D5 - Architecture & Design Document

Group 3

Alexander Lipianu (aglipian), Alex Pawelczyk (apawelcz), Kent Zhu (k36zhu), Sanketh Menda (sgmenda), William Chen (w279chen), Zhengyuan Gao (z73gao)

Section 1: Architecture of Git Trailblazer

The architectural foundation of Git Trailblazer is composed of the client-server and pipe and filter architectures. These architectures play a vital role in ensuring that Git Trailblazer meets the functional requirements and non-functional attributes described in the project proposal. The rest of this section details how the functional requirements and non-functional attributes are supported by the client-server and pipe and filter architectures. Moreover, we justify why we chose to use the client server and pipe and filter architectures for the development of Git Trailblazer.

Section 1.1: Client-Server Architecture

Users of Git Trailblazer (i.e. the clients) frequently communicate with Firebase, GitHub, and GitLab servers as they navigate throughout the app. Since the main purpose of the app is to provide users with recommendations of interesting repositories and issues from the GitHub and GitLab platforms, we needed to make use of the GitHub and GitLab APIs to access important data. We also integrate Git Trailblazer with Firebase because Firebase offers many useful features, such as secure authentication and scalable data storage. Figure 1 visualizes the component and physical representation of the system.

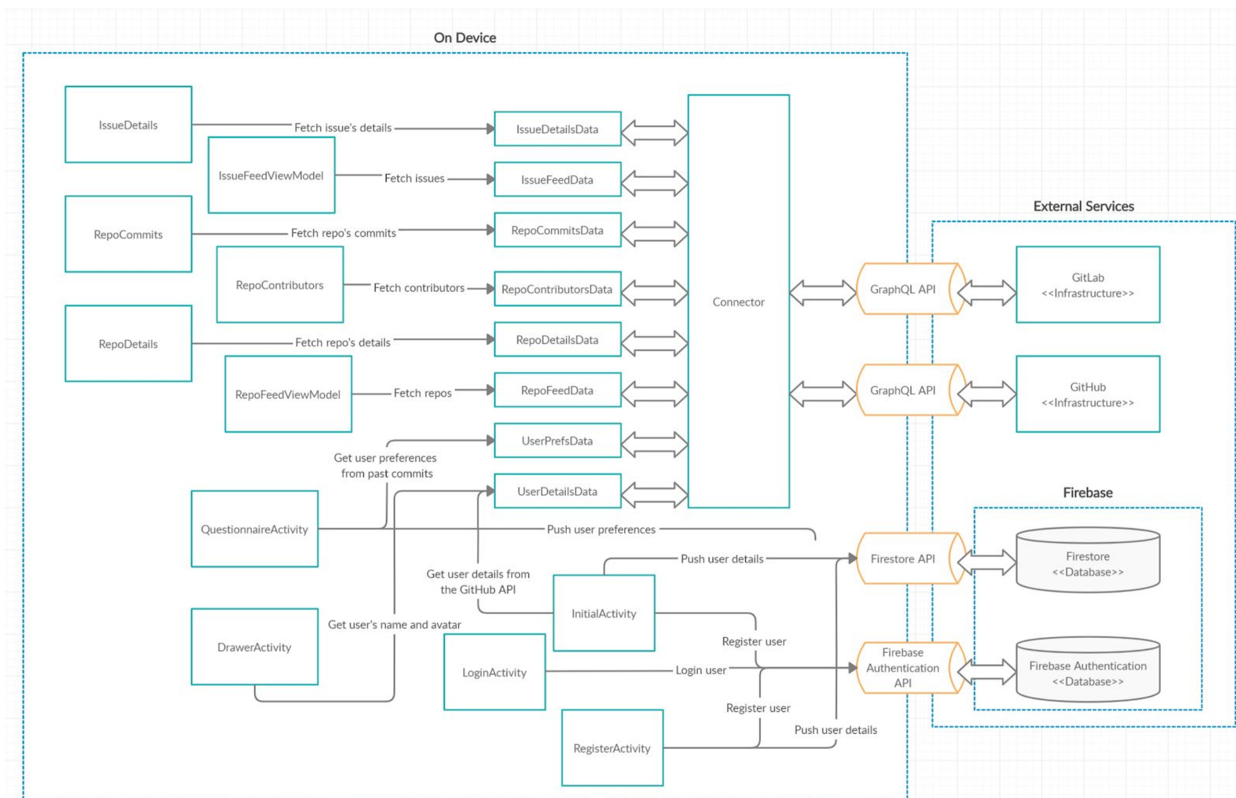

Figure 1. A depiction of the components and physical representation of Git Trailblazer's client-server architecture.

