Solution Architecture Document Template
Sri Krishna Mandir App

1. Project background – ACUITOLOGY has been entrusted by International Sri Krishna Mandir (ISKM) to develop a mobile application for the official Hare Krishna Temple to fulfill its mission of propagating Kṛṣṇa consciousness among its growing numbers of fellow members.

2. Core Technology Overview

2.1 Flutter

2.2 Angular CLI/REST JS

2.3 NodeJS

2.4 MongoDB

2.4 IP Camera & Streaming API

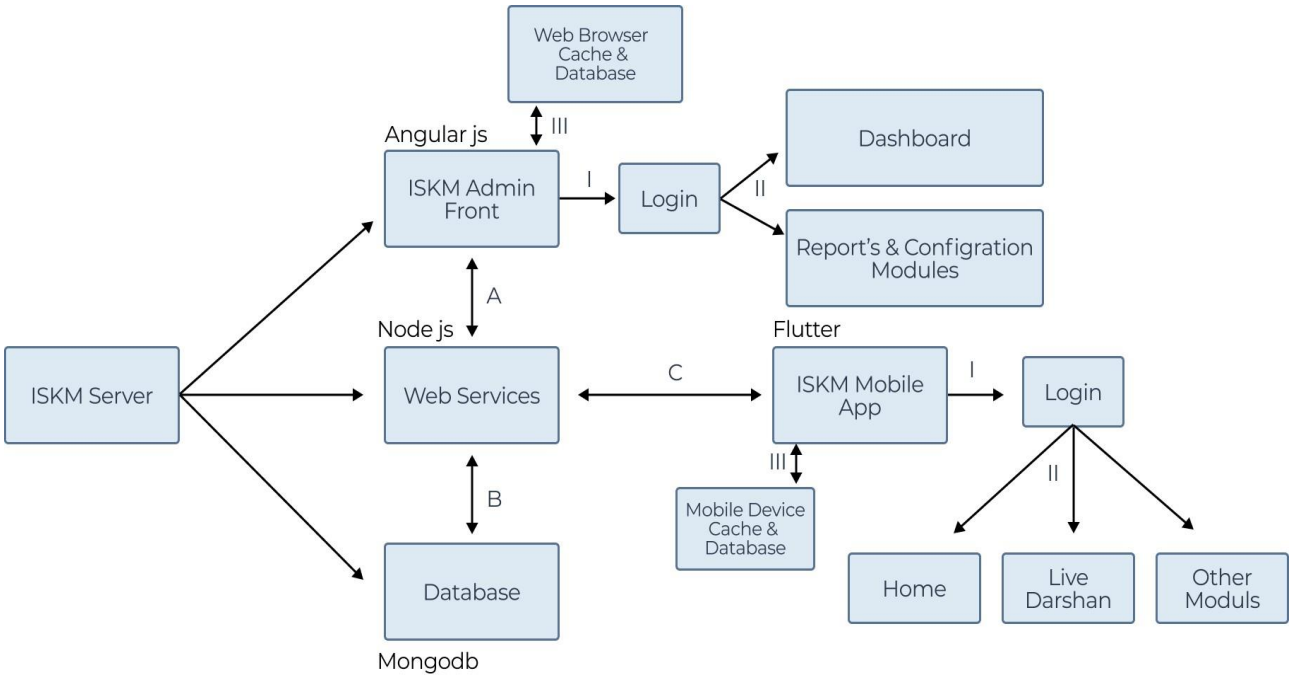
2.5 Zoom Meeting for Interactive classes and course

