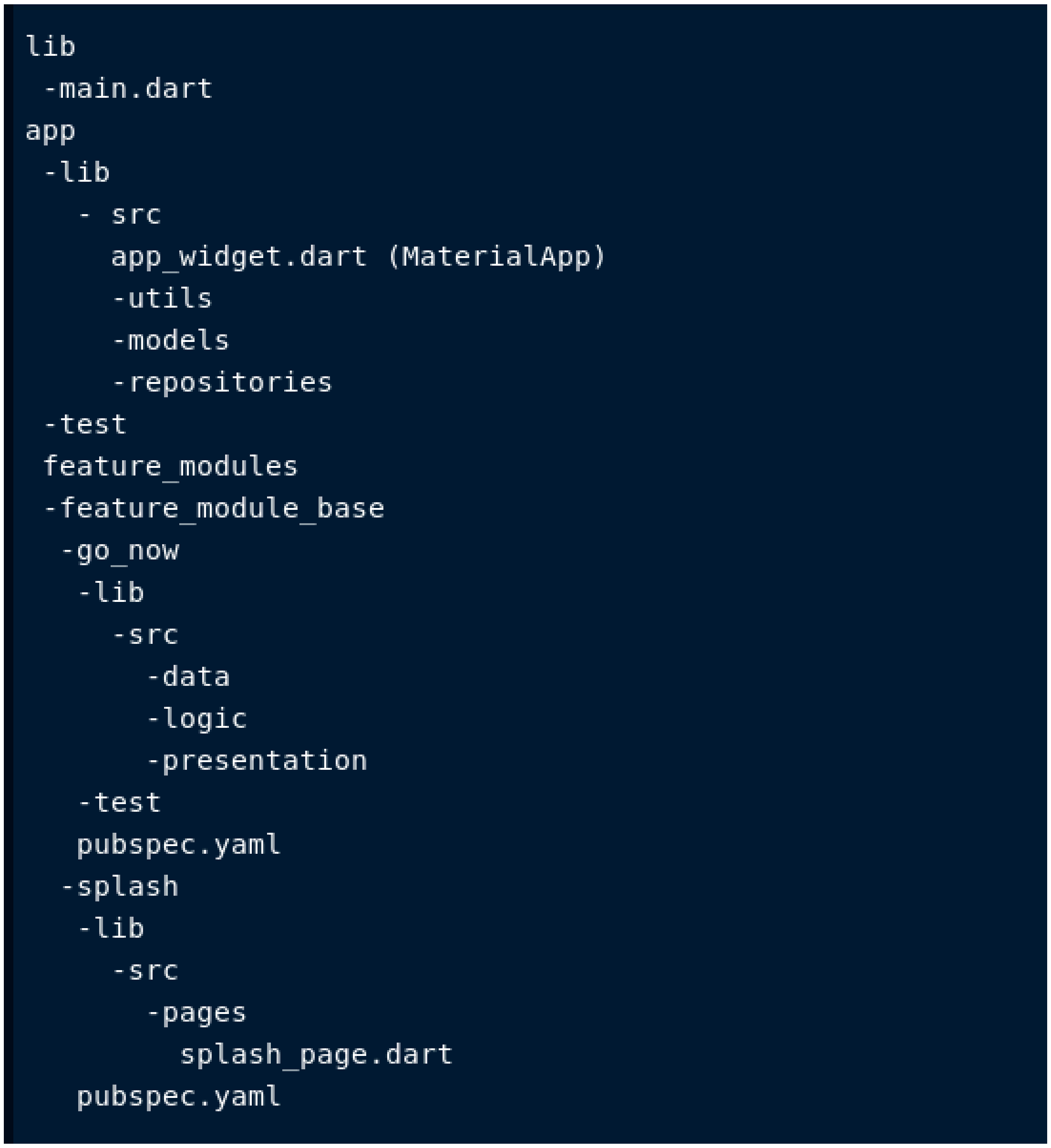
Package-Based Modular Architecture

(or **Feature-Based Modular Architecture**)

Example:



```
lib
 -main.dart
app
 -lib
   - src
     app_widget.dart (MaterialApp)
     -utils
     -models
     -repositories
 -test
 feature_modules
 -feature_module_base
  -go_now
   -lib
     -src
       -data
       -logic
       -presentation
   -test
   pubspec.yaml
  -splash
   -lib
     -src
       -pages
         splash_page.dart
   pubspec.yaml
```

The Flutter structure used, where each module is encapsulated as a separate Dart package created using the flutter create --template=package command, is commonly referred to as a **modular architecture** or **package-based modular architecture**.

