CSC 431
Roll Companion
System Architecture Specification (SAS)

Team 4

Alicia Bilbao	Scrum Master
Robert Bolton	Developer
William Blair	Front End Developer

Version History

Version	Date	Author	Description
1.0	25-March-2023	Alicia	Initial Draft
2.0	04-April-2023	Alicia	Structural Design
3.0	06-April-2023	William	System Analysis
4.0	06-April-2023	Robert	Functional Design
5.0	06-April-2023	Alicia	Structural Design, Formatting

Contents

List of Figures

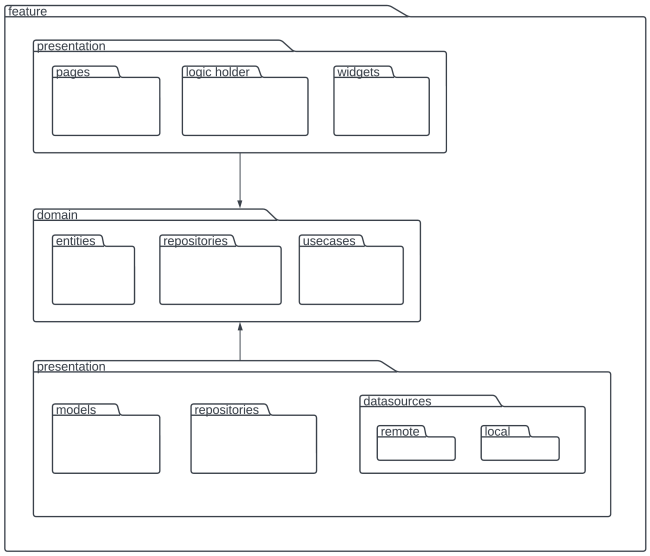
1 System Analysis

1.1 System Overview

Roll Companion is a cross-platform mobile application designed to accelerate the progression and technical proficiency of Brazilian Jiu Jitsu (BJJ) practitioners. The application primarily focuses on assisting novices in overcoming the initial learning curve but also offers valuable insights for experienced fighters. It achieves this by offering a method to track training progress, set goals, and provide a macro view of the sport.

The app utilizes a Clean Architecture, proposed by Robert C. Martin (Uncle Bob), to ensure a maintainable, scalable, and testable codebase.

1.2 System Diagram

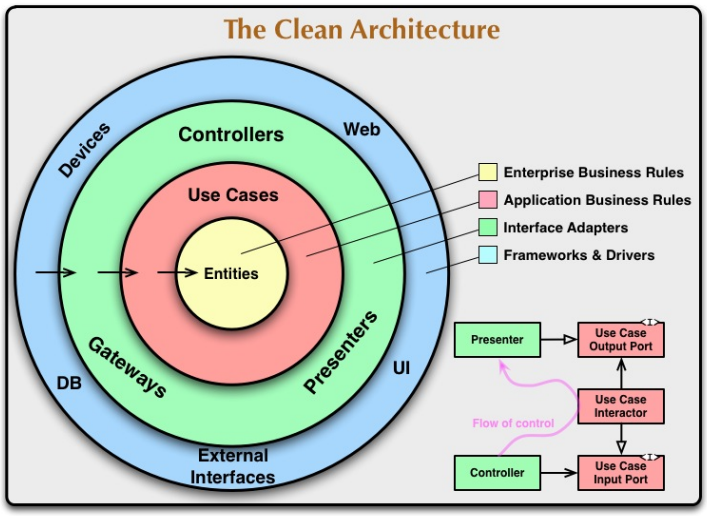


1.3 Actor Identification

- User: A user is someone interested in improving their Brazilian Jiu Jitsu skills, anyone from beginners to advanced practitioners.
- System: The app itself is another actor, relying on the user, and consisting of different features such as a graph neural network, and a video library.

