Flutter core template

# Core

## Creating the core of our Flutter apps.

## Introduction

Creating the core is a very important step because we are now laying the foundation upon which we will build our house. It is crucial to make it as good and simple as possible. My core app will include the following steps in order to create a strong foundation:

1. Project Structure
2. Project Architecture
3. Dependency Injection (DI)
4. Routing System
5. App Theme

6. Localization
7. Network Communication

These tasks must be executed correctly. So, let's get started.

## Project Structure

Template App , is organized as a modular Flutter project. The app is broken down into different folders for better organization and maintainability, with each folder representing a distinct area of functionality. Here's a quick summary of each folder and what it does:

1. `exceptions`: This directory consists of classes that handle exceptions within the app. `AppExceptions` class carries information about the error, while `Catcher` class listens for exceptions and logs them.
2. `config`: Here, you've `AppEnvironment` class that manages application URLs depending on the environment (development or production).
3. `assets/localizations`: This folder contains `LocalizationsStrings` class which is used for defining string constants for localization, such as different messages displayed in various parts of your app.
4. `modules`: This is a vital part of your application where all the business logic resides. It's divided into different modules (like home and auth), each with its own separate pages and logic. Each module is set up using `flutter_modular`, which provides the benefits of Dependency Injection and Route management.
5. `routes`: This is like a roadmap for your app. It helps decide which screen (or page) to show to the user based on their actions or status. It's like giving directions on where to go within the app.
6. `route guards`: This part works like a security guard for your app. It checks whether a user should be allowed to go to a certain page or not. For example, it might stop someone who isn't logged in from getting to a page that's only for logged-in users.
7. services:This is where shared logic across the entire application lives. It houses elements that are used across multiple parts of the application, making it a very important part of your project structure.

8. theme: This folder takes care of the visual style of the application. Here, we find the AppTheme class, a central hub that broadcasts the visual style elements like colors and text styles to all parts of the application
9. network : The Network folder plays a critical role in your app, as it is the gateway between your application and the outside world. This is where your app communicates with the internet, fetching and sending data to remote servers. Two key components in this folder are the Network Client and the Interceptors.

1. **Network Client:** This is the workhorse of your network operations. Think of it as your app's personal courier, responsible for delivering and retrieving data from your server. It handles all network requests - whether it's getting user data, posting updates, or anything else that requires communication with your server.
2. **Interceptors:** These are the gatekeepers of your network operations. They monitor and transform HTTP requests and responses coming in and out of your app. They can be used to add headers, transform the data, handle errors, and more. For example, you could add an interceptor to attach an authentication token to every request, ensuring that all your network calls are authorized.

Together, these components ensure that your network operations are efficient, secure, and error-free. With these in place, your app can effectively communicate with servers and provide a smooth user experience.

Overall, this template is designed in a clean and modular manner, separating different concerns into their own modules and components, which would make the application easier to maintain and scale.

## Project Architecture

Designing an app's architecture effectively calls for smart structuring. A great way to achieve this is by combining the Model-View-ViewModel (MVVM) design with a modular architecture. At the core, we have modules like AppModule, HomeModule, and AuthModule. These are the main building blocks that manage our dependencies, ensuring the app is neatly organized. The MVVM pattern is then implemented, promoting a clear divide between business and presentation logic, which simplifies testing and maintenance.

Services handle tasks like calculations or server communications, keeping operations encapsulated. Repositories manage data storage and retrieval, acting as a link between the app and data sources.

ViewModels (VMs) tackle the business logic, mediating between the View and Model. This clean separation keeps the business logic away from the UI, simplifying debugging.

Lastly, we use Stores for quick, in-memory data access, improving the app's performance.

In a nutshell, combining MVVM with a modular approach offers a robust, scalable design that's easy to maintain and delivers great performance. It's an efficient strategy for developing superior apps.

## Dependency Injection (DI)

In this project, we will be using a Flutter package called Modular. This package will greatly assist us in managing different parts of our app and simplifying the sharing of resources between them.To better understand these different parts, we can think of them as small "modules". Each module functions as a mini app in itself, and Modular facilitates their seamless collaboration. For our project, we have a main module called AppModule, which acts as the primary module with control over elements accessible throughout the app. You can locate its definition in app_module.dart, though the exact location may vary depending on your project's structure. Within AppModule, we create additional smaller modules such as HomeModule and AuthModule using the ModuleRoute class. Let me explain how it's done:

1. **AppModule**: It is the root module of your application, sort of like the trunk of a tree where all branches (other modules) stem from. In `AppModule` you register things (called dependencies) which are needed across the entire app. This is done using

the `binds` list in the module class:

```
You, 3 hours ago | 3 authors (vlmoon99 and others)
class AppModule extends Module {
  @override
  final List<Bind> binds = [
    Bind.singleton((i) => Catcher(i())),
    Bind.singleton((i) => NetworkChecker()),
    Bind.singleton((i) => const FlutterSecureStorage()),
    Bind.singleton((i) => AppTheme()),
    Bind.singleton((i) => InactivityService(i())),
  ];

  @override
  final List<ModularRoute> routes = [
    ModuleRoute(
      Routes.home.module,
      module: HomeModule(),
    ), // ModuleRoute
    ModuleRoute(
      Routes.auth.module,
      module: AuthModule(),
    ), // ModuleRoute      vlmoon99, 9 months ago • init project
  ];
}
```