The use of the client-server architecture plays a crucial role in the successful implementation of Git Trailblazer's functional requirements. When a user creates an account, Firebase Authentication is used to store important account information, such as the user's email address, the sign in provider (i.e., GitHub or email), the account creation date, the date that a user was last signed in, and a unique user id. Logging in with GitHub facilitates communication with both GitHub and Firebase servers, while logging in with email only contacts Firebase servers. Firebase also handles other important account-management functionalities, such as logging out and account deletion. When users sign in for the first time, they are directed to a questionnaire populated with tags that might be of interest to the user (inferred from previous GitHub activity) and have the option to select tags that they are interested in. Firebase stores all selected tags in the Cloud Firestore database, and these tags are then used to instantiate a query for relevant repositories and issues to the user. The search bar can also be used to find repositories and issues that are relevant to the inputted query. Notifications and comments are two other functional requirements that use Cloud Firestore for scalable data storage. We use the GitHub and GitLab APIs to obtain the data that is displayed in the repository and issue cards, and querying for this data (based on user-inputted tags from the questionnaire or search bar) initiates contact with GitHub and GitLab servers.

The client-server architecture also supports various non-functional properties mentioned in the proposal:

- **Privacy/Security**: Since users are authenticated via Firebase, Git Trailblazer does not actively handle or store any of the user's personal data (like passwords). Instead, this sensitive information is handled by Google who is expected to keep up with good security practices (like salting and hashing passwords). Even though the GitHub API is used, the app does not log user's activities such as clicking on a repository within the App, nor does it retrieve and store any of the user's activity (that is, none of the results of the API calls leave the phone).
- **Evolvability**: It is easy to make changes by adding additional servers for new services and upgrading the server. For instance, new authentications can be easily added for platforms other than GitHub such as GitLab and Facebook by updating the connector.
- **Usability**: Integration with Firebase allows us to provide third-party logins which makes it easier for users to sign up. Further integration with GitHub allows us to infer user's preferences from their GitHub activity and use them to tailor the experience.
- **Readability**: By using external, mature, well-documented APIs like Firebase, GitHub, and GitLab, we make it easier to onboard new developers as they can read the existing documentation and consult existing questions on platforms like StackOverflow when stuck.

We chose to use the client-server architecture because it drives the fulfillment of many key functional requirements and non-functional properties of Git Trailblazer. The main purpose of the app is to present users with interesting repositories and issues from GitHub/GitLab. Hence, we needed to use the GitHub/GitLab APIs to communicate with their respective servers and fetch the required data. We also needed a way to store and manage user accounts; thus, we

leveraged the important backend services, user-friendly SDKs, and multiple authentication methods (e.g., email and GitHub) of Firebase Authentication. In addition, Firebase allows developers to create custom authentication methods, enabling Git Trailblazer to scale to more authentication methods in the future (e.g., GitLab or Bitbucket). Finally, we needed a database to store other important information, such as notifications and tags from the questionnaire. We chose to use Cloud Firestore because it provides a relatively easy way to store, sync, and query data for mobile apps in a scalable manner. Table 1 summarizes where the client-server architecture is implemented in the source code.

Table 1. Implementation details of client-server architecture.

