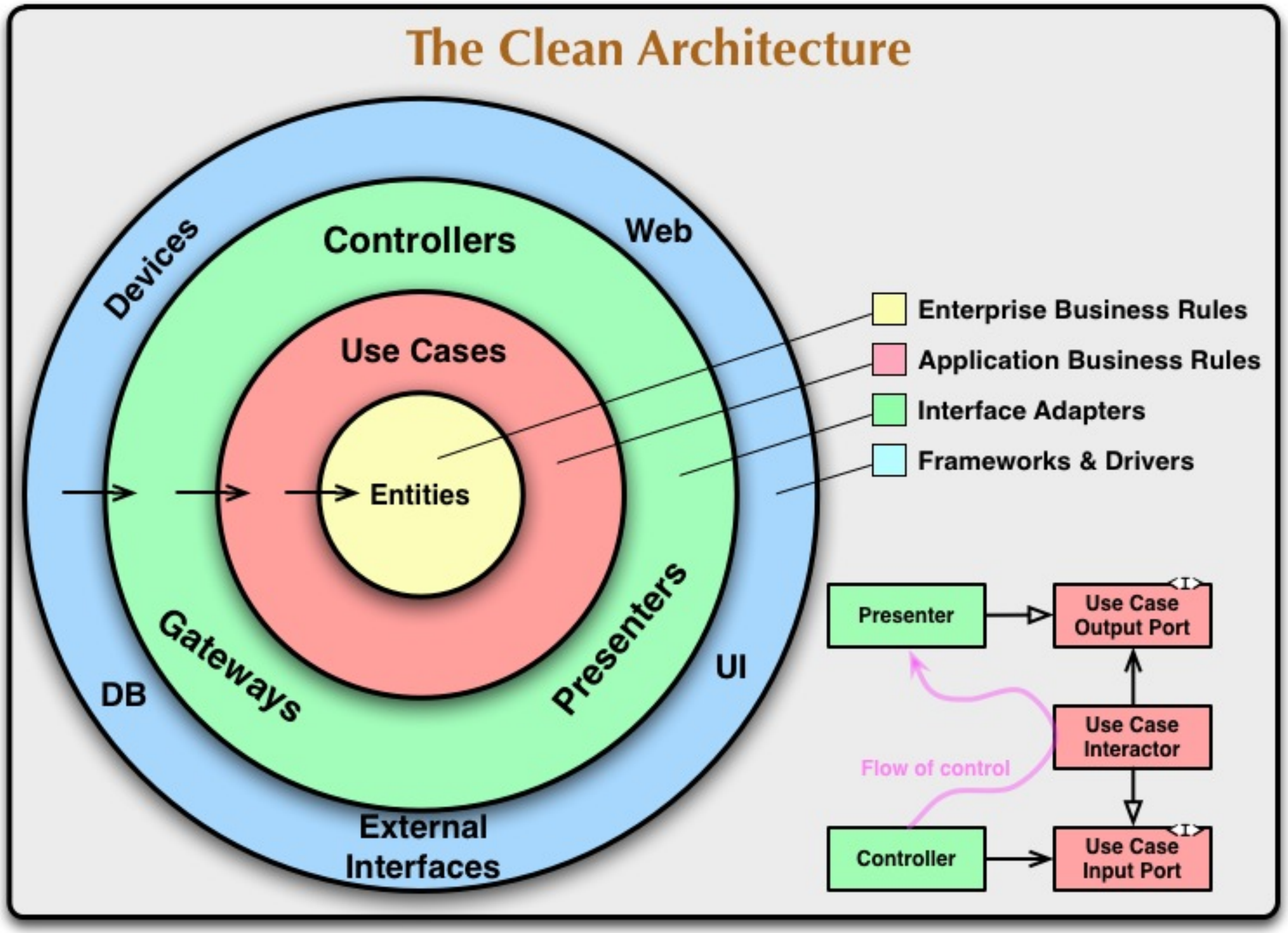
Android Architecture: A RxJava, Retrofit, and VIPER Medley

Andrew Brazina
Charter Communications
Andrew.Brazina@charter.com

Motivations

- We needed a robust service infrastructure
- Single Responsibility Principle
  - Separate data concerns from the UI of our application
  - Reduce dependency
- Testability
  - Single Purpose
  - Simple model objects
- No Event Bus (Universal messaging system)
  - Requires enforced discipline, issues with scaling, difficult to debug
- Adaptable
  - Constantly changing requirements and new features

