



Laravel

—

Period

Lesson Plan

Teacher: Sadhin Halder

Date: 22/03/2023

Overview & Purpose

Laravel Framework strives to provide an excellent experience to the developer, providing powerful features such as-

- 1) Full dependency injection,
- 2) A layer of expressive database abstraction
- 3) Queues and scheduled jobs
- 4) Unit and integration testing and more.

Progressive structure

The main team of the Laravel project likes to call it “progressive” framework. With this, they mean that Laravel is able to track the growth of its project and, in the case of junior programmers or beginners, its professional development.

Laravel offers-

- 1) Robust tools for dependency injection
- 2) Unit test
- 3) Queues
- 4) Real-time events and more.

The framework is set to build professional web applications and ready to handle corporate workloads.

Scalable Structure

The PHP language, when used with good software architecture concepts, offers high scalability to the systems created with it. So it's easy to understand how Laravel can be incredibly scalable.

In addition to the friendly nature of PHP scaling, thanks to the excellent architecture employed in the Laravel framework, and the built-in support for fast and distributed cache systems like **Redis**, **scale horizontally** with Laravel is very simple. In fact, there are hundreds of commercial apps made with Laravel that were easily scaled to handle hundreds of millions of requests per month.

Project Design Patterns in Laravel

Being a framework that applies a robust software architecture, valuing good practices of source code development and structuring, it employs several software design patterns in its infrastructure.

The following list presents the main design patterns implemented by the Laravel framework and describes an overview of each:-

MVC - Model View Controller:

- 1) **Model-View-Controller** is a software **architecture pattern that separates the representation of information from the control of user interaction and business rules.**
- 2) The **Model (model)** is responsible for **managing application data, business rules, logic and functions.**
- 3) A **View (view)** is any **data representation output**, such as an **HTML screen, report, table, or dataset** in **JSON or XML** format. It is possible to have multiple views of the same dataset, such as a bar chart for management and a tabular view for a master data.
- 4) The **Controller (controller)** mediates the input (**customer requests**), converting it into orders for the **model or view**. The controller is responsible for **managing (orchestrating)** the service flow to a **request**.

Facade:

In software design patterns, a facade (**a word of French origin**) is an **object** that makes available a simplified interface for one of the functionalities of an API, for example.

A facade can make a software library easier to understand and use, or reduce dependencies on a library's internals.

Service Provider:

An application is an aggregation of cohesive services.

While an application offers a broad set of functionality in terms of programming interfaces (APIs) and classes, a service provides access to some specific functionality

or features. The service defines interfaces to functionality and a way to retrieve an implementation.

For example, consider an application that provides a variety of information about a geographic location, such as real estate data, weather information, demographics, etc. The weather service, a part of the application, can only define the interface for retrieving weather information.

The service provider interface (**SPI**) is the set of public interfaces and abstract classes that a service defines. The **SPI** can be represented by a single interface (**type**) or **abstract class** or a set of **interfaces** or **abstract classes** that **define the service contract**.

IoC - Inversion of Control:

Inversion of Control (**IoC**) is the name given to the development pattern of computer programs where the sequence (**control**) of method calls is inverted in relation to traditional programming, that is, it is not determined directly by the programmer.

This control is delegated to a software infrastructure often called a **container** or any other component that can take control over execution. This is a very common feature in some frameworks.

In Laravel, the entry point for requests is unique, through the **public/index.php** file, which instantiates the **main controller, or front controller**, which is responsible for the inversion of control, identifying by the request parameters which method of which controller should respond to it.

DI - Dependency Injection:

Dependency Injection is **used to maintain a low level of coupling between different modules and components of a system**.

In this standard, dependencies between components are not defined by direct instantiations by programmers, but by the **configuration or standardization** use of a software infrastructure (**container**) that is responsible for **“injecting”** its declared dependencies into each component.

Dependency Injection is related to the Inversion of Control pattern but cannot be considered a synonym for it.

Front Controller and Dispatcher:

The front controller software design pattern defines a single entry point to your web application that handles all **HTTP requests**. For example in PHP: **index.php**, in Laravel specifically **public/index.php**

This code is responsible for loading **dependencies, executing security rules, internationalization, user authentication.**

Used in conjunction with the Dispatcher design pattern, it uses strategies for the **MVC** application **where the controller module (front controller) sends the processing to a dispatcher (dispatcher) which selects, based on the context of the request (parameters received by the HTTP request), the correct controller for executing the logic related to the request made.**

Front Controller is a design pattern that, together with M-V-C, enables the implementation of IoC - Inversion of Control in Web systems.