Class Name	**Description**
package: *com.example.githubtrailblazer*	
InitialActivity	- *On-click listener for GitHub sign in:* Signing in with GitHub instantiates the connector with an API access token and queries the GitHub API for user data. Cloud Firestore is used to store the user data (i.e., GitHub ID, full name, username, and avatar URL).
RegisterActivity	- *On-click listener for account creation:* Uses email/password credentials to create an account and authenticate a user via Firebase Authentication. Further account details are stored in Cloud Firestore (e.g., user's full name). All error checking is handled by Firebase Authentication and Cloud Firestore.
LoginActivity	- *On-click listener for email/password sign in:* Firebase Authentication validates the inputted email/password combination and signs the user into the app if successful. All error checking is handled by Firebase Authentication and Cloud Firestore.
QuestionnaireActivity	- *Infer user preferences from GitHub:* On initialization, user preferences are inferred from the user's GitHub activity and then used to create a personalized questionnaire. - *Saving tags to Cloud Firestore:* Upon questionnaire submission, the tags that a user selected are stored in Cloud Firestore.
package: *com.example.githubtrailblazer.connector*	
Connector	- *Connector initialization:* Instantiating the (lazy-loading) singleton Connector creates two HTTP clients (GitHub and GitLab) that can send HTTP requests and read their responses.
***Data	- *Creating relevant data*: IssueFeedData and UserDetailsData is generated by querying the GitHub API via the connector, while RepoFeedData is generated by querying both GitHub and GitLab APIs via the connector.

Section 1.2: Pipe and Filter Architecture

Since our application supports multi-platform functionality with GitHub, GitLab, and potentially other platforms, we need a way that allows us to parse different formats of data into a consistent format. The pipe and filter architecture provides us with the functionality we need to perform successive transformations of data streams. We are able to implement independent, reusable filter components and use them repeatedly throughout our application to process data streams we receive from multiple servers. Figure 2 outlines the key classes and services that rely on the pipe and filter architecture.

