# Eventify App - Requirements

## 1. Key Stakeholders

- **Project Manager:** Leads the project to create a solid product making sure all the requirements are met and customers are satisfied.

- **Product Owners:** Understands the domain of the project communicating with clients and specifies requirements in a clear manner that all the stakeholders can rely on.

- **Development Team:** Consists of developers and QA engineers. They are responsible for the software to be delivered by the estimation agreed on considering the feedback from the other stakeholders and their technological expertise.

- **Designers:** UI&UX designers to visualize a user-friendly product understanding user behaviors.

- **Customers:** Specifies project requirements working closely with relevant stakeholders and makes contracts to have a working product by a timeline planned before implementation phase.

- **End Users:** The people who will be using the app to discover, manage and attend events.

## 2. Functional Requirements

- Discover events: Browse events in a dashboard.
- Event details: See name, address and date of a selected event.
- Manage events: Create, edit and delete event through a menu.
- Attend events: Easily attend an event via details view.

## 3. Non-functional Requirements

**a) Performance**
- Lazy loading to be used in components such as ListViews.
- High size assets that are loaded at runtime to be optimized.
- Some views to be kept alive when navigating instead of loading its data over and over again.
- Any UI blocking operation to be profiled and optimized with asynchronous approach if possible.

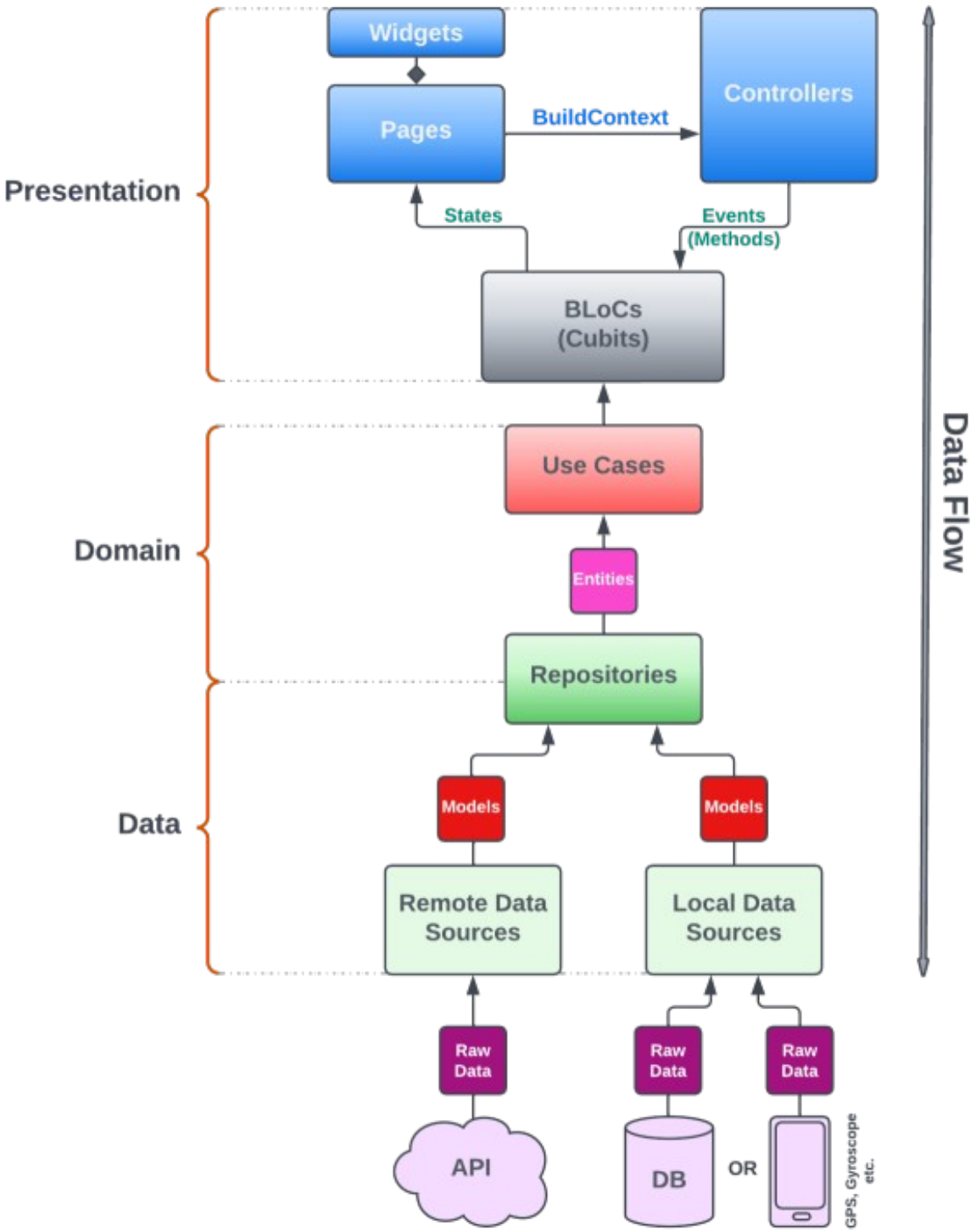
**b) Security**
- Any sensitive information like api keys, secrets and passwords to be remote controlled and not hard-coded.
- All the sensitive information received from a remote service to be stored in a secure storage.
- Code obfuscation to be applied if possible.

