OMNI CHANNEL

SAADA Mobile Application
Architecture and Technical Specifications

Prepared by Development Team

Last Revised: 1.0.0, February 06, 2018

Table of Content

1. Introduction

1.1 Purpose of this Document

This document is prepared on the basis of standard guidelines offered by Google, IEEE.it is written for developers of mobile application as a guide to developing secure and standard apps. It may however also be interest to project managers of mobile application development projects.

1.2 Executive Summary

Omni Channel created SAADA application development guidelines – a set of specific capabilities, tools, resources, coding practices, and security validations that together enable mobile applications to be successfully planned, developed and delivered into production.

2. Application Architecture

The design of a project should be a concern from the beginning. One of the first things we should consider is the architecture that we plan to adopt as it will define how different elements of our applications relate to one another.it will also establish some ground rules to guide us during development. It is recommended to use MVP architecture, which allows us to perform unit testing smoothly.

2.1 MVP (Model View Presenter) - Android

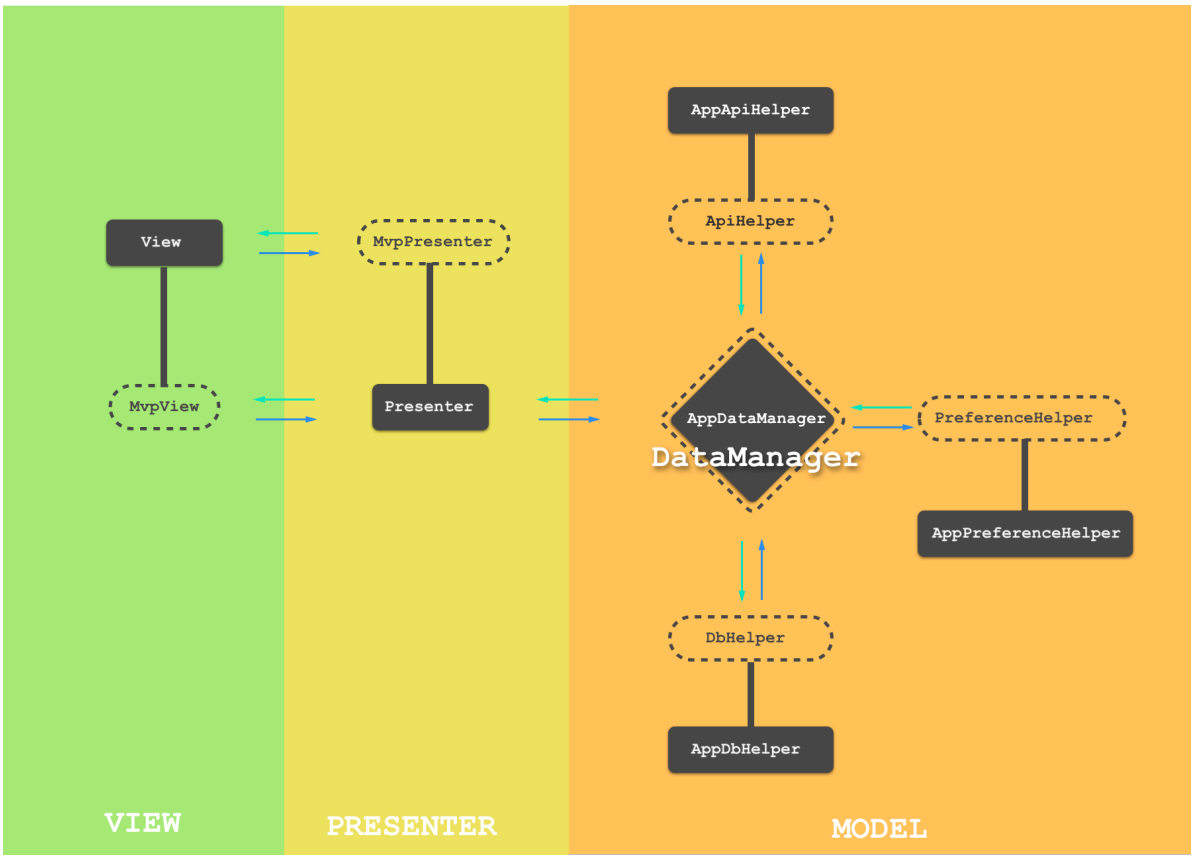
MVP divides the application into three basic components:

- **Model:** It is responsible for handling the data part of the application.
- **View:** It is responsible for laying out the views with specific data on the screen.
- **Presenter:** It is a bridge that connects a Model and a View. It also acts as an instructor to the View.

MVP lays few ground rules for the above-mentioned components, as listed below:

- A View's sole responsibility is to draw UI as instructed by the Presenter. It is a dumb part of the application.
- View delegates all the user interactions to its Presenter.

- The View never communicates with Model directly.
- The Presenter is responsible for delegating View's requirements to Model and instructing View with actions for specific events.
- Model is responsible for fetching the data from server, database and file system.



