TrainingPeaks

Coding Test Architecture Document

George Nikolai Kotula

# Introduction

This document outlines the general architecture of the submitted TrainingPeaks coding test application.

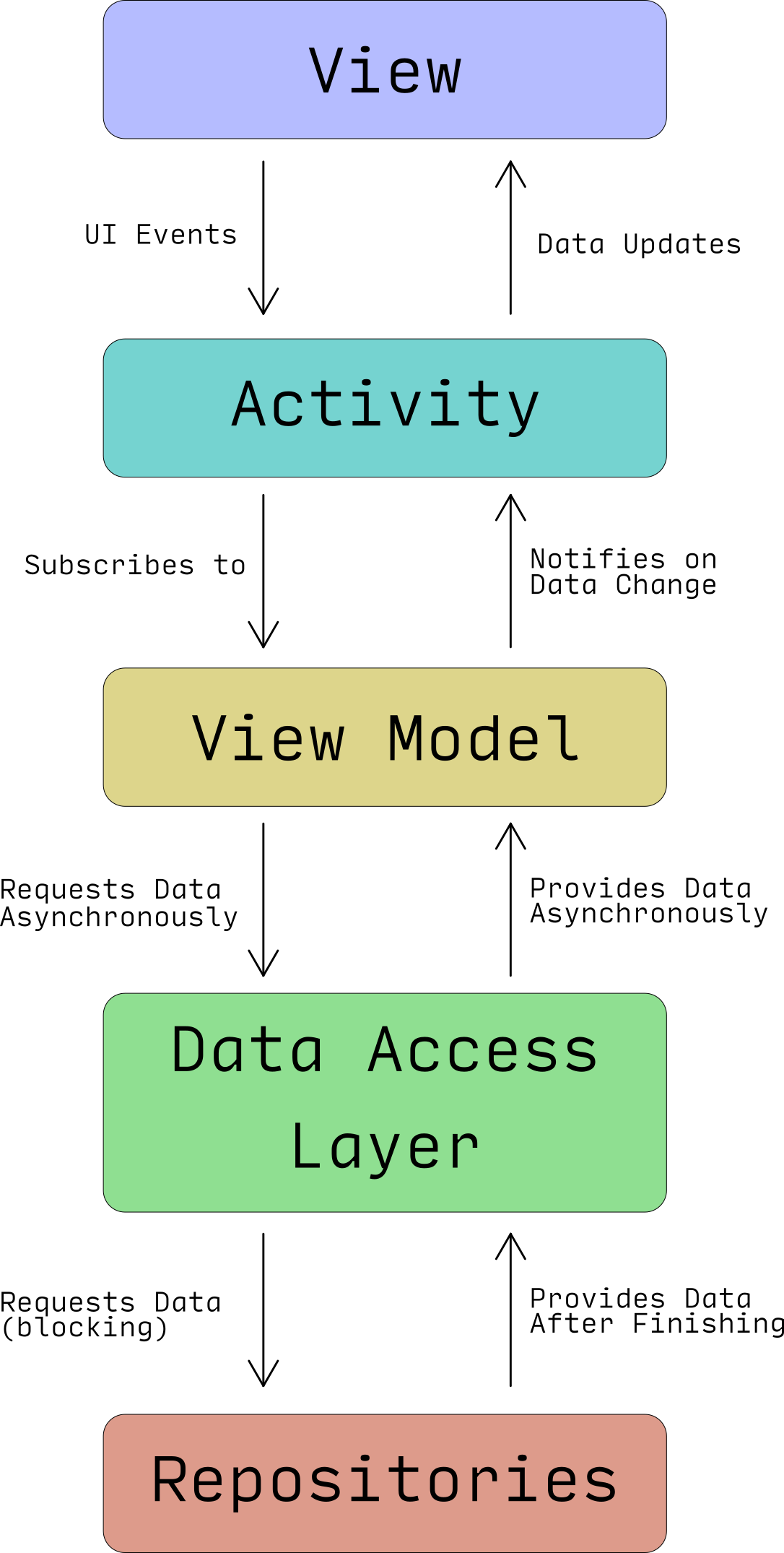
# Overview

The app is designed in a sort of “augmented MVC” pattern. “Augmented” means that the Model has several layers within it to facilitate better encapsulation. The Activities/Fragments act as the traditional Controllers, and the Views are also very traditional according to the MVC pattern.

The Model is broken up into the *View Models*, the *Data Access Layer*, and *Repositories*.

View Models are the special Android implementation which persist data between app state changes such as rotating the device. The Data Access Layer contains the Services responsible for coordinating asynchronous calls to the Repositories. The Repositories are the sources of data, whether that is persistent data in the form of SQLite, or an HTTP client that fetches data.

To this end, the high-level view of the architecture looks like this (see next page):



Note: The “Activity” node here includes Fragments as well.

# Specific Components

This section will detail the specific components and their responsibilities. All the views are associated with their corresponding Activities and are nothing other than Layout XMLs, so they will not be covered.