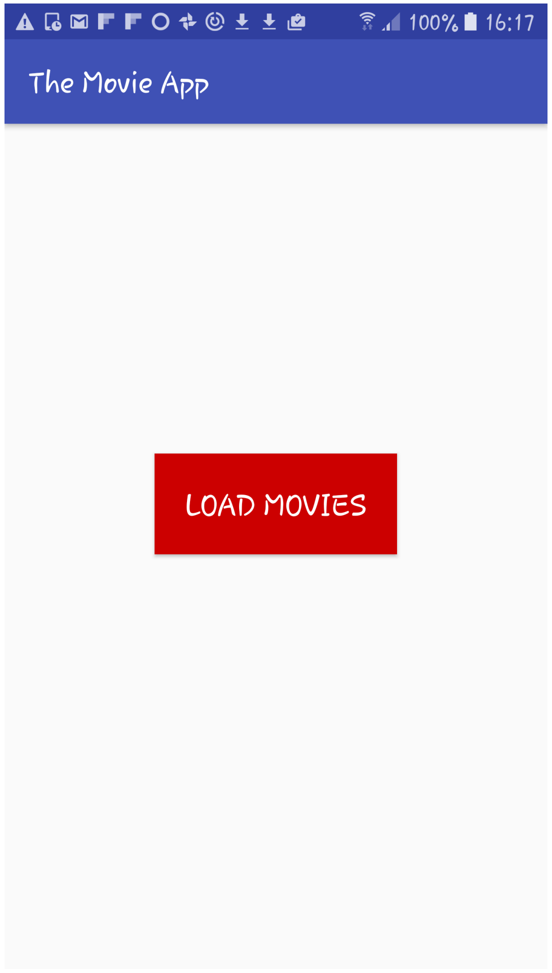
Clean Architecture

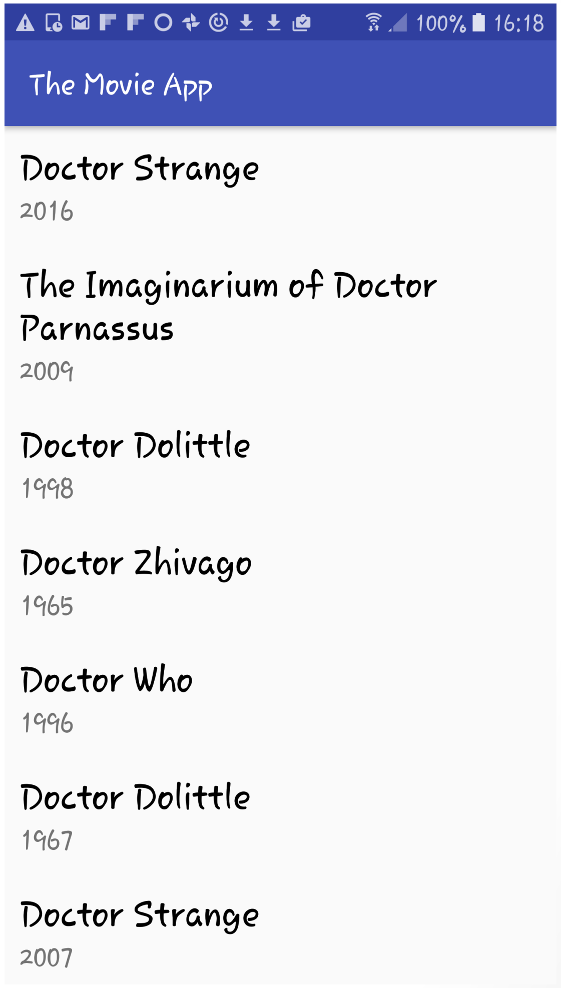


To achieve our results our team built a robust, sustainable, testable and adaptable Android application infrastructure

- Technologies
  - Retrofit
  - GSON
  - RxJava
- Design
  - Observer Pattern
  - Factory Pattern
  - VIPER

Sample Application




100% 16:18

The Movie App

Doctor Strange
2016

The Imaginarium of Doctor Parnassus
2009

Doctor Dolittle
1998

Doctor Zhivago
1965

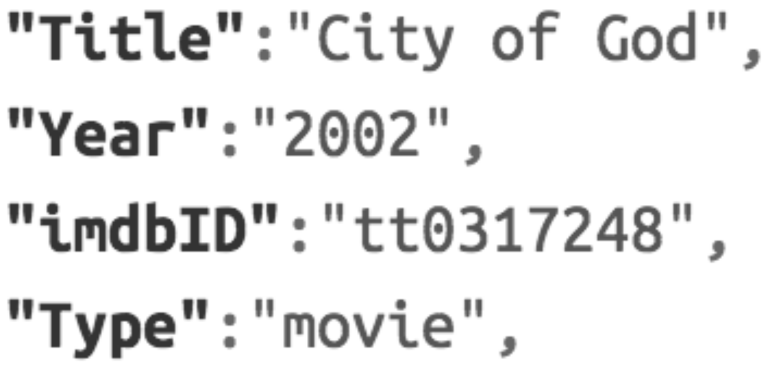
Doctor Who
1996

Doctor Dolittle
1967

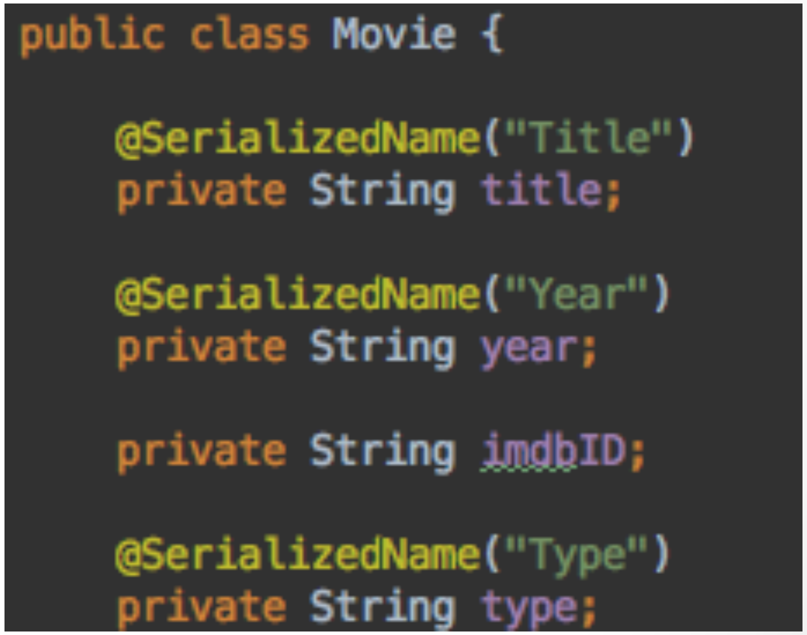
Doctor Strange
2007

GSON

- Ridiculously simple JSON parsing



```
"Title":"City of God",
"Year":"2002",
"imdbID":"tt0317248",
"Type":"movie",
```



