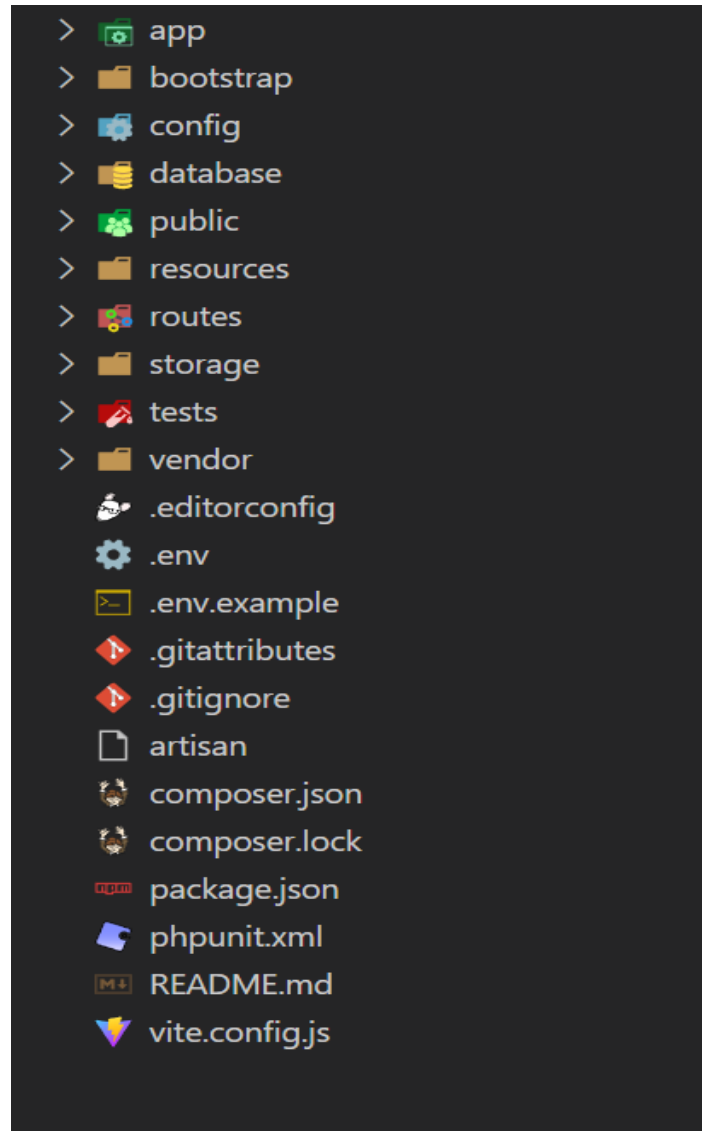
# Databases and Migrations (2)

- Once you have configured your SQLite database, you may run your application's database migrations, which will create your application's database tables:

*php artisan migrate*

- If an SQLite database does not exist for your application, Laravel will ask you if you would like the database to be created. Typically, the SQLite database file will be created at database/database.sqlite.

# The Root Directory Structure of Laravel (1)



# The Root Directory Structure of Laravel (2)

Directory	Description
app	The app directory holds the base code for your Laravel application.
bootstrap	The bootstrap directory contains all the bootstrapping scripts used for your application.
config	The config directory holds all your project configuration files (.config).
database	The database directory contains your database files.
public	The public directory helps start your Laravel project and maintains other necessary files such as JavaScript, CSS, and images of your project.
resources	The resources directory holds all the Sass files, language (localization) files, and templates (if any).
routes	The routes directory contains all your definition files for routing, such as console.php, api.php, channels.php, etc.
storage	The storage directory holds your session files, cache, compiled templates, and miscellaneous files generated by the framework.
test	The test directory holds all your test cases.
vendor	The vendor directory has all composer dependency files.

# Routes in Laravel – Defining Route

- Routes in Laravel are defined in the routes directory, within files like web.php for web routes and api.php for API routes. The simplest form of a route is a closure that returns a string or a view.
- **Example: A Basic Web Route**
  - Let's define a basic GET route that returns a simple greeting. You would place this code in the routes/web.php file:

```
Route::get('/', function () {  
    return 'Welcome to our application!';  
});
```
  - This route responds to the root URL (/) with a greeting.

# Routes in Laravel – Route Parameters (1)

- Routes can have parameters, allowing them to respond to a variety of inputs, such as user IDs or product names. Parameters are defined within { } braces in the route definition.
- **Example:** Required Parameter
  - Here's how you define a route with a required parameter:

```
Route::get('/user/{id}', function ($id) {  
    return 'User ID: ' . $id;  
});
```
  - This route will match any URL like /user/1, /user/2, etc., and the closure will receive the ID as a parameter.