**c) Scalability**
- Architecture to be designed as scalable as possible as the project grows.

**d) Responsiveness**
- App to be responsive to different screens, operating systems, orientation etc.

## 4. Solution Design

### a) Architecture



**Layer Breakdown**

**1. Presentation**

Presentation layer consists of UI related elements such as Pages, Controllers, Widgets and Cubits.

A Controller is a bridge between UI interaction and Cubits. Whenever an event is triggered on UI (like a button tap), Controller is the element to execute the required methods in Cubits. Also, Controllers can do actions like navigating between pages, showing popups, accessing framework APIs through BuildContext.

Pages are basically Scaffolds that cover all the screen. They delegate their presentation logic to the corresponding Controllers by sharing BuildContext in creation. If any state is emitted from Cubits, Pages catch those states to rebuild their parts.

Pages and Controllers have 1-to-1 relationship, so they live and die together.

Widgets are reusable UI components with single responsibilities. They can be either stateful or stateless according to need.

Cubits are where state management happens. They are the interaction point to trigger Use Cases from Domain layer. After Use Case executions, states are emitted for Pages to re-construct the UI.

**2. Domain**

Domain is the only isolated layer of the architecture. It does not encapsulate or depend on any other layer, not even the Flutter framework. So, whatever changes in the other layers should not affect this layer.

It contains Use Cases, Entities and Repository contracts.

Use Cases are application business rules in short. They should contain all the business logic to define how the application behaves. They fetch & update data through Repositories. They can use multiple repositories.

Entities are the business objects. They are mapped from data models in Data layer, thus changes in data models should not affect them directly.

Repositories in Domain layer are just contracts(interfaces) that define how data will be fetched & updated in Data layer. This is one of the methods that makes Domain layer independent from the other layers. So, if anything changes in a Repository implementation in Data layer, it shouldn't affect Domain layer unless the contract itself changes.

**3. Data**

Data layer is the dirtiest layer in the architecture. It basically handles everything about data to be used in the application.

It defines data sources to work with raw data and serializes/deserializes raw data to data models, implements Repository contracts and maps data models to entities to be used in Domain layer, also decides when and how to handle data and error cases.

**b) Security Measures**

1. No data exchange to be made between client and server without authorization (via OAuth, JWT).
2. Secure connection to be executed with backend via https.
3. Api keys to be restricted to the app bundle id / package name with a hash key.
4. No sensitive user data at runtime to be logged remotely.

## 5. Open Items & Questions

- **Deployment:** Which deployment methods will be considered for the stages before production? App Center, Firebase App Distribution or any other provider?

- **Push Notifications:** Will we have push notifications to enhance user experience? If so, what service should we use? Firebase or any other service?

- **Monetization:** What is the method to monetize the app? In app purchases or ads?