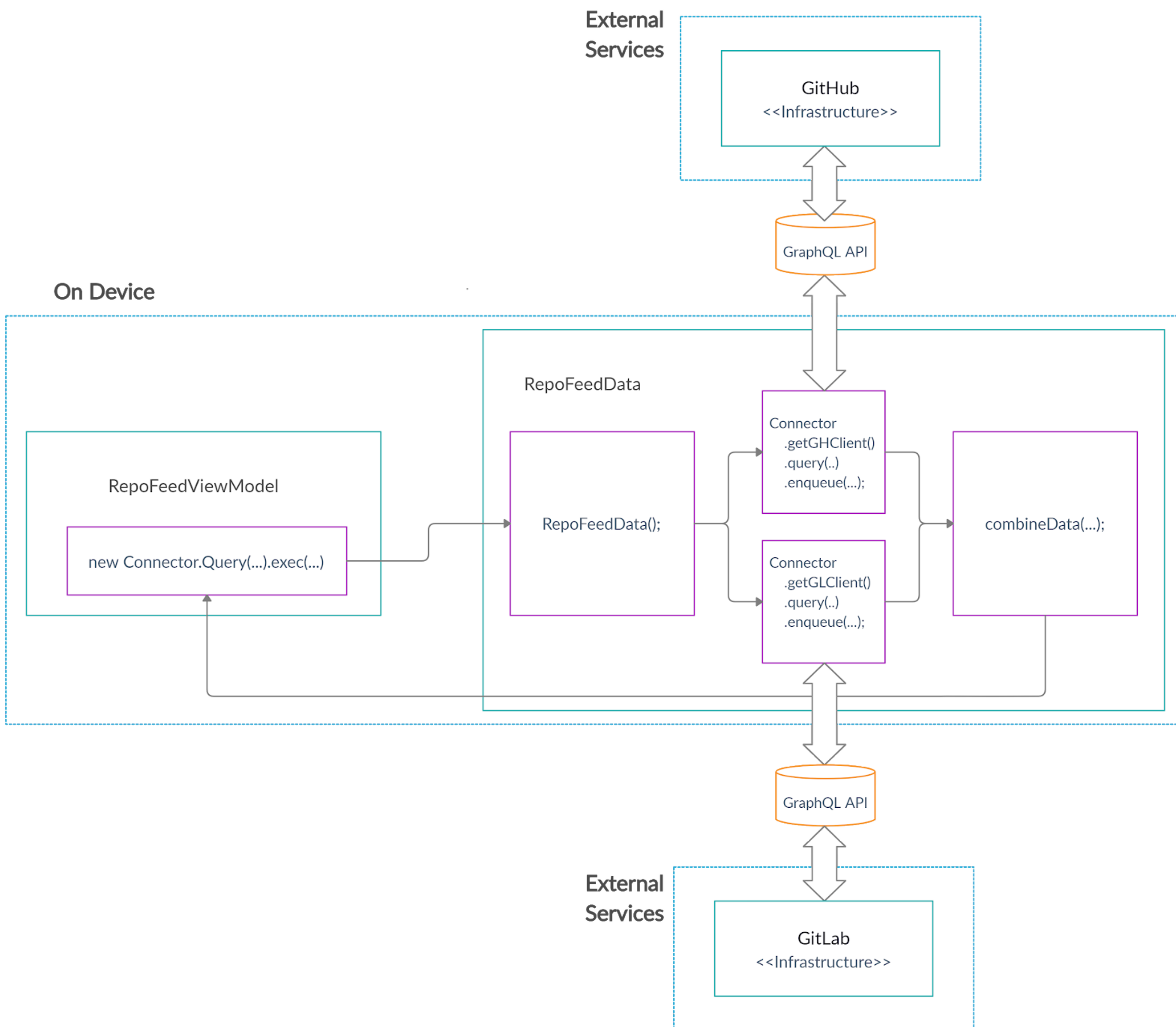

Figure 2: A depiction of the key classes and implementation details that represent Git Trailblazer's pipe and filter architecture.

We use the pipe and filter architecture to support various functional properties that are essential to our application. The pipe and filter architecture enables us to sort, filter, and combine data from multiple APIs, each of which entail different query structure and separate pagination. Moreover, since the API response data format we receive varies by platform, we map the data to a shared, consistent, platform-independent format using the pipe and filter architecture such that the API responses can be easily digested by the application's business logic. Users initiate third-party data filtering on the server side by either performing specific searches, scrolling through their explore feed, and/or choosing a preset repository filter option.

In the proposal document, we mention various functional properties of Git Trailblazer, including "scroll through a feed of repositories", "filter the feed of repositories by different metrics", "see starred repositories", and "see forked repositories". By using server-side filters, we are able to grab a list of repositories by certain criteria, which allows us to support searching repositories by a certain tag. Server-side filters also allow us to grab user-specific repositories such as starred repositories or forked repositories at little cost. Client-side filters give us the ability to sort the repository feed by different metrics, such as "newest" or "most stars", providing users with more flexibility and liberty in viewing the repositories they want.

The pipe and filter architecture also supports multiple non-functional properties mentioned in the proposal:

- **Evolvability**: By using the pipe and filter architecture to map different formats of data into one shared format, we currently are able to support two platforms. However, our application has the potential to expand to more platforms in the future given that we implement new filters to process the data.
- **Usability:** Instead of having two separate interfaces for viewing repositories from different platforms, we are able to combine them into one feed since pipe and filter processes the data into one shared format.
- **Fault tolerance:** Using the pipe and filter architecture to combine and filter multiple API data streams into one general data raises the fault tolerance of our repo feed to allow for possible API failures since it is highly unlikely that multiple APIs fail at the same time. Users are still able to receive content from the remaining functional data streams.
- **Readability:** The pipe and filter architecture is relatively comprehensible to new developers. Although filters are implemented differently, they share the same structure in reformatting the data stream thus can be easily referenced when a new developer implement new filters.

The pipe and filter architecture provides essential utility to our application since the services that we provide are heavily dependent on external APIs. By using this architecture, we are able to support multiple platforms with little extra cost, allowing for further evolvability in the future. In a team-based environment, it is also comprehensible thus easily maintainable by different developers. Our main repo feed can also be cohesive using one single data format processed from different sources, and one combined feed allows for higher fault tolerance in the scenario of API failures. In addition, the pipe and filter architecture satisfies our user features to search,

sort, and filter through different repositories, giving more depth to our application functionality. Table 2 illustrates how the pipe and filter architecture is used in our code.

Table 2. Implementation details of pipe and filter architecture.

package: *com.example.githubtrailblazer.connector*	
Class Name	**Description**
RepoFeedData	Fetches repository feed data from GitHub and GitLab APIs, then filters, sorts, and combines the different repository data formats into a shared data type.

Section 2: Design Patterns

The lazy loading singleton and model-view-controller (MVC) are two key design patterns used for the development of Git Trailblazer. This section details these patterns, along with important classes, abstractions, and data structures that are critical to our app's success. Moreover, we justify our design choices and explain how coupling is minimized. We also compare our design with alternative approaches, identify areas for future development, and detail how our chosen design can handle these new requirements.