Observer:

Defines a **one-to-many dependency between objects so that when an object changes its state, all its dependents are automatically notified and updated.**

Allows interested objects (**observers**) to be notified of state changes or other events occurring in another object.

The Observer pattern is also called **Publisher-Subscriber, Event Generator, and Dependents.**

In the Laravel framework it is implemented with the concept of Listener.

ORM and ActiveRecord:

ORM (**Object Relational Mapping**) is an approach used to persist business object (**application**) information in a relational database (**SQLite, Oracle, Sybase, PostgreSQL, Mysql, etc**).

It performs, **through annotations in the classes that will represent the entities in memory, in the object-oriented pattern, the mapping with the tables and fields of the relational database.**

Active record is a design pattern where the interface of a given object has actions to manipulate its data, such as **Insert (Insert)**, **Update (Update)**, **Delete (Delete)** and **access properties and actions that directly correspond** to the *associated database columns*.

An instance of an object is associated with a single record (**tuple**) in the table. After creating and saving an object, a new record is added to the table. A loaded object gets its information from the database of data. When an object is updated, the corresponding record in the table is also updated.

Laravel Features and Functionality

We will list some of the functionalities delivered to the developer natively in its structure, providing productivity and security for our application.

Authentication:

The Laravel framework provides the implementation of standardized authentication in a simple way. This feature simplifies the development of applications with little complexity and simple authentication requirements, for systems with specific requirements, such as the use of access directories (eg **LDAP**), **OAUTH**, it is possible to override the default implementation.

It even provides packages for implementing authentication via tokens for RestFul APIs.

Routing:

Modeling service routes from requests to resources and actions of our system.

Routes are specified by context like web requests for classic apps, **API, console** .

It also offers limiting control of requests per minute and per client IP.

Security filters:

The framework easily protects our application from cross-site request forgeries (**CSRF**) attacks; applies sanitization of data inputs and allows the implementation of middleware for other filters if needed.

Cache:

Provides a unified **API** for various application caching systems. By specifying a driver, the framework uses it by default throughout our application.

Laravel supports tools like **Memcached and Redis**. By default, it is configured to use the drive file, which stores serialized objects in the file system. For large applications, it is recommended that you use an in-memory cache such as Memcached or APC.

Events:

Events in Laravel provide a simple implementation of the observer pattern, allowing **classes/objects** to subscribe to and listen to (**listeners**) events in our application.

Event classes are typically stored in the **app/Events directory**, while their listeners are stored in the **app/Listeners** directory.

An **example** of use would be the creation of a **UserCreated** event, which, when triggered, allows its listeners to perform actions such as **sending email, creating specific profiles, validating access after creating a user**.

Final considerations:

If your development Stack is based on the PHP programming language, but even if you work in a company or team that does not adopt the Laravel framework, it is essential to know its potential and, at least, have an overview of its characteristics and functionalities because, with around 22 million downloads by site metrics **Packagist.com**, it should not be ignored.



PHP Laravel Framework Architecture Components

Every development framework delivers standardized architecture to organize the activity of software project teams with good programming practices. With Laravel framework this is no different, and in this article you will have an overview of the components that make up your architecture.

Main elements of Laravel Architecture

The architecture created for the framework was modularly developed by characteristics.

Thus several components are delivered, each with a goal and a solution to common problems in the development of web softwares.

Next an overview of the main (or more used) components available in the core of the Laravel framework:

Service Containers:

Laravel service containers are a powerful tool for **managing class dependencies and performing dependency injection**. Through them the dependencies of a particular class are **“injected” (as instant objects)** in the class that will use it, through the builder or, in some cases, the **“setter”** methods.

Service Providers:

The service providers are the **central place for all bootstrapping** of Laravel applications.

Our own application built with the framework, as well as all of **Laravel’s core services, are initialized through service providers**.

In general, this **means registering service container bindings, event listeners, middleware, and even routes**. In summary, **service providers are the central location for configuring your application**.

If we open the **config/app.php** file, we’ll see an **array of configured providers**. **These are all of the service provider classes that will be loaded in our application**. By default, a set of Laravel’s core service providers are listed in this array.