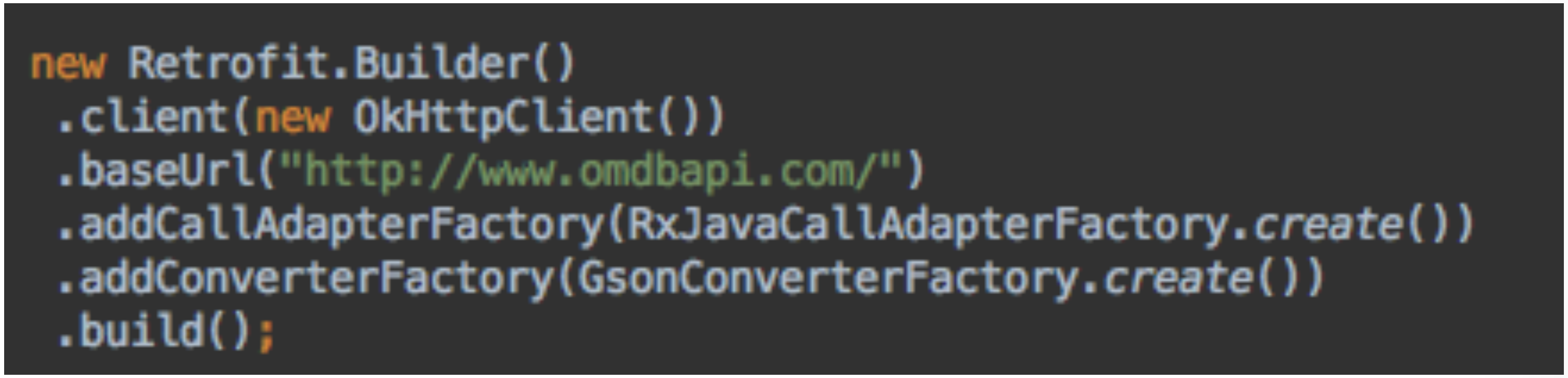
```java
public class Movie {

    @SerializedName("Title")
    private String title;

    @SerializedName("Year")
    private String year;

    private String imdbID;

    @SerializedName("Type")
    private String type;
```

Service Layer

- Retrofit
  - Very popular and well supported
  - “Turns your HTTP API into a Java Interface”
  - Simple and easy to use
  - Fast and efficient
  - Lightweight and customizable

Retrofit

- Define Service Builder



```
new Retrofit.Builder()
 .client(new OkHttpClient())
 .baseUrl("http://www.omdbapi.com/")
 .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
 .addConverterFactory(GsonConverterFactory.create())
 .build();
```

Retrofit – Services as Interfaces

- Dynamic URL

```
@GET
Observable<List<FavoriteChannel>> getFavoritesList(@Url String url);
```

- Path with global base URL

```
@GET("users/favorites")
Observable<List<FavoriteChannel>> getFavoritesList();
```

- Dynamic path

```
@GET("users/{userId}/favorites/")
Observable<List<FavoriteChannel>>
                    getFavoritesList(@Path("userId") String userId);
```

Retrofit – Services as Interfaces

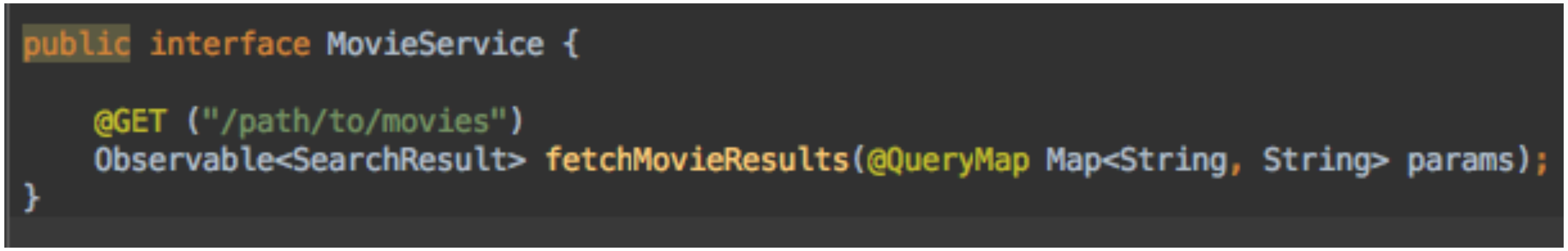
- Query parameters

```
@GET("users/favorites/")
Observable<List<FavoriteChannel>>
                                   getFavoritesList(@QueryMap Map<String, String> params);
```