1.4 Design Rationale

1.4.1 Architectural Style



Clean Architecture provides the following:

- Separation of Concerns: Clean Architecture promotes the separation of concerns, which helps to create a more organized and modular codebase. Each layer has a specific responsibility, and the layers are loosely coupled, which makes the code more maintainable and testable.
- Testability: It makes it easier to write automated tests, as the different layers are isolated and can be tested independently. This improves the quality of the software and reduces the risk of introducing bugs.
- Flexibility: Clean Architecture promotes a modular design that allows developers to swap out components or change the implementation details without affecting the rest of the system. This makes it easier to adapt to changing requirements or to introduce new features.
- Independence from external frameworks: It makes it possible to switch out external frameworks or libraries without affecting the rest of the system. This reduces the risk of being locked into a particular technology or vendor.
- Maintainability: It helps to create code that is easier to understand and maintain. By separating concerns and using clear interfaces between components, developers can work on different parts of the system without affecting other parts.

The Layers

Clean Architecture is built around the *Use Cases* of the application, as they should be independent of implementation details and do not change unless the purpose and objective of the application changes. Thus they are at the centre, in the domain layer, of the architecture.

Domain Layer

The domain layer encapsulates *Enterprise Business Rules* through entities. An entity can be an object with methods, or it can be a set of data structures and functions. They encapsulate the most general and high level rules, and are the least likely to change when something external happens. Such as:

- A user
- A technique video... comprised of:
- Steps
- Instructor
- Topic

Domain services (logic that expands across multiple domain models) and data abstractions such as interfaces for repositories and units of work also reside in the domain layer. Only abstractions! These are ports that are expected to be implemented by adapters in the infrastructure layer.

Use Cases are classes which encapsulate all the business logic of a particular use case such as Sign-Up, Login, View Graph, Add Friends,... etc. These use cases do not change and are independent of implementation details.

Data Layer

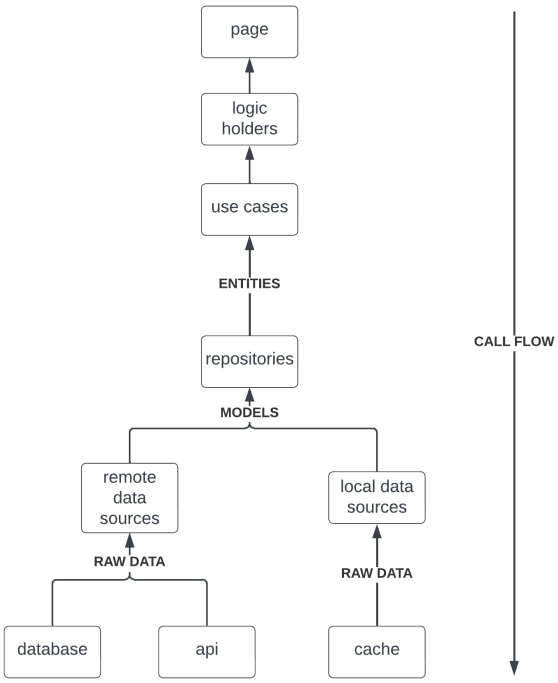
The data layer consists of a Repository implementation (the contract comes from the domain layer) and data sources - one is usually for getting remote (API) data and the other for caching that data. Repository is where you decide if you return fresh or cached data, when to cache it and so on.

Data sources return models, that extend the entities in the domain layer. These models contain methods that convert data from specific formatted code such as JSON, into the domain entity formats. We would use the models to transform data from firebase, hive, etc into the base entities.

Presentation

Within the context of the flutter framework, we will be working with 'widgets' and BLoC (Business Logic Component) as the logic holder. The widgets display graphics to the screen, and dispatch events to the BLoC and listen for state changes. Note, the logic holder here delegates all its work to the use cases; at most it handles basic input conversion and validation.