This structure provides a clean way to separate concerns and manage dependencies, making the application scalable and maintainable. Here's a breakdown of the technical details:

Key Technical Terms for This Structure:

1. **Modular Architecture:** This approach divides the app into independent, self-contained modules that handle specific features. It makes the app more maintainable and allows for easier scaling as each module is isolated and can be developed independently.
2. **Feature-Based Modularization:** Organizing the code based on different features of the app (like auth, profile, splash) rather than by technical layer (like UI, business logic, data layer). This allows for better organization and reusability.
3. **Flutter Package Template:** When creating a package using flutter create --template=package, you generate a Flutter package that can contain Dart code and resources. These packages can then be used as dependencies by other parts of your app or even in other apps.

Key Characteristics

- **Encapsulation**
  - Each feature or logical module (e.g., `go_now`, `splash`) is encapsulated in its own Dart package. This allows for isolation and reuse.
- **Independent Packages**
  - Each module is a standalone package with its own `lib` directory and `pubspec.yaml`.
  - The `pubspec.yaml` allows you to define specific dependencies for each module.
- **Centralized App Layer**
  - The `app` directory serves as the central hub where shared utilities, models, and widgets reside.
  - The `app_widget.dart` (or `MaterialApp`) is the entry point for combining the modules.
- **Reusability**
  - Since each module is a package, it can be reused across multiple projects.

- **Dependency Management**
  - The main `pubspec.yaml` file in the root of the project includes all modules as dependencies, either by path or hosted location (if published).

Example of `pubspec.yaml` in Root

```
dependencies:
  flutter:
    sdk: flutter
  app:
    path:app
```

How It Works:

Here is a more detailed explanation of the structure and its terminology:

1. App Package

This is the main entry point of the application, often referred to as the **core app package**. It contains:

- **main.dart**: The entry point of the Flutter application.
- **app_widget.dart**: This is where you define the MaterialApp (and/ other app-level configurations like themes, routes, etc.).
- Other app-wide configurations or utility classes might also live here.

The **app package** serves as the foundation and is responsible for setting up app-wide services, routing, and other global logic.

2. Feature Modules (Packages)

These are Flutter packages that encapsulate a specific feature or set of related functionality. Each feature module can be independently developed, tested, and maintained. The modular approach helps in making the app more maintainable and scalable. For example:

- **go_now**: The main screen of application ().
- **splash**: The splash screen feature.
- **profile**: The user profile feature.

Each feature module can contains:

- **lib** folder: Contains Dart code for the feature.
- **pages**: Flutter widgets representing different pages/screens for this feature.
- **repositories**: Business logic and data repositories for this feature.
- **widgets**: UI components specific to this feature.

The feature modules can have their own `pubspec.yaml` files, which allow them to have independent dependencies and configurations.

Advantages

- **Scalability:** Each module can grow independently without affecting the others.
- **Separation of Concerns:** Helps enforce boundaries between different parts of the application.
- **Testing:** Modules can be tested independently.
- **Team Collaboration:** Different teams can work on different modules simultaneously.
- **Ease of Maintenance:** Isolated changes to a module minimize risks to the entire project.

When to Use

- Large applications require clear separation of concerns.
- Projects that might expand in scope or involve multiple teams.
- When you want to reuse modules across projects.

Integration with State Management

When combined with **BLoC** or other state management solutions:

- Each module independently manages its state.
- Facilitates clean and modular state handling.

This structure is similar to the **"Clean Architecture"** or **"Layered Architecture"** but with an emphasis on feature isolation using independent Flutter packages. It is also commonly used in large-scale enterprise applications to ensure separation of responsibilities.