These providers initialize Laravel's core components like the **mailer**, **queue**, **cache**, **database**, and **others**. Many of these providers are "**deferred providers**", meaning they **won't be loaded on every request but only when the services they provide are actually needed**.

Facades:

The facades provide a "**static**" interface to the **classes available in the application's service container**. Laravel comes with many facades that provide access to almost all of its **resources**.

Laravel's facades serve as "**static proxies**" to **underlying classes in the service container**, providing the benefit of a concise and expressive syntax while maintaining more testability and flexibility than traditional static methods.

All Laravel facades are defined in the Illuminate\Support\Facades namespace.

Routing - Laravel routes:

Routes are configurations that teach Laravel which "**routes**" our application is prepared to accept in **HTTP requests**, and who (**controller actions**) will handle each one of them.

All Laravel **routes are defined in its route files, which are located in the routes directory**. These files are automatically loaded by our application's **App\Providers\RouteServiceProvider**. The **routes/web.php** file **defines the routes that are for your web interface**. These routes are assigned to the web middleware group, which provides resources such as **session state and CSRF protection**. The routes in **routes/api.php** have no state and are assigned to the api middleware group.

Middlewares:

Middlewares are a **mechanism for inspecting and filtering HTTP requests that enter our application**. For example, Laravel includes a middleware that checks if the user of our application is authenticated. If not, the middleware will redirect the user to the login screen of our application. However, if the user is authenticated, the middleware will allow the request to proceed in the application's control flow.

We can build additional middlewares to perform a variety of tasks beyond authentication. For example, an auditing middleware can log all HTTP requests received in our application.

There are several middlewares included in the Laravel framework, and they are located in the `app/Http/Middleware` directory.

Controllers:

Controllers are used so that instead of defining all the request handling logic as scripts or anonymous functions in our route files, we can organize this behavior using “controller” classes.

Controllers can group related request handling logic into a single class. For example, a `UserController` class can handle all incoming requests related to users, including displaying, creating, updating, and deleting users. By default, controllers are stored in the `app/Http/Controllers` directory.

Views:

Obviously, it's not practical to return entire strings of HTML documents directly from our routes and controllers.

To solve this, **Laravel implements the views layer**. Views - views provide a convenient way to put all of your HTML in separate files. In this way, views separate our **controller/application logic from our presentation logic** and are stored in the `resources/views` directory.

Blade Engine Template:

So that you **don't have to mix PHP code in our views, along with HTML code, Laravel provides the Blade template engine.**

It is a simple yet powerful template engine that, unlike some PHP view template engines, does not restrict you from using pure PHP code in them. In fact, **all Blade templates are compiled into pure PHP code and cached until modified, which means that Blade essentially adds zero overhead to our application.** Blade template files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

HTTP Requests:

The `Illuminate\Http\Request` class in Laravel provides an **object-oriented** way to interact with the current HTTP request being handled by our application, as well as retrieve **input, cookies, and files** that were sent with the request.

HTTP Responses:

Typically, we won't just return strings or simple arrays from our route actions or controllers. Instead, we'll return instances or full views of the `Illuminate\Http\Response` **object**.

By returning a **complete response instance, the object allows us to customize the HTTP status code of the response and headers**. A Response instance is inherited from the `Symfony\Component\HttpFoundation\Response` class, which **provides a variety of methods to construct HTTP responses**.

Session:

Since HTTP-based applications **do not maintain state, sessions provide a way to store information about the user between client requests**. This information is typically placed in a persistent **storage/backend** that can be **accessed from subsequent requests**.

Laravel comes with a variety of session backends that are accessed through an expressive and unified API. It includes support for popular backends such as **Memcached, Redis, and databases**.

Validations:

Laravel provides several different approaches to validate the input data of our application. The most common is through the validation method available on all incoming HTTP requests, but the framework also enables other strategies to validate and sanitize user input data.

It also includes a wide variety of validation rules that make it easy for us to apply to the data, even providing the ability to validate if the values are unique in a specific database table.

In addition to these **advanced features in data validation**, it is still possible to write our own rules for reuse throughout the application.

Error Handling:

When we start a new Laravel project, error and exception handling is already natively configured for you in our application.

The **App\Exceptions\Handler class** is where all exceptions thrown by our application are registered and then processed and managed for the user.

