#### **Avoid Internal JavaScript and CSS:**

* **Why:** To keep code organized and scalable.
* **Rule:** Don't write JavaScript and CSS directly in your HTML files. Keep them in separate files. Use Laravel Mix for combining and minifying them.

#### **Custom Static Helper for Common Functionality:**

* **Why:** Encourage reusable code for common tasks.
* **Rule:** Create a helper in the **app/Helpers** directory for each module. For example, an EmailHelper class can handle common email operations.

| **namespace App\Helpers;  use App\Mail\Admin\Emails\WelcomeMail; use Illuminate\Support\Facades\Mail;  class EmailHelper {  public static function sendWelcomeEmail($emailData): void  {  try {  Mail::to($emailData['email'])->send(new WelcomeMail($emailData));  } catch (\Exception $e) {  \Log::error('Error sending welcome email: ' . $e->getMessage());  }  } }** |
| --- |

#### 

#### 

#### 

#### 

#### **Use Components for Country, State, and City Selection:**

* **Why:** Simplify and standardize country, state, and city selections.
* **Rule:** Utilize Blade components for a consistent approach. Use them like this in your Blade views:

| **<x-common.country/> <x-common.state/> <x-common.city/>** |
| --- |

**\*\*Note:\*\* These components come with default names and IDs (like name="country" id="country"). Customize them based on your application's needs.**

#### **Migration Best Practices:**

* **Why:** Ensure a structured approach to database changes.
* **Rule:** When creating a **new table** or adding a new column to an **existing table**, always use migrations. For each **migration**, create a corresponding **factory** and **seeder**. This ensures consistent data generation during testing.
* When creating table columns, adhere to a consistent naming convention to ensure clarity and uniformity. Specifically, enforce the **use of underscores instead of camel case or other styles for column names**. For example, if creating a column for **user mobile**, name it "**user\_mobile**," and if creating a column for **user name**, name it "**user\_name**." This mandatory naming convention **fosters** a standardized and easily comprehensible database schema, promoting readability and maintainability across the development team.

#### **Using Theme Components and Blade Syntax:**

* **Note:** Don't add any jQuery as we've included all required files. Follow the **documentation for using theme components** [**here**](https://preview.keenthemes.com/html/metronic/docs/index)**.**
* **Blade Syntax Example:**Use the following syntax when creating a new Blade file:

| <**x-admin.layout.app-layout**>  <**x-slot** name="title">  Page title here   </**x-slot**>   {{-- Page layout design and data here - this is page of body --}}   {{-- Always place this at the end of the page, just above the closing body tag --}}  <**x-slot** name="script">  <**script** src="{{ asset('assets/js/extranal.js') }}"></**script**>  </**x-slot**> </**x-admin.layout.app-layout**> |
| --- |

### **ORM Model Rules:**

**Objective:** Ensure consistency and efficiency in managing your database.

**Rule 1:** Stick to Eloquent **ORM** rules. Use migrations to set up and modify your database structure.

**Rule 2** - Best Practices for Relationships:

* Always use Eloquent relationships like **HasOne**, **HasMany**, **BelongsTo**, etc., to link your database tables.
* When you create models and migrations, immediately define and set up relationships to ensure a well-connected and organized database.

**Rule 3**: For every **migration** or **model** you create, always make sure to create a **seeder** and a **factory**.

By following these rules, you're making sure your database is structured logically, relationships between tables are well-defined, and you have consistent and meaningful test data for efficient testing. This approach improves the overall efficiency and reliability of your database operations.

**Rule 4:** When defining relationships between tables using **foreign keys**, always specify them as **foreign keys** instead of just using **integers**. Follow a proper structure for enhanced clarity.

### 

### **Database Naming Convention:**

Rule: Use lowercase or underscores for creating database table columns and names.

**Explanation:**

* Always use lowercase letters or underscores to name database tables and columns.
* Avoid capital letters, special characters, or uppercase words in table names.
* This convention ensures consistency and compatibility across various database systems.

**Example:**

| **CREATE TABLE product\_categories (  id INT PRIMARY KEY,  category\_name VARCHAR(255),  description TEXT );  CREATE TABLE users (  user\_id INT PRIMARY KEY,  username VARCHAR(50),  email VARCHAR(255),  password VARCHAR(255) );** |
| --- |

### 

### 

### 

### **Controller Naming Convention:**

**Rule:** Use capitalized naming convention for controllers.

**Explanation:**

* Capitalized naming convention involves capitalizing the first letter of each word, also known as "Pascal Case."
* Apply this convention when creating controllers to enhance readability and maintain a consistent coding style.
* For example, if you have a controller related to products, name it "**ProductController**" or "**ProductCategoryController**" for a product category-related controller.

Example Controller Naming:

| app/ |-- Http/ | |-- Controllers/ | |-- Product/ | |-- ProductController.php | |-- ProductCategoryController.php | |
| --- |

### **View Naming Convention:**

**Rule:** Use kebab case for naming view files and directories.

**Explanation:**

* **Kebab case** is a naming convention where words are separated by hyphens ("-").
* It improves readability and consistency in your project structure.
* For example, if you have a view related to user creation, name the directory and file as "user/create-form."

| resources/ |-- views/ | |-- user/ | |-- create-form.blade.php | |-- edit-profile.blade.php | | |-- admin/ | |-- dashboard.blade.php | |-- settings/ | |-- general-settings.blade.php | |-- security-settings.blade.php |
| --- |

### **Git Branch Naming and Structure Guidelines:**

**Rule 1:** Always follow the **kebab case** for Git branch names.

**Rule 2:** Use separate branches for UI and backend development within a module.

**Branch Naming Example:**

* If working on the "**Product**" module, UI developers should create a branch like **product-ui**, and backend developers should use **product-backend.**

**Variable Naming Convention:**

**Rule: Enforce Lowercase and Descriptive Naming for Variables**

When declaring variables within your code, adhere to a consistent naming convention for enhanced readability and maintainability. Utilize **lowercase** letters for variable names and avoid the use of special characters or numbers. Additionally, ensure that variable names are **contextually relevant**, providing **meaningful** insights into their purpose within the code.

For example, use variable names like **$name**, **$username**, or **$user\_name** to represent different aspects of user information. **Avoid ambiguous** or unrelated names such as $a, $xyz, $name1, $1name, $Name, or $UserName, as they hinder code understanding and collaboration.

| $name = "surinder singh"; $author\_name = "surinder singh"; $mobile\_number = "9876567890";  $age = 26;  function displayData($name,$author\_name,$mobile\_number ){  return "User Name" .$name; } |
| --- |