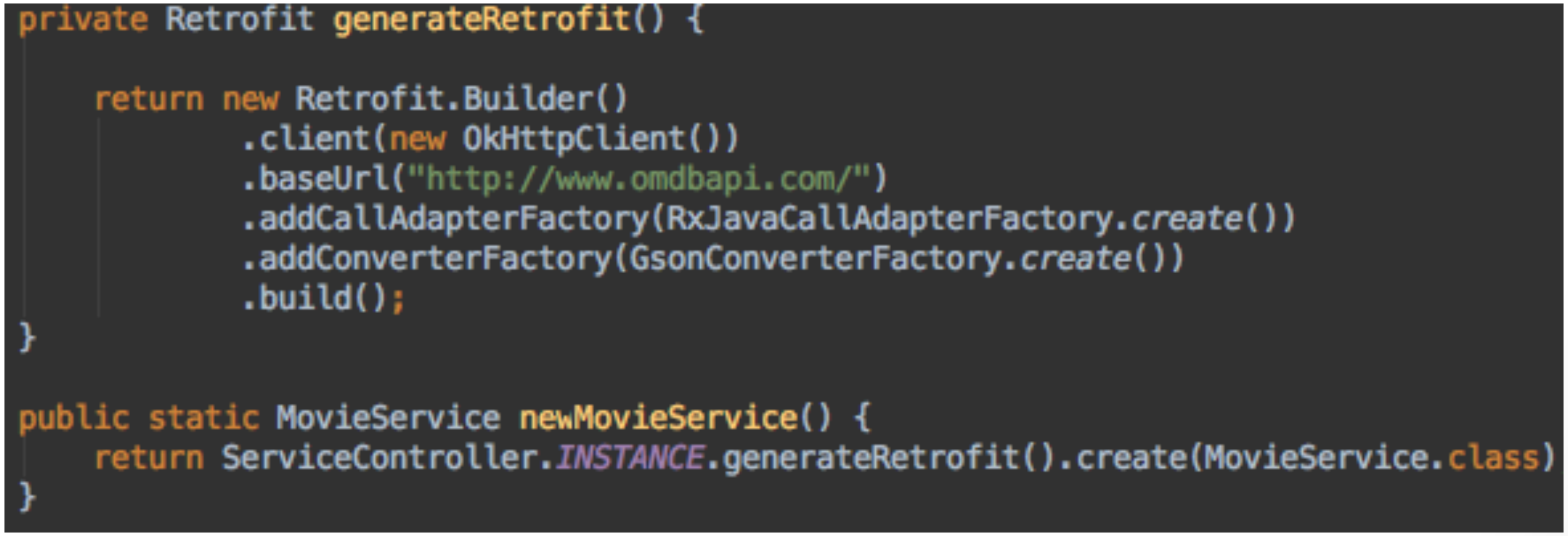
- Posting with body

```
@POST("users/favorites")
Observable<Void> addFavorites(@Body FavoriteChannel favoriteChannel);
```

Retrofit – Coding Example



```java
public interface MovieService {

    @GET ("/path/to/movies")
    Observable<SearchResult> fetchMovieResults(@QueryMap Map<String, String> params);
}
```



```java
private Retrofit generateRetrofit() {

    return new Retrofit.Builder()
            .client(new OkHttpClient())
            .baseUrl("http://www.omdbapi.com/")
            .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
            .addConverterFactory(GsonConverterFactory.create())
            .build();
}

public static MovieService newMovieService() {
    return ServiceController.INSTANCE.generateRetrofit().create(MovieService.class)
}
```

RxJava

- "The Observer pattern done right. ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming" – reactivex.io
- RxJava is a port of the Reactive Extensions (Rx) to java
- In reactive programming the consumer reacts to data as it comes in. (Asynchronously)
- Reactive programming allows propagation of events and changes to registered observers

RxJava

Components

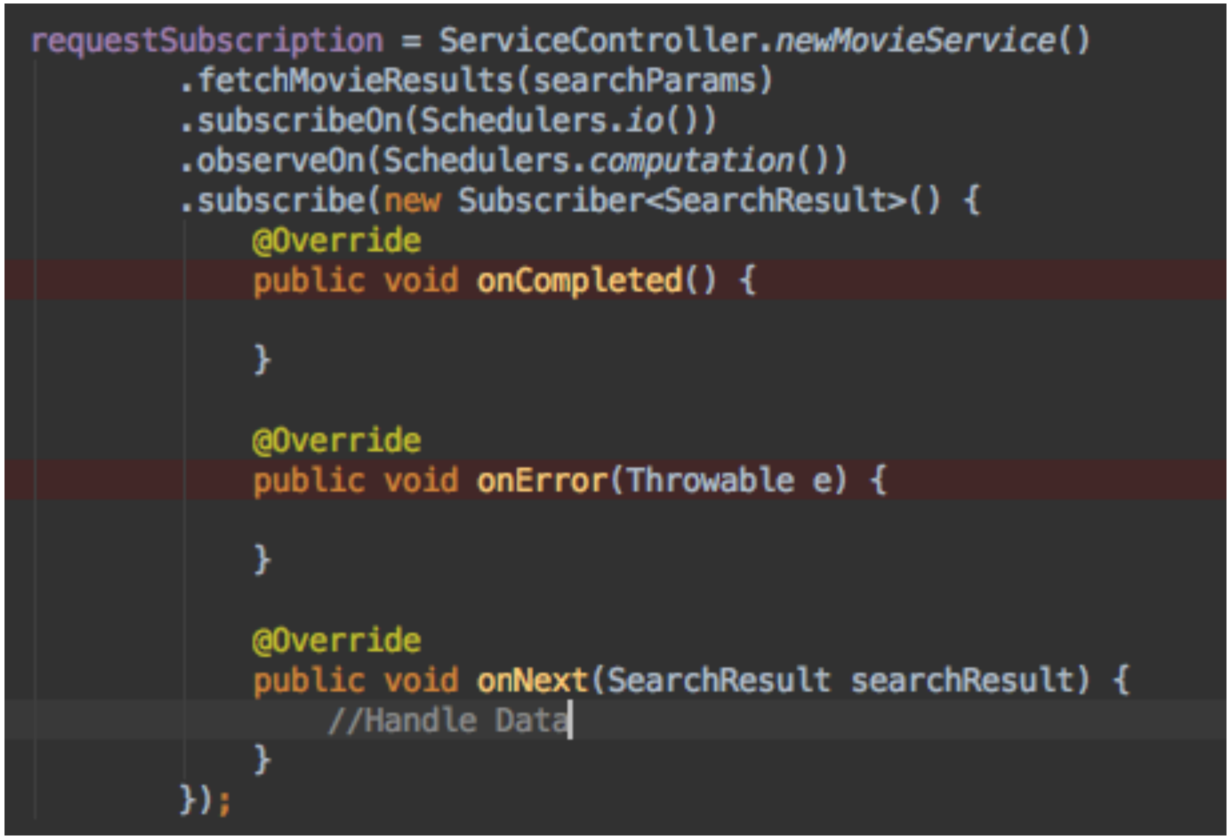
- Observable<T>
  - Emits items or sends notifications