Logging:

To help us keep track of what's happening in our application, Laravel provides robust logging services that allow us to record messages in files, the operating system's error log, a database, and even to Slack to notify our entire team.

Laravel's log is based on "**channels**". Each channel represents a specific way of logging information. **For example**, the single channel logs log files to a single log file, while the slack channel sends log messages to Slack. Log messages can be recorded on multiple channels simultaneously and based on their severity.

Under the hood, **Laravel uses the Monolog library**, which provides support for a variety of powerful log handlers. Laravel makes it easy to configure these handlers, allowing you to mix and match them to customize our application's log management.

Database:

Almost all modern web or mobile applications interact with a database.

Laravel makes the use of databases for information management extremely simple, supporting a variety of databases through: raw SQL, a fluent query builder, and the Eloquent ORM.

Currently, Laravel provides native support for four databases:

- 1) MySQL 5.7+
- 2) PostgreSQL 9.6+
- 3) SQLite 3.8.8+
- 4) SQL Server 2017+

Other databases are supported using extension packages of the framework such as:

- Sybase ASE / SQLAnywhere, with the package developed by the State University of Ponta Grossa - UEPG, available at: <https://github.com/uepg/laravel-sybase>
- MongoDB, with the package developed by Jens Segers, available at: <https://github.com/jenssegers/laravel-mongodb>

Query Builder:

The Database Query Builder is a Laravel component that **provides a convenient and fluent interface for creating and executing SQL queries to the database.**

It can be used to perform most **database operations in our application and works seamlessly with all database systems supported by Laravel or extended with packages.**

The Laravel Query Builder uses parameter binding through **PDO (PHP Data Objects)** to **protect our application against SQL injection attacks.** Therefore, we don't need to worry about cleaning or sanitizing user inputs with strings passed to the query builder as query bindings.

Eloquent ORM:

Laravel includes **Eloquent, an Object-Relational Mapping (ORM)** that makes using and interacting with our database a simple and flexible task.

When using the Laravel Eloquent component, each database table has a corresponding **"Model" (from the MVC - Model View Controller)** that is used to interact with that table.

Eloquent is an ORM that applies the Active Record design pattern.

In addition to retrieving records from the database table, Eloquent models also allow for direct and highly productive **insertion, updating, and deletion** of records in the table.

Advanced components of Laravel

In addition to the most common and essential components for any Laravel application, described earlier, the framework delivers a rich architecture of more advanced elements, allowing us as a developer to build large-scale web applications with high productivity and reduced complexity.

The following are some of the **advanced components** that we can use with Laravel and are natively delivered by the framework:

Console Artisan:

Artisan is the command-line interface included in Laravel.

Artisan exists at the root of our application as the artisan script and provides various useful commands that can help you as you build our application.

To see a list of all available Artisan commands, we can use the list command at the root of the project:

```
php artisan list BASH
```

In addition to delivering a series of useful commands with Artisan, we can build our own commands in our application, making administrative tasks easier to run via command line.

Tinker (REPL):

Laravel Tinker is a **powerful REPL for the framework, developed with the PsySH package.**

All Laravel applications include Tinker by default.

Tinker allows us to interact with our entire Laravel application on the command line, including our Eloquent models, Jobs, events, and much more.

To enter the Tinker environment, run the tinker command of Artisan:

```
php artisan tinker BASH
```

Broadcasting:

In many modern web applications, **WebSockets** are used to implement real-time updated user interfaces.

Their common use is for when some data is updated on the server, a message is sent over a WebSocket connection to be received and handled by the client - web browser or mobile application.

WebSockets provide a more efficient alternative to continuously querying our application's server for data changes that need to be reflected in its UI - user interface.

For example, imagine that our application is capable of exporting a user's data to a CSV file and sending it via email.

However, creating this CSV file takes several minutes, so we opt to create and send the CSV in a queue job.

After the job runs and the CSV is created and sent to the user, we can use event broadcasting to dispatch an **App\Events\UserDataExported** event that is **received by our application's JavaScript**.

As soon as the event is received, we can display a message to the user that their CSV has been sent via email without them having to constantly refresh the browser page or implement timed Ajax calls asking the server if the export has completed.

To help us build features of this nature in our application, Laravel makes it easy to "broadcast" our server-side events over a WebSocket connection.