2. **AuthModule**: AuthModule is a child module. It handles everything related to authentication (logging in, signing up, etc.)

```
You, 1 second ago | 3 authors (vlmoon99 and others)
class AuthModule extends Module {
  @override
  final List<Bind> binds = [];

  @override
  final List<ModularRoute> routes = [
    ChildRoute(
      Routes.auth.login,
      child: (_, args) => const LoginPage(),
    ), // ChildRoute
  ];
}
```

3. **HomeModule**: Similar to `AuthModule`, `HomeModule` is a child module that handles everything related to the home screen of the app. It might look something like this:


Once all the bindings in the modules have been defined, we can access our dependencies in widgets and other classes using the Modular class. This class provides a convenient way to retrieve instances of our dependencies.

To access a specific dependency, we use the `Modular.get<T>()` method, where `T` represents the type of the dependency we want to retrieve. For example, if we have a `SomeViewModel` class that we want to access, we can use `Modular.get<SomeViewModel>()`.

## Routing System

In our core Flutter application, we use a system referred to as "routing" to navigate between different screens, otherwise known as "pages". This routing system functions as a map for our app, indicating how we can navigate from one page to another.

Our routes are neatly organized in a dedicated place, a file named `routes.dart`. This file acts as the central hub for all routes within the application. Within this file, we have a class named `Routes` that lists all the potential pages in our app, such as 'home', 'auth', and so forth.

Each of these routes is connected to a specific 'module'. Think of a module as a compact bundle of related pages and associated functions. For instance, everything pertaining to 'authentication' (like login and register) is bundled together in the `AuthModule`.

To navigate to a new page, we just call a straightforward function. We use `Modular.to.navigate(Routes.home.getModule())` to go to the home module's starting page or `Modular.to.navigate(Routes.auth.getModule())` to go to the auth module's starting page. It's really that simple! Do remember, however, that when `Modular.to.navigate` is called, it will clear all your previous paths. If you want to retain your previous page in memory, use `Modular.to.pushNamed(Routes.auth.getModule())`.

In this illustration, I've only shown you how to navigate between modules. Here's an example of how to navigate inside a module: `Modular.to.pushNamed(Routes.auth.getRoute(Routes.auth.login))`. Therefore, every time you navigate in your app, please consider these details about where you're going and how the path is created (whether or not it clears previous routes).

## App Theme

In our app, we use a smart way to handle colors and text styles called `AppTheme`. It holds all the theme data, such as colors and text styles, and sends them out to the app using a 'stream'. This way, when the theme data changes, every part of the app knows about it immediately.

We can use these themes easily in our pages. For example, in the `HomePage` or `LoginPage`, we get our `AppTheme` by calling `Modular.get<AppTheme>()`, and use it to style our widgets. So, `colors.customColor` gives us a special color from our theme, and `textTheme.error` gives us a style for titles.

We also can create many themes (like light and dark) and switch between them easily. We just update the theme data in `AppTheme`, and the new theme is automatically applied everywhere in the app.

In short, `AppTheme` gives us an easy and flexible way to make our app look great and adapt to user preferences.

## Localization

Localization in our app is handled by the `LocalizationsStrings` class. It's like a dictionary for all the words and phrases that our app uses. This means we can change the language of our app just by switching to a different dictionary.

Here's how it works:

In `LocalizationsStrings`, we have different sections like `_Auth`, `_Home`, `_Login`, etc., each for a specific part of our app. In each section, we define terms we use in that part of our app. For example, in `_Login`, we have terms like `title`, `forgotPassword`, `registerNewAccount`.

When we need to display a text in our app, we use these terms instead of writing the text directly. So, if we want to show the title of the login page, we use `LocalizationsStrings.auth.login.title`.tr()

Now, imagine we want to switch our app to a different language. Instead of going through all our pages and changing the texts, we just update our `LocalizationsStrings` dictionary with the new language, and all the texts in our app get updated automatically!

This way, our app can support multiple languages easily and efficiently. It makes our app more user-friendly and accessible to people from different parts of the world.

## Network Communication

1. `NetworkClient`: This class is responsible for managing network requests using the Dio package. It initializes Dio with the provided network configuration and handles different types of HTTP requests (GET, POST, PUT, DELETE). It also includes error handling for Dio errors and returns an `ApiResponse` object with the result of the request.

2. `ApiResponse`: This class represents the response of a network request. It has properties to store the response data, status code, and whether the request was successful or not.
3. `NetworkResource`: This is an abstract class that defines the configuration for the network resource. It includes properties such as the base URL, interceptors, and timeout values.
4. `NetworkChecker`: This class handles connectivity status monitoring using the `Connectivity` package. It listens for changes in the connectivity status and updates the `connectionStatus` property accordingly.
5. `LoggingInterceptor`: This class is an interceptor for the Dio package that logs the request and response information. It overrides methods for onRequest, onResponse, and onError to log relevant details.

These classes work together to provide network functionality, handle errors, manage connectivity, and log network-related information. They are essential components for building a robust networking layer in an app.