Section 2.1: Singleton for GitHub and GitLab API Calls

We chose to use the *lazy-loading singleton* design pattern for one of the key components of our project: the GitHub/GitLab Connector (see Figure 3). We preferred the lazy-loading version of the singleton pattern because this class is memory intensive and choosing to initialize it when first required reduces initial loading time and makes our app more efficient. The Connector is physically located on the phone and facilitates communication between GitHub/GitLab API sites backed by GitHub/GitLab servers. On the backend, a HashMap is leveraged by the Connector for query abstraction, which allows us to add new queries easily; in particular, we have Data classes that make the GraphQL query and do the necessary post processing before giving a SuccessCallback which can be used by other components. Git Trailblazer does not incorporate any algorithms itself, but the GitHub/GitLab APIs use algorithms to filter and sort queries.

```
1   public class Connector {
2       private ApolloClient ghclient = null;
3       private ApolloClient glclient = null;
4       private boolean isInitialized = false;
5
6       // Implementing the Singleton pattern with a private initializer.
7       private static volatile Connector instance = null;
8       private Connector() {}
9
10      // function adapted from https://en.wikipedia.org/wiki/Singleton_pattern#cite_ref-6
11      public static Connector getInstance() {
12          if (instance == null) {
13              synchronized (Connector.class) {
14                  if (instance == null) {
15                      instance = new Connector();
16                  }
17              }
18          }
19          return instance;
20      }
21
22      public ApolloClient getGHClient() throws Exception {
23          if (isInitialized) {
24              return ghclient;
25          }
26          throw new Exception("Trying to access uninitialized GH client!");
27      }
28
29      [...]
30  }
```

Figure 3. An abridged version of the Connector class emphasising the lazy-loading Singleton pattern.

The main reason that we chose to use the lazy-loading singleton pattern is that we only need one instance of the Connector at any time. To meet this requirement, the singleton Connector (see Figure 4) prevents the app from instantiating new objects when they are not needed. In addition, the app avoids costly instantiation overhead (e.g., two HTML clients for each instantiation). Having only one instance of the Connector also reduces the complexity of the development process because we do not need to manage multiple Connectors. Since all alternative approaches allow multiple instances of the Connector to exist, we decided to use the singleton approach.

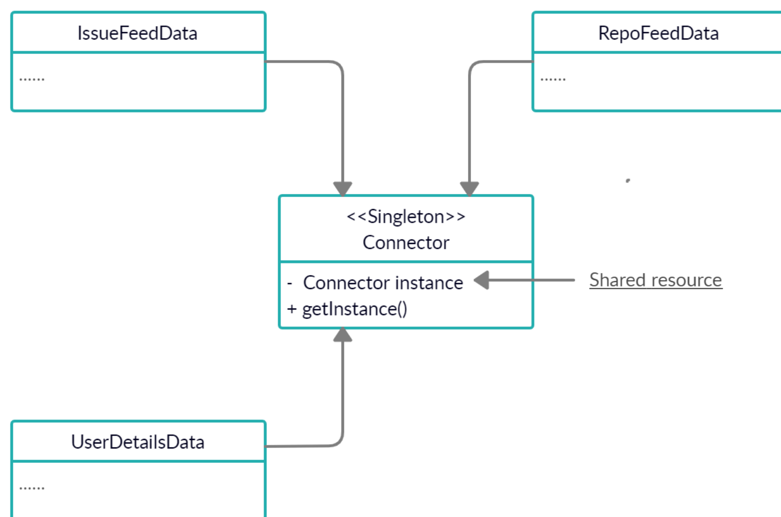

Figure 4. Connector is one of the Singleton classes in Git Trailblazer. The app only keeps one instance of the connector, IssueFeedData, UserDetailsData and RepoFeedDate all collect data from Connector.