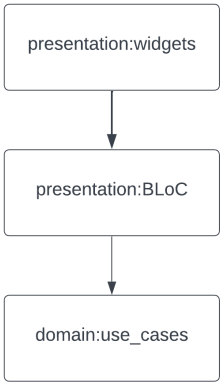
Clean Architecture Call Flow



1.4.2 Design Patterns

BLoC Pattern (Presentation Layer)
Components:

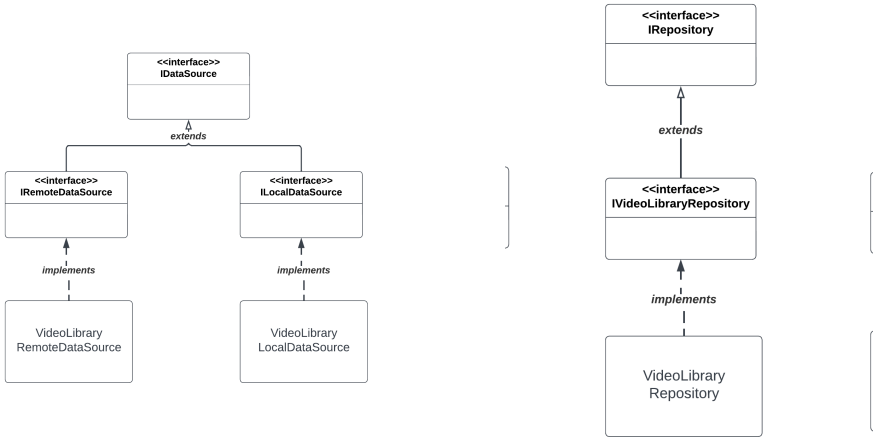
- Views / Pages (UI)
- BLoCs (Logic Holders)

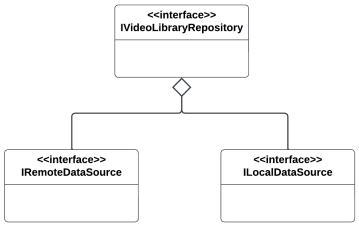


Views listen to state updates from BLoCs and send user interactions to BLoCs. BLoCs handle business logic and emit state updates.

Repository Pattern (Data Layer)
Components:

- Repositories
- Data sources (Local & Remote)





Repositories use Data Sources to access and manage data. Use Cases depend on Repositories to access data.

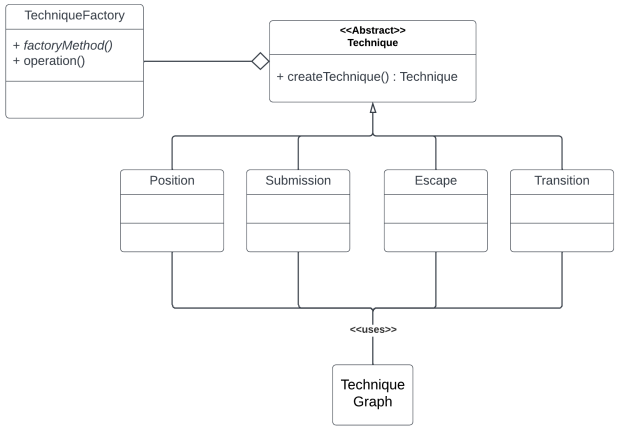
Dependency Injection (Throughout app)
Components:

- All components that depend on other components or services (e.g., BLoCs, Use Cases, Repositories)
- Data sources (Local & Remote)

Dependencies are injected into components instead of being instantiated directly.

Factory Pattern (Domain & Data Layer)
Components:

- Factory
- Technique Classes (Position, Submission, Escape, Transition)



The TechniqueFactory is responsible for creating instances of Technique subclasses based on input parameters. The arrows represent the relationships between the Factory, the abstract base class Technique, and its concrete subclasses (Position, Submission, Escape, Transition). This is useful as the graph feature will use loads of different techniques for its nodes!

Singleton Pattern (Appropriate Components)
Components:

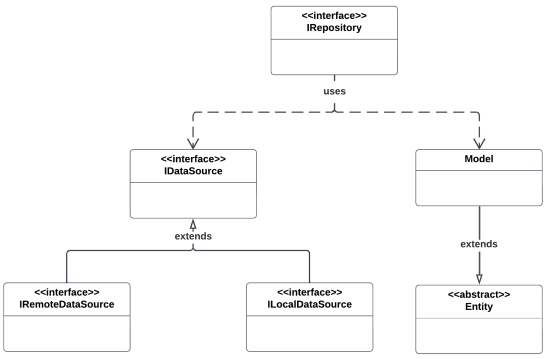
- Singleton Classes (e.g., Database Connection, GraphService)

Only one instance of a Singleton class exists throughout the app's lifecycle.