→

- Subscriber
  - Receives events from an observable

RxJava – Coding Example

Integrates with Retrofit to provide thread management



```java
requestSubscription = ServiceController.newMovieService()
        .fetchMovieResults(searchParams)
        .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.computation())
        .subscribe(new Subscriber<SearchResult>() {
            @Override
            public void onCompleted() {

            }

            @Override
            public void onError(Throwable e) {

            }

            @Override
            public void onNext(SearchResult searchResult) {
                //Handle Data
            }
        });
```

Design

- Separate UI from the business logic
  - Created separate module to contain all services and business logic
  - Pure Java library
  - Use RxJava and the Observer pattern to message the UI

- Provide highly structured architecture to guide feature development

VIPER

- View – displays what it is told by the presenter
- Interactor – contains the business logic
- Presenter – contains view logic for preparing content for display
- Entity – contains basic model objects
- Routing – contains navigation logic for describing which screens to show

VIPER

- Great since it adheres to the Single Responsibility Principle
  - Each section is responsible for a clearly defined role
  - Each content area of the app has it’s own VIPER module

- Provides a clearly defined path for feature development

VIPER

- Challenges when applying to Android
  - Activities and Fragments are not instantiated they are started by the framework

  - We had multiple existing infrastructures to integrate
    - Making updating existing UI classes difficult

  - Decided to focus on the business layer of the application and update the UI where possible

VIPER -Modified

$$\begin{tikzcd}
\text{View} \arrow[r] & \text{Activity/Fragments} \\
\text{Interactor} \arrow[r] & \text{Controllers} \\
\text{Presenter} \arrow[ru] & \\
\text{Entity} \arrow[r] & \text{Presentation Data} \\
\text{Routing} \arrow[r] & \text{Flow Controller}
\end{tikzcd}$$

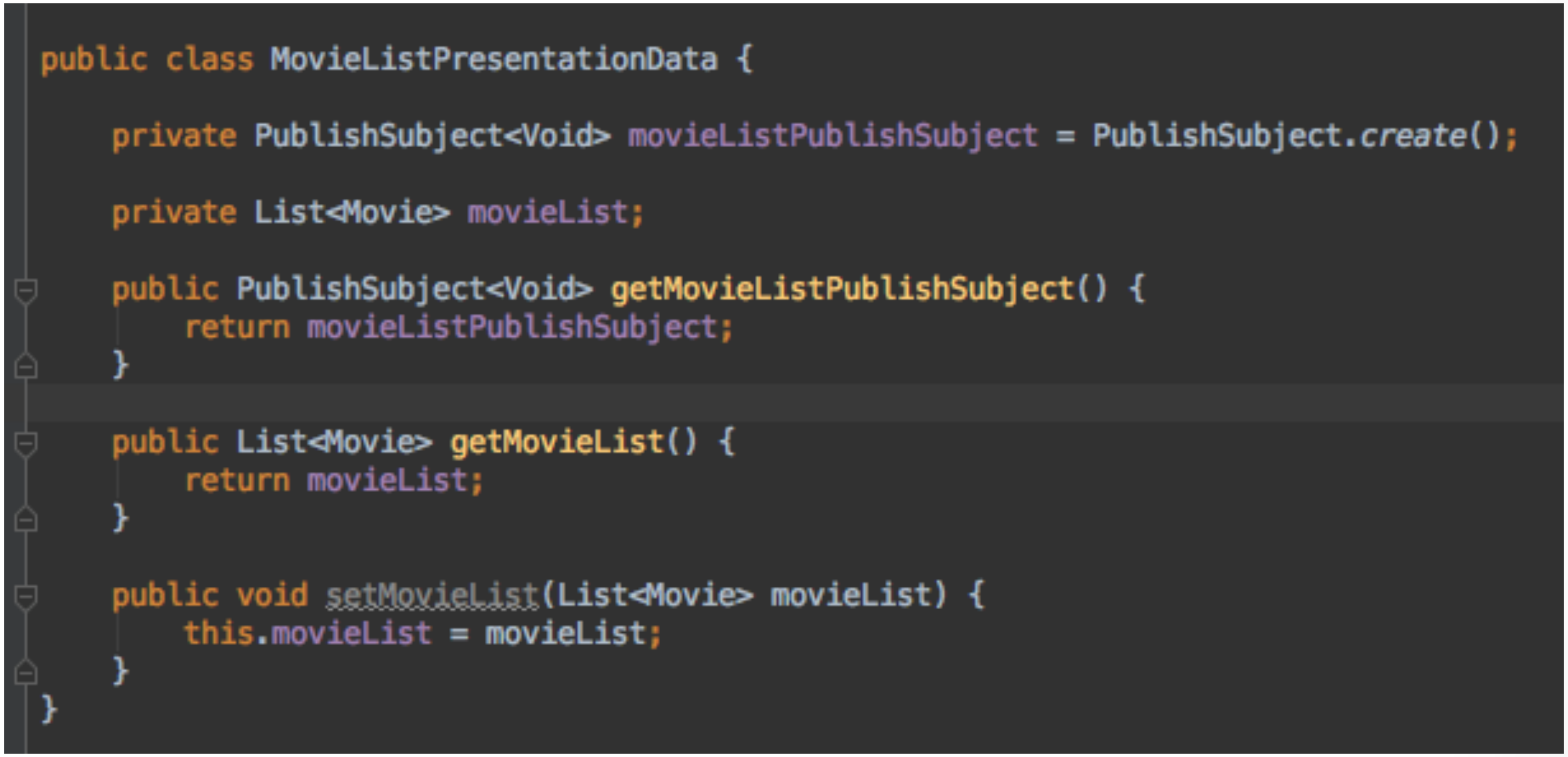
VIPER(Modified)