Broadcasting our Laravel events allows us to share the same event names and data between our server-side Laravel application and our client-side JavaScript application.

The basic concepts behind broadcasting are simple: clients connect to named channels on the frontend, while our Laravel application broadcasts events to these channels on the backend. These events can contain any additional data we want to make available to the front-end.

Cache:

Some data retrieval or processing tasks performed by our application may require a lot of CPU or take several seconds to complete.

For these cases, it is common to cache the data retrieved for a certain amount of time so that they can be quickly retrieved in subsequent requests for the same data contexts.

Cached data is usually persisted in a very fast data store, such as Memcached or Redis.

Fortunately, **Laravel provides an expressive and unified API for various cache back-ends, allowing us to take advantage of extremely fast data retrieval and speed up our web application.**

Collections:

The **Illuminate\Support\Collection class** provides a convenient and fluent wrapper for working with arrays and data vectors.

For example, consider the following code. We will use the collect helper to create a new collection instance from an array, execute the strtoupper function on each element, and then remove all empty elements:

```
$collection = collect(['ademir', 'alexandra', null])->map(function ($name) {  
    return strtoupper($name);  
})->reject(function ($name) {  
    return empty($name);  
});
```

As we can see, the **Collection** class allows us to chain our methods for fluent mapping and reduction of the underlying array. In general, collections are immutable, which means that each **Collection** method returns an entirely new **Collection** instance.

Events:

Laravel events provide an implementation of the **Observer** design pattern, allowing us to subscribe and listen to various events that occur within our application.

Event classes are usually stored in the **app/Events** directory, while their listeners are stored in **app/Listeners**.

Events serve as a great way to decouple various aspects of our application since a single event can have multiple listeners that don't depend on each other.

For example, we can send a Slack notification to our user whenever an order is placed. Instead of coupling our order processing code with the Slack notification code, we can generate an **App\Events\OrderShipped** event that a listener can receive and use to perform the logic for sending a Slack notification.

File Storage:

Laravel provides a powerful file system abstraction component, thanks to Frank de Jonge's PHP package Flysystem.

Laravel Flysystem integration provides simple drivers to work with local file systems, SFTP, and Amazon S3.

It is incredibly easy to switch between these storage options between the local development machine and the production server, as the API remains the same call for each system.

Inside the component configuration file, we can configure all of our file system's "disks". Each disk represents a specific storage driver and storage location.

Example settings for each supported driver are included in the configuration file so that we can modify the configuration to reflect our storage preferences and credentials.

For example, the local driver interacts with files stored locally on the server that runs the Laravel application, while the s3 driver is used to write to Amazon's S3 cloud storage service.

HTTP Client:

Laravel provides a minimal and expressive API around the Guzzle HTTP client, allowing us to make HTTP requests to communicate with other web applications.

Laravel's wrapper around Guzzle focuses on its most common use cases and on an easy-to-use developer experience.

Localization:

The localization features of Laravel provide a convenient way to retrieve strings in multiple languages, allowing us to easily support multiple languages in our application.

It offers two ways to manage translation strings.

Firstly, language strings can be stored in files in the **resources/lang** directory.

In this directory, there may be subdirectories for each language supported by the application. This is the approach that Laravel uses to manage translation strings for Laravel built-in resources, such as validation error messages.

The other way is translation strings that can be defined in JSON files that are placed in the resources/lang directory.

By adopting this approach, each language supported by our application would have a corresponding JSON file in this directory. This approach is recommended for applications that have a large number of translatable strings.

Mail:

Sending emails doesn't have to be complicated, so Laravel provides a clean and simple email API developed by the popular **SwiftMailer library**.

Laravel and SwiftMailer provide drivers for sending emails via SMTP, Mailgun, Postmark, Amazon SES, and sendmail, allowing us to quickly start sending emails through a local or cloud-based service of our choice.

Notifications:

In addition to support for email sending, Laravel offers support for sending notifications through a variety of delivery channels, including **email, SMS (via Vonage, previously known as Nexmo), and Slack.**

In addition, a variety of community-created notification channels are available as packages and can provide notification deliveries for dozens of different channels.

Notifications can also be stored in a database so that they can be displayed in our web interface.

Notifications should typically be short and informative messages that notify users about something that has occurred in our application. **For example**, if we are writing a billing application, we might send a “Invoice Paid” notification to our users through email and SMS channels.