2.6 Google Firebase Cloud Messaging (FCM)

2.7 JSO Rules Engine with Visual Editor

2.8 Payment Gateway

3. Architectural Overview –



Control Flow of Architecture

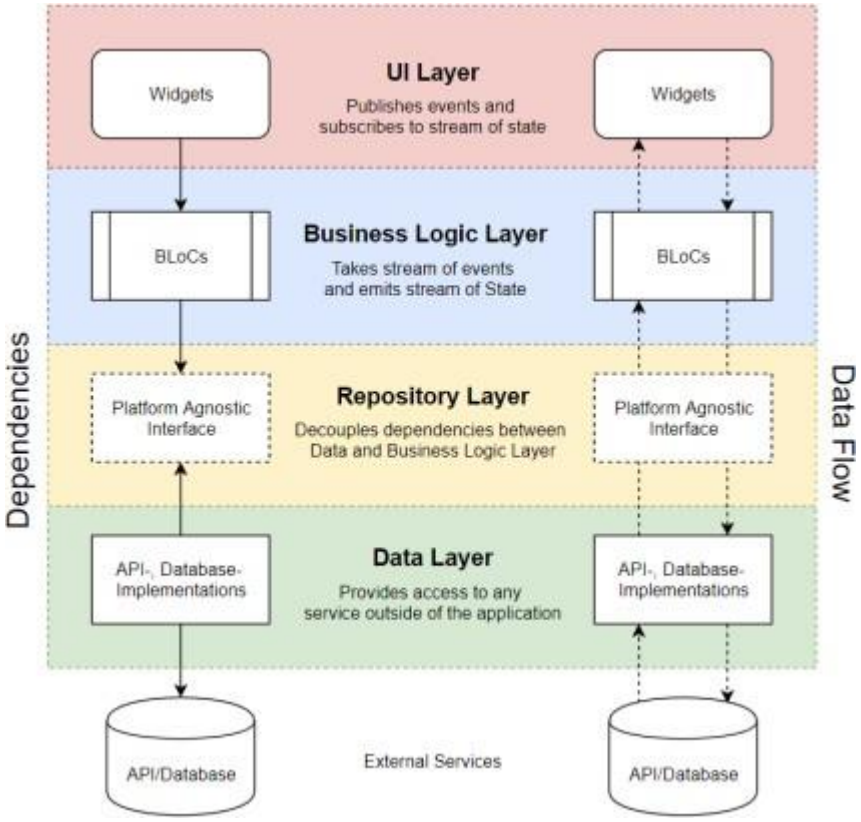
4. Mobile App Architecture & Design Best Practices

The layered architecture split an application into layer such that each layer is responsible for one aspect of the application. The goal is to keep these layers as independent as possible from one another by implementing the principle of one-way dependencies.

(1) Code with the same responsibility is in one location.

(2) Changes in one layer do not (or only slightly) affect the functionality of other layers.

Flutter Bloc Pattern and Layered Architecture



User Interface Layer – Consists of Flutter Widgets

Forms the user interface of the application. Contains as little business logic as possible. This layer takes in user inputs and processes them by emitting events to the BLoCs in the business logic layer. Displays the user interface based on the state of the BLoCs.

Business Logic Layer – Consist of Flutter BLoC

Contains the business logic of the application. All state changes are defined here. Consumes events that are emitted by the UI layer and mutates state based on those events. New state is then emitted to all classes that subscribe to it.

Can only communicate with the data layer through a platform-agnostic interface. The implementation of that interface must be injected into the BLoC

Repository Layer – Consist of Platform-agnostic interfaces

Decouples business logic layer and data layer.

Data Layer – Consist of External service/device implementations

Communicates with external devices or services. Every class in this layer must extend an interface from the repository layer.

Design of Widgets

Stateless Widgets- Stateless Widgets are the most basic type of widget. These widgets must be completely immutable which means that none of their values can change after initialization. As per the recommendations of the Flutter team, Stateless Widgets should make up the vast majority of a Flutter application to improve its performance.

Stateful Widgets- Stateful Widgets are responsible for holding the mutable state of a Flutter application. As per the recommendations, Stateful Widgets should be used as little as possible for performance reasons. There are primarily three reasons for why one would use a Stateful Widget over a Stateless Widget:

A. The widget needs to hold any kind of data that has to change during its lifetime.
B. The widget needs to do some sort of initialization at the beginning of its lifetime.
C. The widget needs to dispose of anything or clean up after itself at the end of its lifetime.

Rules for Widgets

A. Each “complex enough” widget has a related BLoC.
B. Widgets do not format the inputs they send to the BLoC.
C. Widgets should display the BLoCs state with as little formatting as possible.
D. If you do have platform branching, it should be dependent on a single bool state emitted by a BLoC

State-Management using BLoCs

A state-management solution using BlOC is one of the most central decisions when building a Flutter application. An architectural pattern that functions as a state management solution. All business logic is extracted from the UI into BLoCs (Business Logic Components). The UI should only publish events to the BLoCs and display the UI based on the state of the BLoCs.

Cubit

The Cubit is a subset of BLoC Pattern It’s a class that stores an observable state, the observation is powered by Streams but in such a friendly way that it is not necessary to know reactive programming.

Designing Rules for the BLoCs

A. Input/outputs are simple sinks/streams only.
B. All dependencies must be injectable and platform agnostic.
C. No platform branching is allowed.
D. The actual implementation can be anything if rules 1-3 are followed.

The BLoC Package

The BLoC package enforces some additional architectural rules it limits the number of input and output streams of a given BLoC to one create small BLoCs with single responsibilities. Experience it is always better to favour small and concise BLoCs over large BLoCs that are responsible for multiple aspects of an application.

Bloc Consumer

Bloc Consumer exposes a builder and listener in order react to new states. Bloc Consumer is analogous to a nested Bloc Listener and Bloc Builder but reduces the amount of boilerplate needed.

Repository

A Repository to handles data operations. It knows where to get the data from and what API calls to make when data is updated. A Repository can utilize a single data source as well as it can be a mediator between different data sources, such as database, web services and caches.