- Benefits to modified VIPER
  - Allows us to take advantage of existing Activity functionality by using Activities and Fragments
  - Minimizes rework to existing application flow
  - By combining the Interactor and Presenter into the Controller, allows one class to handle all data manipulation
    - By keeping Controllers single responsibility, monolith classes are avoided

Presentation Data

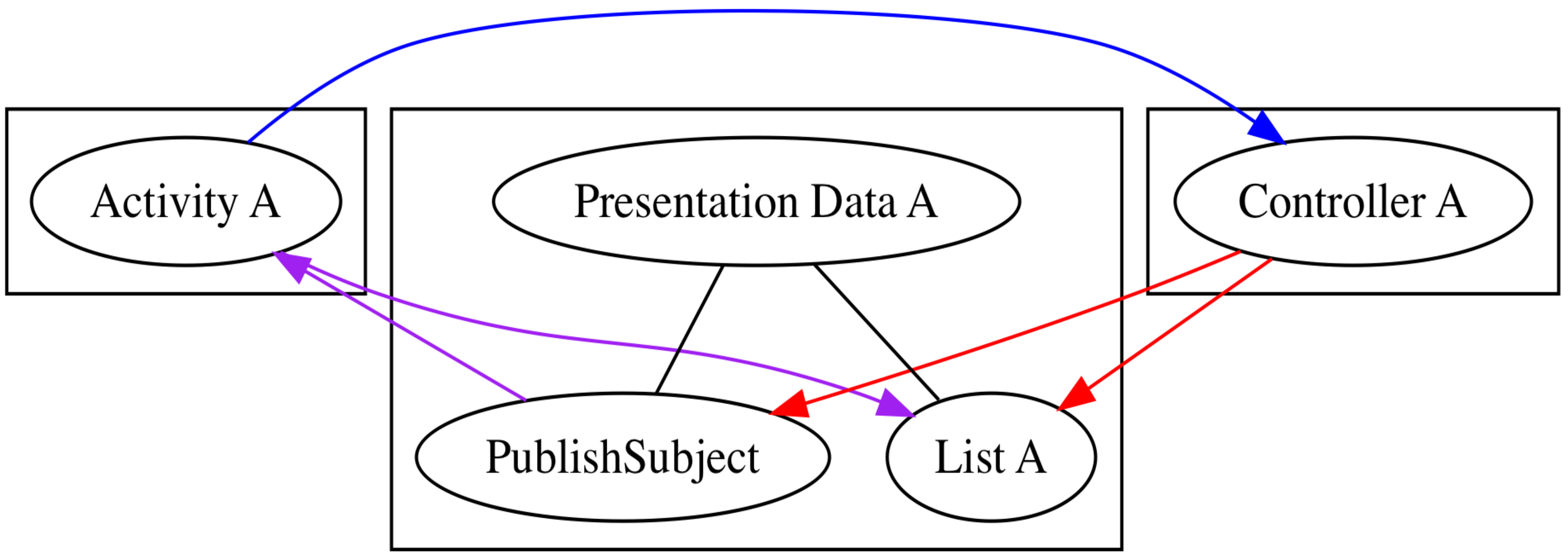
- Provides data to the UI and how it should be presented to the user
- Notifies the UI when data is ready to be displayed or has been updated using the observable pattern

Presentation Data - Example



```
public class MovieListPresentationData {

    private PublishSubject<Void> movieListPublishSubject = PublishSubject.create();

    private List<Movie> movieList;

    public PublishSubject<Void> getMovieListPublishSubject() {
        return movieListPublishSubject;
    }

    public List<Movie> getMovieList() {
        return movieList;
    }

    public void setMovieList(List<Movie> movieList) {
        this.movieList = movieList;
    }
}
```

PublishSubject



Presentation Factory

- The source of all presentation data for the application
- Handles the instantiation of presentation data objects

Controllers

- Each feature of the app has one controller to process data and prepare display data
- Each controller is defined by an interface
  - Makes code contract driven
  - Makes dependency injection simpler
  - Clearly defines what needs to be tested
- Data from controllers is exposed to application through presentation data