Adapter Pattern (Data Layer)
Components:

- Adapters (Models)
- Data Sources
- Domain Entities

Models extend the entities in the Domain Layer and include additional methods for converting data between external formats (e.g., API response, Firebase document) and the internal domain format. Data Sources retrieve data from various sources (e.g., API, Firebase) and return the data in the external format. Repositories depend on both Models and Data Sources. They use Data Sources to fetch data in the external format and then use Models to convert the data to the internal domain format before passing it to the Domain Layer.



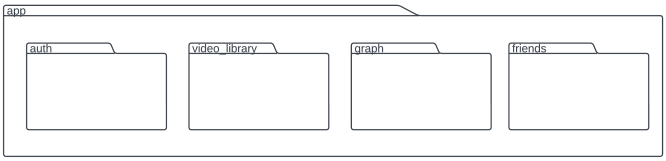
The Adapter Pattern is applied through the Model class, which extends the Entity class and includes additional methods to convert data between external formats and the internal domain format. The Repository class uses DataSources to fetch data in external formats and then uses the Model class to convert the data into the internal domain format before passing it to the Domain Layer.

1.4.3 Framework

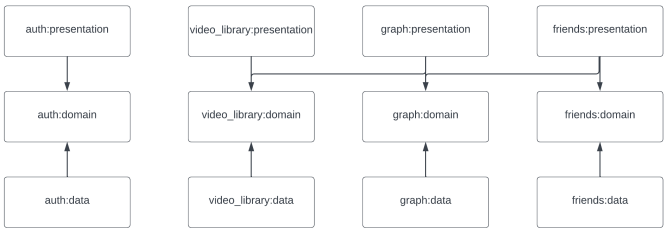
Roll Companion will be using the Google's opensource Flutter Framework. It enables building beautiful, natively compiled, multi-platform applications from a single codebase.

Modularisation within the Flutter Project

In order to achieve high cohesion and low coupling throughout the system, the project will by organised by feature-by-layer.



Each feature will have implement the clean architecture. Here you can see the dependencies between the features.