2.2 MVVM (Model View ViewModel) (iOS)

- **View:** It informs the ViewModel about the user's actions
- **ViewModel:** It exposes streams of data relevant to the View
- **Model:** Abstracts the data source. The ViewModel works with the DataModel to get and save the data.

Remarks: *Use **MVP** in situations where binding via a datacontext is not possible. Use **MVVM** in situations where binding via a datacontext is possible.*

2.3 Technical Specification : (Android)

- **Editor** - Android Studio(Android)

- **Min SDK version** - **TBD (Recommendation - 19)**
- **Device Support** - **TBD (Samsung, HTC, LG, Nexus)**
- **Orientation support** - Portrait (Video playing screen will support the landscape mode also)
- **Programming Language** - Java, support Kotlin.
- **Reactive programing** - RxJava
- **UI/UX Design** - Android Material Design (to be discussed)
- **Localization** - **(English & Arabic)**
- **Networking** - Retrofit 2, OKHttp
- **Database** - **N/A**
- **Image loader** - Glide or Picasso

2.4 Technical Specification : (iOS)

- **Editor** - Xcode
- **Min OS version** - **iOS 10**
- **Device Support** - **iPhone 5s and above**
- **Orientation support** - Portrait (Video playing screen will support the landscape mode also)
- **Programming Language** - Swift.
- **Reactive programing** - N/A
- **UI/UX Design** - Storyboards
- **Localization** - **(English & Arabic)**
- **Networking** - Alamofire
- **Database** - **N/A**
- **Image loader** - Alamofire, SDWebimage

3. Coding Guidelines:

3.1 Java Code Style Rules

Android development has to follow the java coding conventions but should support kotlin.

- Don’t Ignore Exception
- Don’t Catch Generic Exception
- Don't Use Finalizers
- Fully Qualify Imports
- Java Library Rules to be followed
- Java Style Rules to be followed
- Define Fields in Standard Places
- Order Import Statements
- Follow Field Naming Conventions
- Use Standard Brace Style
- Limit Line Length
- Use Standard Java Annotations
- Log Sparingly
- Unit Test cases has to be created
- Use common naming conventions for icon assets.

3.2 Generic Policies

- Wildcard imports, static or otherwise, are not used.

- When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no other code in between (not even private members).

- Braces are used with if, else, for, do and while statements, even when the body is empty or contains only a single statement.

- One variable per declaration- Every variable declaration (field or local) declares only one variable: declarations such as int a, b; are not used.

- Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, com.example.deepspace,
not com.example.deepSpace or com.example.deep_space

- Method names are written in lowerCamelCase.

- Method names are typically verbs or verb phrases. For example, sendMessage or stop.

- Constant names use CONSTANT_CASE: all uppercase letters, with each word separated from the next by a single underscore.

- Non-constant field names (static or otherwise) are written in lowerCamelCase.

Local variable names & method parameter names are written in lowerCamelCase.

- A method is marked with the @Override annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method

- Never catch Generic exception. Catch specific exceptions like NPE and not Exception
- Don't use static methods to be called from adapter, Instead use Interfaces and use them from Adapters

- For utility classes we should follow correct design pattern. For e.g For Alert Utility we can use builder pattern.

- Create own custom utility classes to achieve common tasks.

- Their should be no duplicacy in the code. Use BaseActivity or Base classes for repetive tasks.

- Use XMLs for creating custom layouts. Do not create views in code. Just inflate them from the xmls.

-
- There should be separate Animation Utility classes.

- For custom views there should be separate .java file for it.

- There should be no hardcoded strings in the code. Use strings.xml file for localization and UI messages.

- Use LogUtility class. So that logs can be switched off from one boolean flag.

3.3 Version control

- Git to be used for the version controlling. JIRA to be interconnected with the git to manage(Management Tool).

- Every commit should have a appropriate description about the changes/fixes that is done along with JIRA ticket link.
- Code commit should be done between (6 pm and 7 pm Dubai Time)
- We should have access to the master repository.
- Every committed code should be executable.

3.4 Modular approach

Configuration File:

- It will contain values that need to be configured for the application. The file format will be JSON.e.g. it will contain keys required for application like **BaseURL** and any other configurable keys.
- It will be downloaded each time on the launch of the application. After this file will get loaded in the app the further app flow will be configured.

Note: configuration.json file will be different for **UAT** and **Production** environments. This file will be maintained from development team.

4. Provisioning/Distribution

4.1 UAT

4.1.1 Unit Testing

Your local unit test class should be written as a JUnit 4 test class. Mockito to be used to mock the methods.

Test coverage reports are an important tool to measure how much our tests actually exercise our code. Although not guaranteed a bug-free software, to have a high percentage of coverage can avoid a lot of headaches in the project.

To generate the coverage report in Android, we use **Jacoco** (**Ja**va **Co**de **Co**verage), one of the most used tools in Java for this purpose.

4.1.2 Manual Testing

Manual testing has to be performed using Android debug tools. All the flows has to be tested and well documented

4.2 Production Level

Release note has to be created with all the abbreviations and usage.