Controller Factory

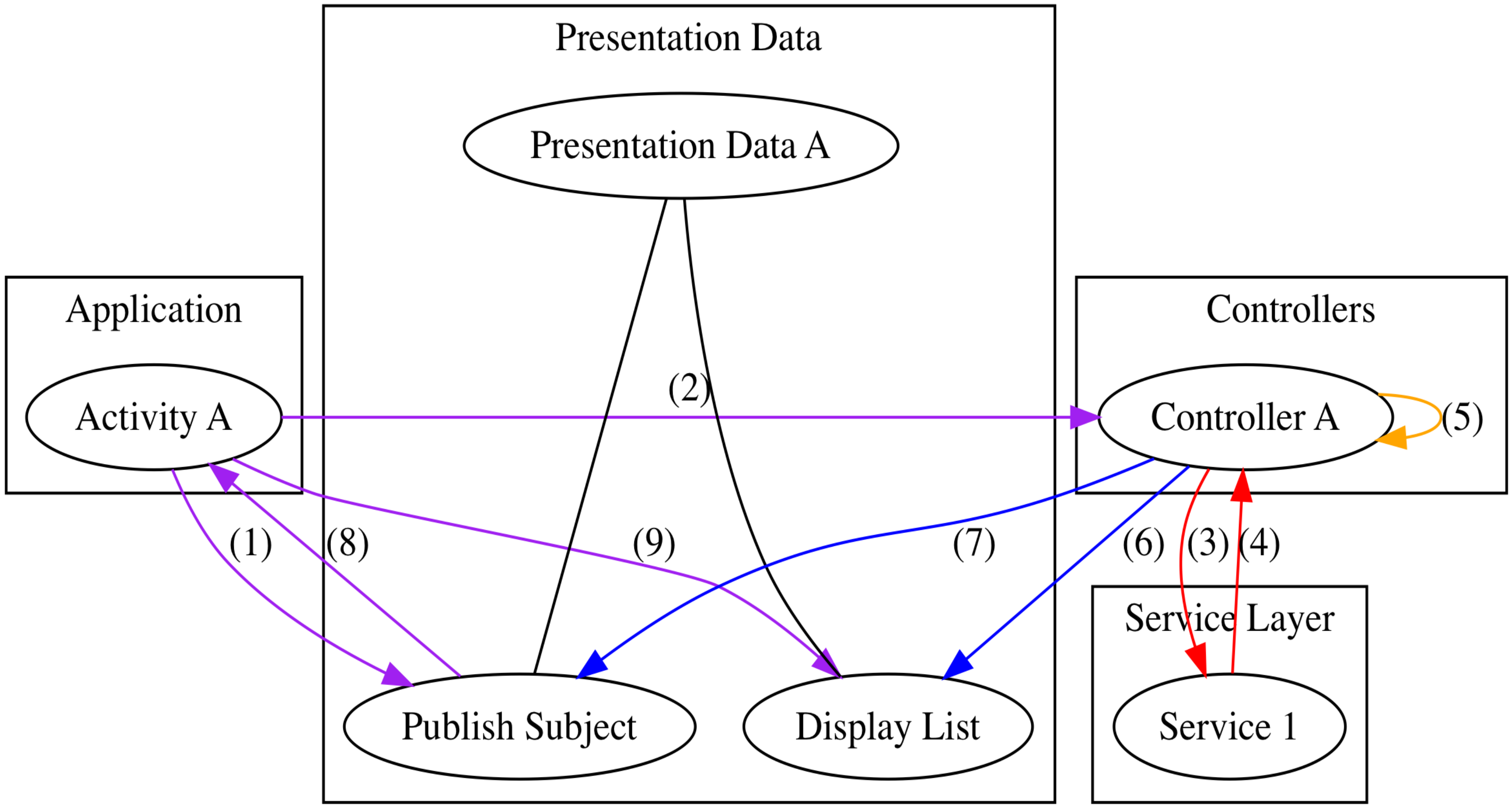
- Handles instantiation of controllers for the application
- Determines specific implementation of controller to instantiate
  - At build time based on build parameters
  - At runtime based on services

Service Controller

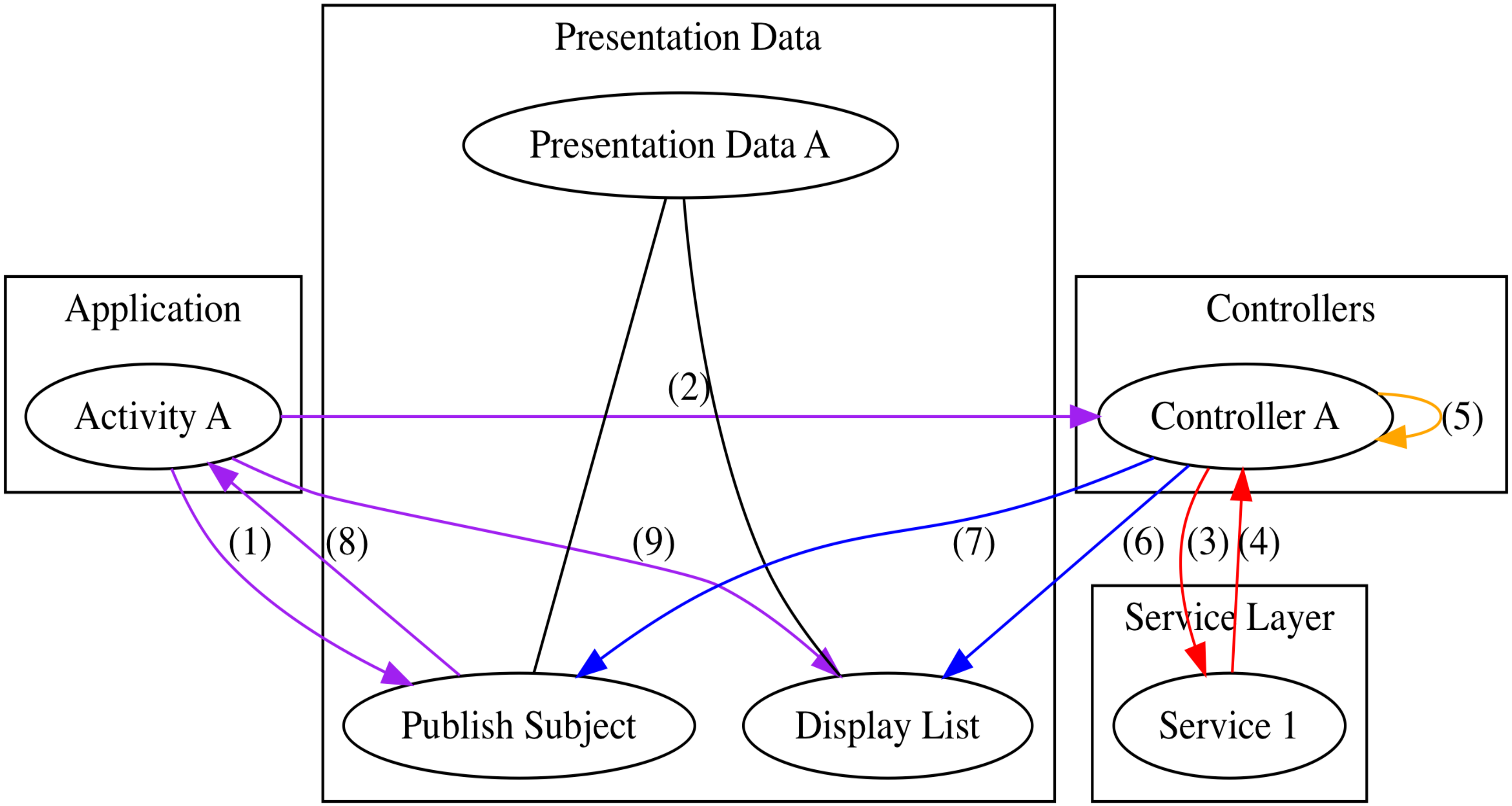
- Configures retrofit with application specific requirements
  - Configure HttpClient with interceptors
    - Adding authentication, universal headers, caching
  - Specify response handling
  - Specify CallAdapter