2 Functional Design

2.1 Add Skill

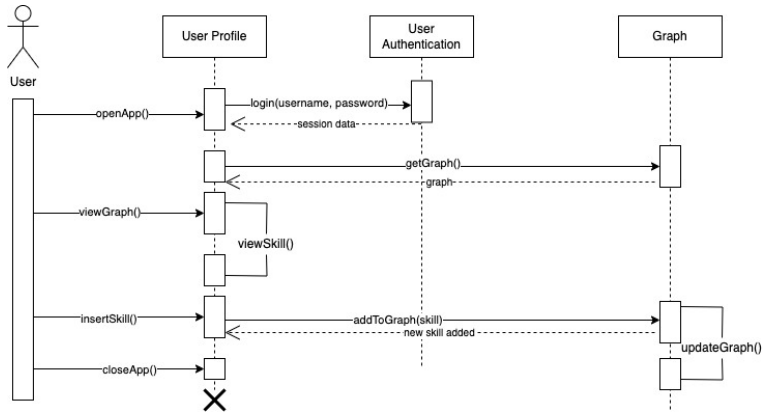


Figure 1: User inserts a new skill

2.2 Friend Request

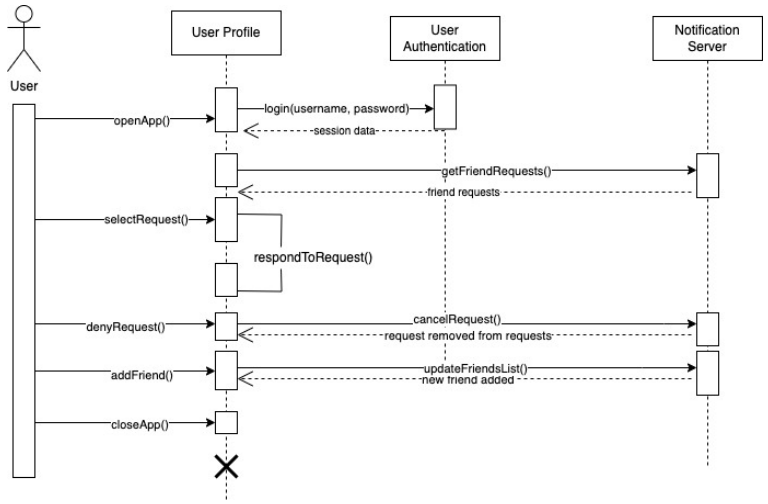


Figure 2: User has a friend request

2.3 View Video

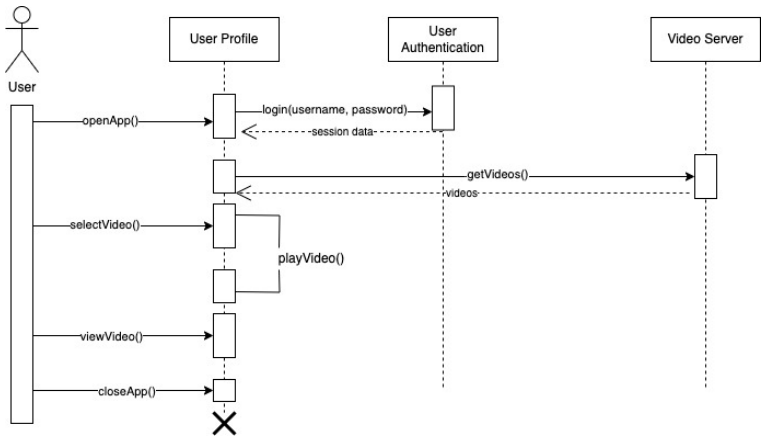


Figure 3: User views a video

3 Structural Design

3.0.1 Class Diagram

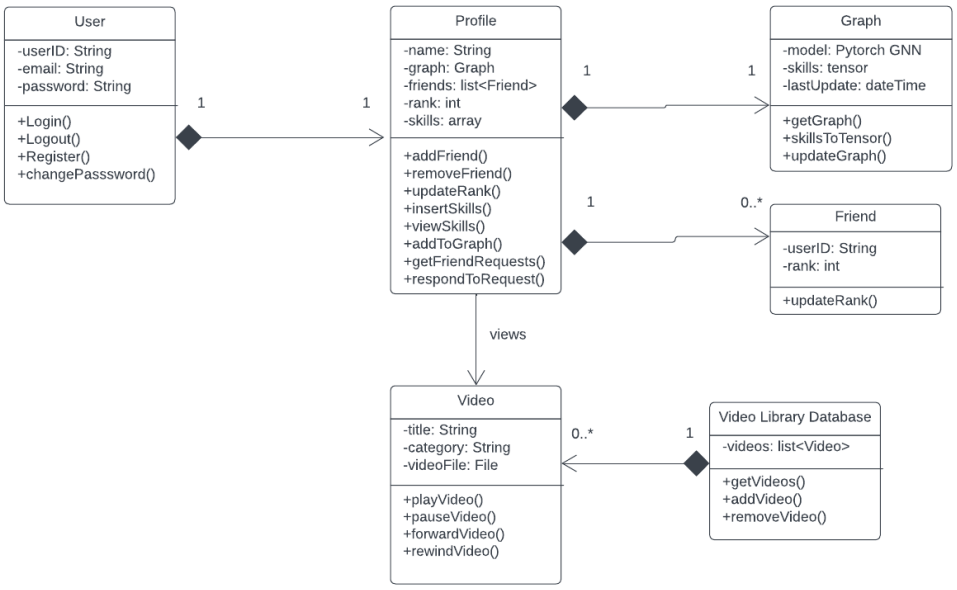


Figure 4: Class Diagram