Dependency Injection (DI)

DI provides two advantages, More reusable code and More testable code, as injected dependencies can easily be mocked. DI can be implemented using the "Repository Provider" class of the BLoC package. The Repository Provider functions very similarly to the BLoC Provider, in that it provides classes from its position in the widget tree to all its descendants

REST API

A REST API is an application programming interface that conforms to specific architectural constraints, like stateless communication and cacheable data. It is not a protocol or standard. While REST APIs can be accessed through a number of communication protocols, most commonly, they are called over HTTPS, so the guidelines below apply to REST API endpoints that will be called over the internet.

Data Layer/ Model Classes - The communication with external services. Each class in the data layer extends a version of the Service interface, These classes are used for modelling objects present in a specific data repository response.

Navigation and Routing- In this project we use **Generated Routes** Navigate to a new screen and back old screen.

Form Validation

The "Form BLoC" package was developed to realize form validation in this project. The aim of this package is to remove validation logic from the UI layer and instead realize it inside of BLoCs. The

package provides a set of Field BLoCs for common validation use cases such as email, non-empty fields, dates etc. but it also gives the option to create new Field BLoCs for other use cases.

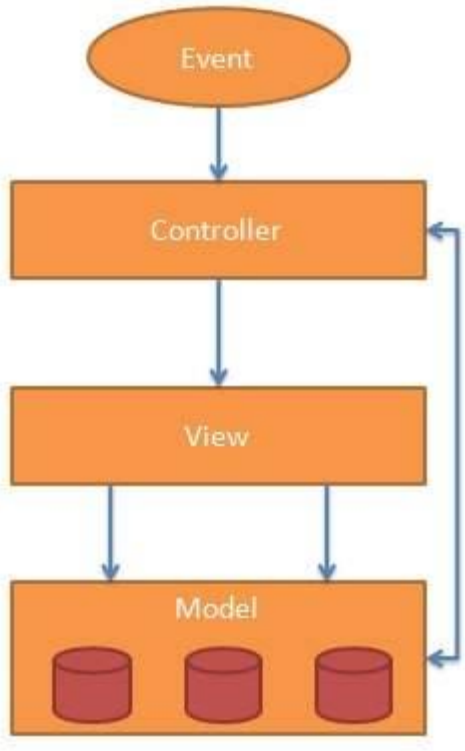
5. Web App Architecture & Design Best Practices

MVC Pattern

Model **V**iew **C**ontroller or MVC as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts −

- **Model** − It is the lowest level of the pattern responsible for maintaining data.
- **View** − It is responsible for displaying all or a portion of the data to the user.
- **Controller** − It is a software Code that controls the interactions between the Model and View.

MVC is popular because it isolates the application logic from the user interface layer and supports separation of concerns. The controller receives all requests for the application and then works with the model to prepare any data needed by the view. The view then uses the data prepared by the controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.



The Model

The model is responsible for managing application data. It responds to the request from view and to the instructions from controller to update itself.

The View

A presentation of data in a particular format, triggered by the controller's decision to present the data. They are script-based template systems such as JSP, ASP and very easy to integrate with AJAX technology.

The Controller

The controller responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of the data model.