# Routes in Laravel – Route Parameters (2)

- **Example:** Optional Parameters

- Parameters can also be optional, which is indicated by adding a ? at the end of the parameter name.
- In the route's closure, you must provide a default value for optional parameters:  

```
Route::get('/dept/{id?}', function (string $id = 11) {  
    return 'Dept. Id: ' . $id;  
});
```
- This route can match both /dept and /dept/any-id, providing a default value if the id is not provided.



# Routes in Laravel – Named Route (1)

- **Named routes** allow you to reference a route by its name instead of its URL path when generating URLs or redirects, which can make your code more robust and easier to maintain.
- **Example:** Defining a Named Route
  - Here's how you define a named route:

```
Route::get('/user/profile', function () {  
    // Your route logic here.  
})->name('profile');
```

- You can then generate a URL to this route using the route function:  

```
$url = route('profile');
```
- And to generate a redirect response to this route, you can use the redirect function:  

```
return redirect()->route('profile');
```

# Routes in Laravel – Named Route (2)

- The image below can lead to **page not found error** if we change the route **about** to **about-us**.

<http://localhost/page/about>

```
Route::get('/page/about-us', function () {  
    return 'About Page';  
})->name('about');
```

first.blade.php

```
<a href='/page/about'>About</a>
```

second.blade.php

```
<a href='/page/about'>About</a>
```

third.blade.php

```
<a href='/page/about'>About</a>
```

# Routes in Laravel – Named Route (3)

- The solution to that error can be named route shown below.

<http://localhost/page/about>

```
Route::get('/page/about-us', function () {  
    return 'About Page';  
})->name('about');
```

first.blade.php

```
<a href='{{ route('about') }}'>About</a>
```

second.blade.php

```
<a href='{{ route('about') }}'>About</a>
```

third.blade.php

```
<a href='{{ route('about') }}'>About</a>
```

- **Note:** Using named routes is especially beneficial in larger applications, where the path of the route might change. Since you're referencing routes by name rather than by their actual path, updating the path in one place (the route definition) automatically applies everywhere that route is used.

# Routes in Laravel – Passing Data to Views From Routes (1)

- In Laravel, "**passing data to views from routes**" refers to the method of sending information from your application's routes (defined in your routes files, like web.php) directly to your views (blade templates).
- This is a common task in web development, allowing dynamic content to be displayed on web pages based on logic defined in your application. Let's go through a simple example to illustrate this concept.
- **Step 1: Define the Route**
  - First, you define a route in Laravel. This route will be responsible for handling a specific URL and returning a view. In this example, we'll create a route that passes an array of names to a view.
  - Open the routes/web.php file and add the following code:

```
Route::get('/greet', function () {  
    $names = ['Alice', 'Bob', 'Charlie'];  
    return view('greet', ['names' => $names]);  
});
```

# Routes in Laravel – Passing Data to Views From Routes (2)

- **Step 2: Create the View**
  - Next, you'll create the view file that will display the data passed from the route. Create a new file named greet.blade.php in the resources/views directory.
  - In this file, you can loop through the names array and display each name. Here's how the greet.blade.php file might look:

```
<h1>Greetings</h1>
<ul>
    @foreach($names as $name)
        <li>Hello, {{ $name }}!</li>
    @endforeach
</ul>
```

- **Step 3: Access the Route**
  - Now, if you start your Laravel application using php artisan serve and navigate to <http://localhost:8000/greet> in your browser, you will see the greeting messages displayed for each name in the array.

# Example: Passing Data to Views From Routes

- Code in routes/web.php

```
Route::get('/users', function () {  
    $users = [  
        ['id' => 1, 'name' => 'Ayush Ghalan', 'email' => 'ag@gmail.com', 'address' => 'Nepal'],  
        ['id' => 2, 'name' => 'Angel Dimaria', 'email' => 'a@gmail.com', 'address' => 'Argentina'],  
        ['id' => 3, 'name' => 'Cristiano Ronaldo', 'email' => 'c@gmail.com', 'address' => 'Portugal'],  
        ['id' => 4, 'name' => 'Lionel Messi', 'email' => 'm@gmail.com', 'address' => 'Argentina'],  
        ['id' => 5, 'name' => 'Ricardo Kaka', 'email' => 'rk@gmail.com', 'address' => 'Brazil']  
        // Add more users as needed  
    ];  
  
    return view('userpage', compact('users')); //compact() function creates an array from variables  
});
```

# Example: Passing Data to Views From Routes

- Code in resources/views/userpage.blade.php

```
<h2>Users List</h2>
<table border="1">
  <thead>
    <tr>
      <th>ID</th>
      <th>Name</th>
      <th>Email</th>
      <th>Address</th>
    </tr>
  </thead>
  <tbody>
    @foreach($users as $user)
      <tr>
        <td>{{ $user['id'] }}</td>
        <td>{{ $user['name'] }}</td>
        <td>{{ $user['email'] }}</td>
        <td>{{ $user['address'] }}</td>
      </tr>
    @endforeach
  </tbody>
</table>
```

**THANK YOU!**