Package Development:

Packages are the primary way to add functionality to Laravel by extending its native functions.

Packages can solve any software problem, **from a great way to work with dates like Carbon to a package that allows us to associate files with Eloquent models like the Spatie Laravel Media Library.**

There are different types of packages. Some packages are independent, meaning they work with any PHP framework. **Carbon and PHPUnit are examples of independent packages.** Any of these packages can be used with Laravel, just include them in our **composer.json** file’s requires.

On the other hand, other packages are specifically planned for use with Laravel. **These can have routes, controllers, views, and configurations specifically aimed at enhancing a Laravel application.**

All of this means that we can write packages with reusable **logics or modules** within our company, or even distribute them for this within the community.

Queues:

When building our web application, we may have some tasks that take a long time to execute during HTTP requests, such as parsing and storing a CSV file.

Laravel enables us to create “delayed” execution queues, allowing us to create queued jobs that can be processed in the background.

By moving time-consuming tasks to a queue, our application can respond to web HTTP requests at high speed and provide a better user experience to our customers.

Laravel’s queues provide a unified queuing API across a variety of different queue back-ends, such as Amazon SQS, Redis, or even a relational database.

The Laravel queue configuration options are stored in our application’s **config/queue.php** configuration file.

In this file, we will find connection settings for each of the **queue drivers included in the framework, such as database, Amazon SQS drivers, Redis, and Beanstalkd**, as well as a synchronous driver that will execute jobs immediately (for use during local development).

A null queue driver is also included, which can be used in testing and discards queued jobs.

Task Scheduling We may have written a cron configuration script for every task that needed to be scheduled on our server. However, this can quickly become complex to manage because our task scheduling isn’t in the same source control location as our system, so we have to use **SSH** on our server to view our existing **cron** entries or add additional entries.

The Laravel command scheduler offers a new approach to managing scheduled tasks on our server.

The scheduler allows us to fluently and expressively define our command schedule within our own Laravel application.

By using Laravel’s scheduler, only a single cron entry is required on our server.

This way, our schedule of periodic execution tasks is defined in the **scheduling method** of the **app/Console/Kernel.php** file.

Automated testing:

Laravel was built with testing in mind.

In fact, **support for testing with PHPUnit is included “out of the box” in the phpunit.xml file, and is already configured for our application.**

The framework also provides convenient helper methods that allow us to expressively test our applications.

By default, our application’s **test directory contains two subdirectories: Feature and Unit.**

Unit tests are tests that focus on a very small and isolated part of our code. In fact, **most unit tests probably focus on a single method.** Tests within our “Unit” test directory do not initialize our Laravel application and therefore cannot access our application’s database or other framework services.

Resource tests - Feature - can test a larger part of our code, including how various objects interact with each other or even a complete HTTP request to a JSON endpoint.

Generally, most of our tests should be resource or system functionality tests. **These types of tests provide the greatest confidence that the system as a whole is working as expected.**

Final considerations

The Laravel framework is an excellent addition to any **PHP-based web application development stack.**

It presented the most common components in the use of applications developed with Laravel, as well as some of the advanced components that can facilitate the scalability of our systems without necessarily requiring us as a developer to implement all the necessary complexity of logic for this.



Folder Structure for Modern Web Applications

It is critical to create a maintainable folder structure while developing web apps, having the right files in the correct folder helps organize your code and makes other developers have an idea of how the architecture of your web application is or will be during development. In this post, I am going to explain some folder names when building your modern web project.

Maintaining a well-organized folder structure is crucial when developing web applications, even though it may not be the first thing that comes to mind when working alone or with few resources. If not, you run the risk of coming across as unprofessional.

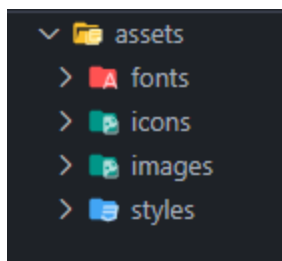
Some Tips In Designing Your Folder Structure

- Understand the purpose of your web project: In order to figure out how to organize your web project, you will need to establish a good understanding of what you have, depending on how many assets you are trying to organize and the features in your web applications.
- Use proper naming convention for your folders and files, they should be descriptive of the purpose in your web application.