## Activities

### WorkoutSubmission

The [WorkoutSubmission](#_WorkoutSubmission) Activity is the main Activity the user lands on when starting the app. It is very simple in its execution, only handling a single button press to launch the [WorkoutExplorer](#_WorkoutExplorer) Activity.

### WorkoutExplorer

The [WorkoutExplorer](#_WorkoutExplorer) Activity shows the Peak Heart Rates and Peak Speeds tabs. As such, it implements a ViewPager where each tab has its own Fragment dedicated to showing the corresponding data.

## Fragments

Note that the \*RecyclerViewAdapter components are not covered because they simply provide the implementation necessary to present the data to the RecyclerView.

### PeakFragment

This Fragment is the abstract base class for the [PeakHeartRateFragment](#_PeakHeartRateFragment) and the [PeakSpeedsFragment](#_PeakSpeedFragment). Its purpose is mainly to consolidate as much shared code between the two Fragments as possible. The key part of this component is the implementation of the LifecycleOwner interface so that child objects (like the View Models) can do the appropriate cleanup to avoid leaking Activity/Fragment references.

### PeakHeartRateFragment

As its name suggests, this Fragment is responsible for displaying Peak Heart Rate data. It uses a RecyclerView to display the data. It is also responsible for binding to and kicking off the associated View Model for the Peak Heart Rates, [HeartRateModel](#_HeartRateModel).

### PeakSpeedFragment

As its name suggests, this Fragment is responsible for displaying Peak Speed data. It uses a RecyclerView to display the data. It is also responsible for binding to and kicking off the associated View Model for the Peak Speeds, [SpeedModel](#_SpeedModel).

### ErrorDialog

Again, this is pretty self-explanatory. Its purpose is to provide a dialog to inform the user of errors that occur within the app.

## View Models

This is under the package “Models.”

### PeakModel

This is the abstract base class for the other View Models. Like with the [PeakFragment](#_PeakFragment), its purpose is mainly to consolidate shared code between the View Models. One special thing about this class is its implementation of the LifecycleObserver interface. This interface reacts to changes in the parent Fragment, and it is important that all references to anything governed by the Android component lifecycle are updated in response to the various lifecycle events.

### HeartRateModel

Concrete implementation of the [PeakModel](#_PeakModel), this spins up IntentService instances to fetch the data it needs. It also acts as a filter for the returning data to remove duplicate entries and otherwise sanitize the data before handing it off to the Activities/Fragments.

### SpeedModel

Concrete implementation of the [PeakModel](#_PeakModel), this spins up IntentService instances to fetch the data it needs. It also acts as a filter for the returning data to remove duplicate entries and otherwise sanitize the data before handing it off to the Activities/Fragments.

## Data Access Layer

If I was going to modify anything about this code, it would be focused on this layer. Right now, there is no persistence of data fetched from the Repos, and what’s worse, each different type of data is its own HTTP request. That means that each Fragment generates its own HTTP request and we wind up fetching the data from the REST endpoint twice. This is just awful, and it is a result of neglect on my part in exchange for smoothing out the user experience.

The reason I chose UX over this core component is the following. I put myself in the shoes of someone getting a product done for a deadline (to show the full development process and the effort put in on all fronts). I observed my own behavior, and I saw that after glancing over the first tab’s data, the second tab’s data was always loaded when I swiped to it. In other words, the user never even notices this slowdown because they are looking at the first data to show up while the other data loads. With this in mind, I made the choice to put my energy elsewhere to make the result look and feel really slick.

This is under the package “Services” for no other reason than I named it “Services” at the beginning and never got around to changing it.

NOTE: Everything from the DAL should only depend on [IWorkoutRepo](#_IWorkoutRepo). No concrete implementations of that interface should ever be used by the DAL.

### WorkoutService

Currently, this is the only component in the DAL, and it does all the coordinating of fetching data from the Repos. It is an IntentService which uses a ResultReceiver to pass the data asynchronously up to the View Models. Since each call to get the data starts its own job scheduled by the Android OS, the implementation is very straightforward when handling requests. It simply queues them up by calling startService() from static methods.

### WorkoutResultReceiver

Acts as the glue between the [WorkoutService](#_WorkoutService) and the View Models that are waiting for the data.

## Repositories

Note this is under the package “Repos.”

### IWorkoutRepo

This is the interface that allows different data sources to be used with Dependency Injection. The Data Access Layer should *never* depend on concrete implementations of this interface. Leave that up to the Dependency Injection.

### HttpWorkoutRESTClient

A data source that fetches Workout data from a REST endpoint. It parses the JSON response and provides the parsed POJOs to the DAL.

## POJOs

Plain Old Java Objects are the method of encapsulating the Workout data retrieved from the data sources. They are pretty self-explanatory insofar as they are immutable data-holding objects with methods for Serialization/Deserialization, comparison, equality, etc.

These are under the package “Data.”