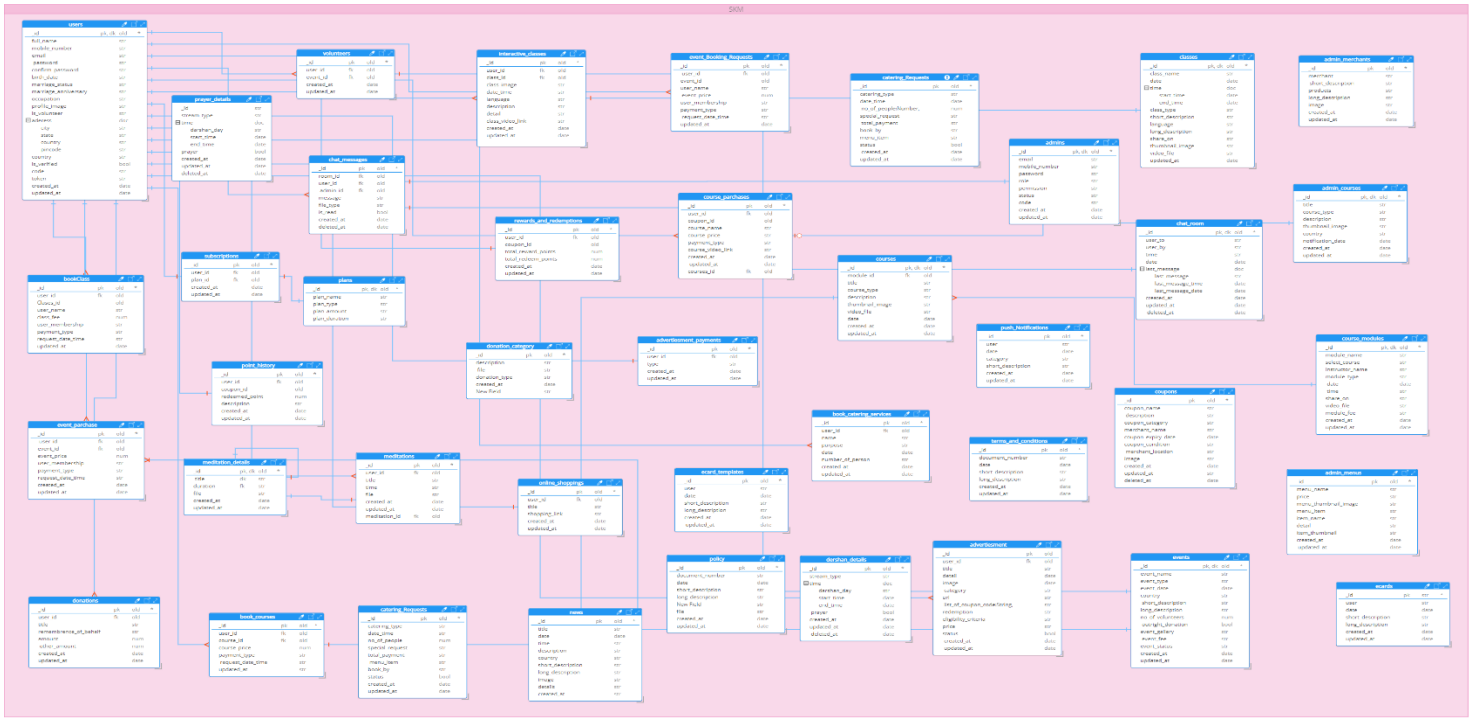
AngularJS is a MVC based framework. In the coming chapters, we will see how AngularJS uses MVC methodology.

6. Database Design Best Practices

Normalization: For storing different data into the different collections, we use normalization.
Denormalization: For storing different data into the same collection, for this we use denormalization.

General Rules for MongoDB Schema Design:

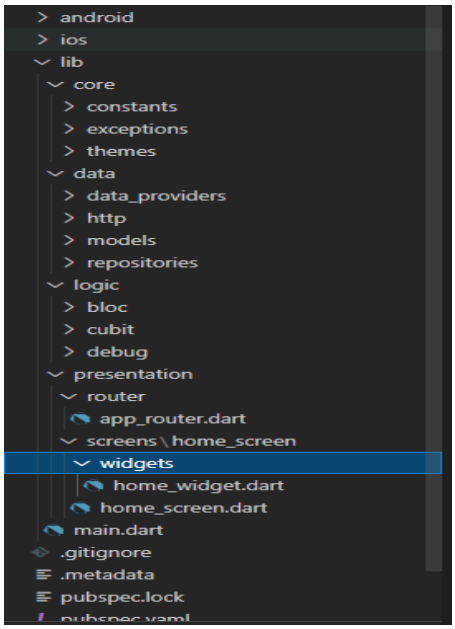
1. Favor embedding unless there is a compelling reason not to.
2. Needing to access an object on its own is a compelling reason not to embed it.
3. Avoid joins and lookups if possible, but don't be afraid if they can provide a better schema design.
4. Arrays should not grow without bound. If there are more than a couple of hundred documents on the many side, don't embed them if there are more than a few thousand documents on the many side, don't use an array of ObjectID references. High-cardinality arrays are a compelling reason not to embed.
5. As always, with MongoDB, how you model your data depends entirely on your particular application's data access patterns. You want to structure your data to match the ways that your application queries and updates it



Database schema: https://drive.google.com/file/d/1BzfY_rX5J_pEjlZq6O1Oj9e5gZYHBQ7O/view?usp=sharing

7. Filing Structure Best practices

The file structure of a project dictates how easily that project can be navigated and how easily different aspects of it can be found. With large codebases, the necessity of these attributes is amplified significantly.



8. Naming Standards - Best Practices


WeatherBloc
WeatherFetchSuccess
WeatherFetchInProgress
WeatherFetchFailure
THE STATES
"Bloc(or Cubit) subject + action + state of the action"
UserCubit
UserCreateSuccess
UserCreateInProgress
UserDeleteFailure
UserFetchSuccess
UI
bloc
states
events
THE EVENTS
"Bloc(or Cubit) Subject + action(event) in past tense"
UserBloc
UserFetched
UserDeleted
UserCreated
UserModified
THE FUNCTIONS
"Action on what the cubit is working on"
UserCubit
Fetch
Delete
Create
Modify