Folder Structures and their explanation

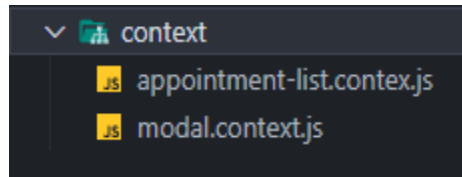
Assets

The assets folder contains all images, icons, css files, font files, etc. that will be used in your web application. Custom images, icons, paid fonts are being placed inside this folder.



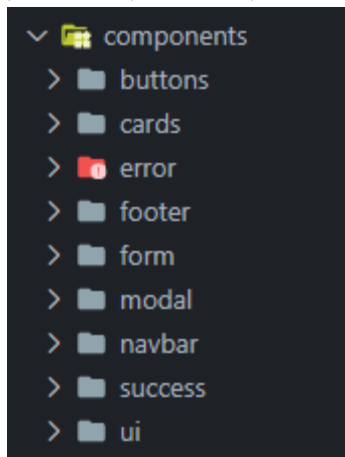
Context

When using React Js as your preferred frontend ui library, the context folder stores all your react context files that are used across components and multiple pages.



Components

The components folder holds the UI for your application. It contains all our UI components like navbar, footer, buttons, modals, cards, etc.



Composables

In the context of Vue applications, a "composable" is a function that leverages Vue's Composition API to encapsulate and reuse stateful logic.

Data

The data folder is used for storing text data which will be used in different sections and pages as JSON files. Doing this will enable updating of information easier.

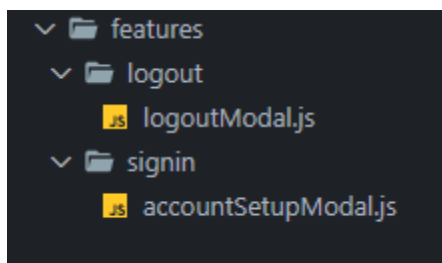


```
data
└─ navigation.json

navigation.json > ...
{
  "navigation": [
    {
      "name": "Home",
      "path": "/"
    },
    {
      "name": "About",
      "path": "/about"
    },
    {
      "name": "Services",
      "path": "/services"
    }
  ]
}
```

Features

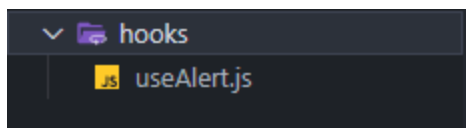
This folder contains individual folder feature for each page (authentication, theme, modals). For example each page might have a modal feature.



```
features
logout
└─ logoutModal.js
signin
└─ accountSetupModal.js
```

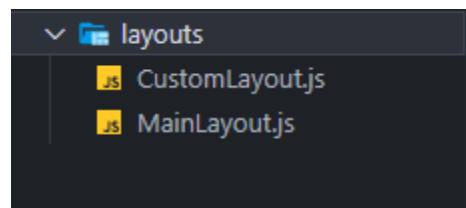

Hooks

Hooks are functions that let you “hook into” React state and lifecycle features from function components. Also we can create custom hooks whose name starts with 'use' and can be used to call other hooks.



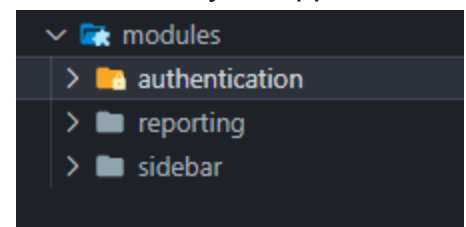
Layouts

When defining the general look and feel of the web page, the Layouts folder comes in handy. It is used to place layout-based components such as the sidebar, navbar, and footer. If your web application has many layouts, this folder is a fantastic place to save them.



Modules

Modules folder handles specific tasks in your application.



Pages

The pages directory contains your web application views. Pages directory in frontend frameworks like Next Js and Nuxt Js reads all files inside the directory and automatically creates the router configuration for you.

Public

The public directory is directly served at the server root and contains public files that won't change e.g favicon.ico.

Routes

The routes folder is just a place in your web application to store the routes path to different screens.

Utility/Utils

This folder is for storing all utility functions, such as auth, theme, handleApiError, etc.

Views

Views folder are like the pages folder, The views are used to represent your pages properly, that users can navigate back and forth.

Conclusion

A good folder structure allows you and other developers to find files faster and manage them more easily. A well-organized folder structure makes you appear professional.