- Handles returning all services generated by retrofit

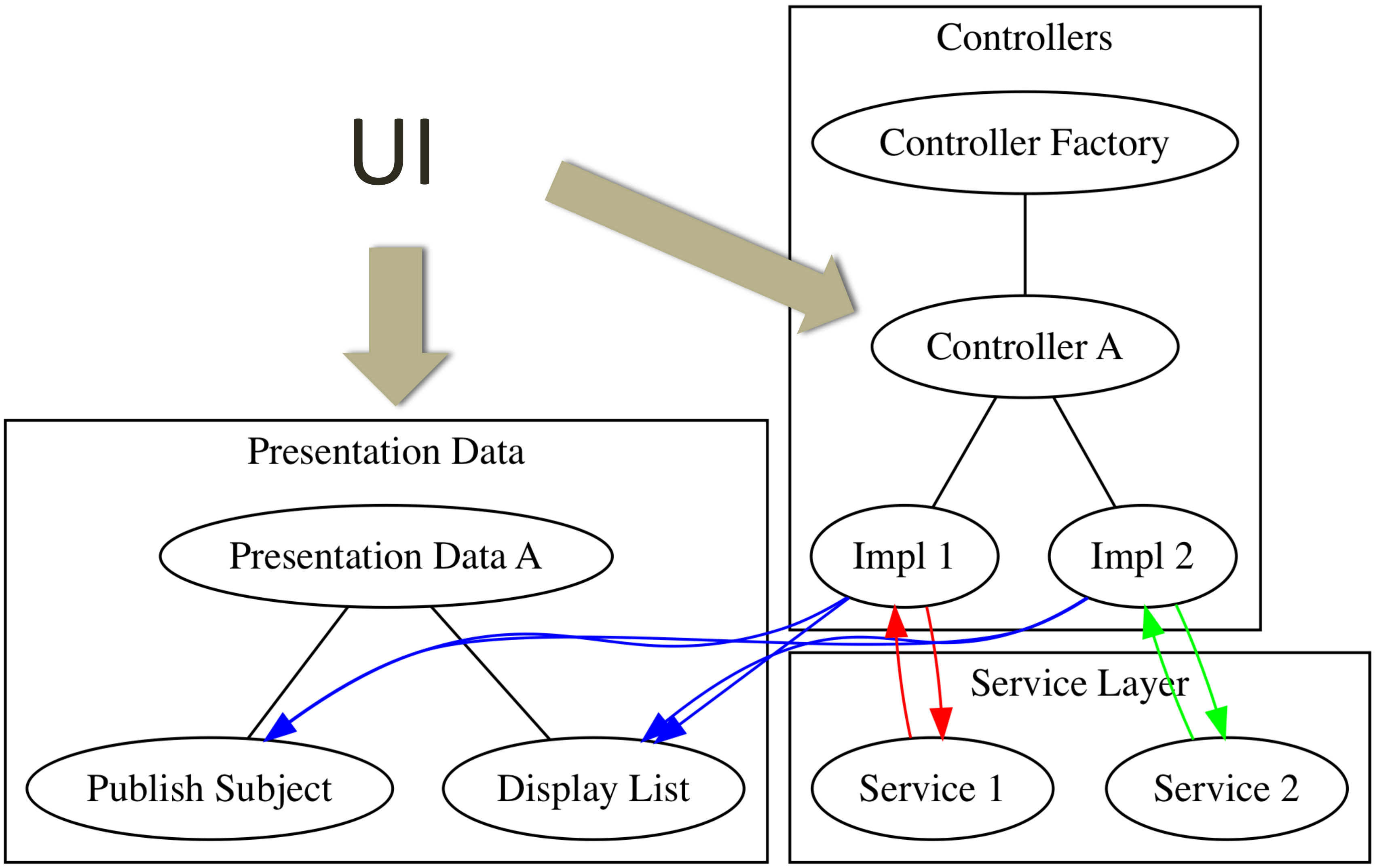
Components of a feature



Clearly defined flow of the application



Feature Change



Benefits

- Single Responsibility
  - Controller handles data
  - All networking contained in ServiceController utilizing retrofit
- Unit Testable (Nearly 100% coverage of Controller Classes)
- Sustainable (New features and refactors can be done in parallel and easily flagged on or off)
- Classes are small and easy to maintain
- Interdependencies reduced

Stumbling blocks

- Lots of code to handle simple feature
  - However, the UI is not aware of most of this and only has to know of two components Presentation Data and Controller

- Ramp up time
  - Has taken longer for new devs to get up to speed

Links

- http://square.github.io/retrofit/
- https://github.com/google/gson
- https://github.com/ReactiveX/RxJava
- https://github.com/gershwin88/charter_architecture_sample