Employing the lazy-loading singleton design pattern minimizes coupling because there is only one instance of the connector. This results in a low memory footprint and allows us to, in the future, implement API rate limiting. The singleton design pattern can also be useful for implementing future functional requirements. For example, if we ever need to include issues and repositories from other platforms (e.g., BitBucket or SourceHut), the connector can scale to support their API providers. Specifically, the singleton Connector with query abstraction abstracts which services are used to retrieve the data, while the ***Data classes abstract how the data is retrieved and how the many data streams are combined and mapped to a 'common/shared' format. As our app continues to grow in popularity, we plan to enhance the singleton Connector to support API rate limiting. More users means more API calls, and it is important to control this to prevent heavy computational and financial costs. Moreover, using an alternate design would be costly because it requires an expensive synchronization between the concurrent instances. Table 3 outlines where the lazy-loading singleton pattern is implemented in the source code, and a code snippet of an example Data class can be seen in Figure 5.

Table 3. Implementation details of lazy-loading singleton design pattern.

package: *com.example.githubtrailblazer.connector*	
Class	**Description**
Connector	- *Connector initialization:* We use a private and static initializer to implement the lazy-loading singleton pattern. Instantiating the Connector creates two singleton HTTP clients (GitHub and GitLab) that can send HTTP requests and read their responses.
***Data	- Creating relevant data: IssueFeedData and UserDetailsData is generated by querying the GitHub API via the connector, while RepoFeedData is generated by querying both GitHub and GitLab APIs via the connector.

UserDetailsData.java

```java
package com.example.githubtrailblazer.connector;

import ...
// Import generated class corresponding to the GraphQL query
import com.example.githubtrailblazer.github.UserDetailsQuery;

public class UserDetailsData {
    // Properties accessible in success callback
    public String id;
    public String name;
    public String username;
    public String avatarUrl;

    /**
     * Create user details data by querying API via connector
     * NOTE: must be PUBLIC and must have the following signature (for reflection):
     *        UserDetailsData(Connector.QueryParams, Connector.ISuccessCallback, Connector.IErrorCallback)
     * @param queryParams - the parameters
     * @param successCallback - the success callback (may be NULL)
     * @param errorCallback - the error callback (may be NULL)
     */
    public UserDetailsData(@NotNull Connector.QueryParams queryParams,
                           Connector.ISuccessCallback successCallback,
                           Connector.IErrorCallback errorCallback) throws Exception {
        final UserDetailsData _instance = this;
        Connector.getInstance().getGHClient().query(UserDetailsQuery.builder().build())
            .enqueue(new ApolloCall.Callback<UserDetailsQuery.Data>() {
                @Override
                public void onResponse(@NotNull Response<UserDetailsQuery.Data> response) {
                    UserDetailsQuery.Data data = response.getData();
                    if (data != null) {
                        id = data.viewer().id();
                        name = data.viewer().name();
                        username = data.viewer().login();
                        avatarUrl = data.viewer().avatarUrl().toString();
                        if (successCallback != null) successCallback.handle(_instance);
                    } else if (errorCallback != null) {
                        errorCallback.error("Failed query: data is NULL");
                    }
                }

                @Override
                public void onFailure(@NotNull ApolloException e) {
                    if (errorCallback != null) errorCallback.error("Failed query: " + e.getMessage());
                }
            });
    }
}
```

Figure 5. An abridged version of the UserDetailsData class demonstrating how a ***Data class looks.

Section 2.2: MVC Design Pattern

We've decided to separate the code related to distinct, repetitive standalone entities that appear in our application into components. The components then further separate their own user interfaces, state management, and event handling into individual view, model, and controller classes respectively (MVC).

This design allows components to perform separation of concerns as follows:

1) The view (user interface) is responsible only for rendering the latest component state data provided by the model when a change in the model state occurs, as well as relaying user interface events to the controller.
2) The controller is responsible only for mapping user interface events to their corresponding model actions.
3) The model is responsible only for managing component-related state, invoking external application-specific side effects when necessary, and broadcasting updated state to all dependent component views.

The component view, model, and controller classes follow specific implementation details to make way for some abstractions and enforce consistency across the codebase. The component controller(s) must extend some interface invoked by views when events occur, such as android.view.View.OnClickListener, enabling the functional abstraction of component view(s) that are able to set component controller instances as event handlers for view events.

The component view(s) must extend some descendant of the android.view.View class so that instances can be upcast to android.view.View, making way for the following functional abstractions:

- The component view(s) can be styled by creating a separate res/layout/*.xml file in which the view is included, styled with attributes, and supplied any necessary children.
- The component view(s) is instantiated by inflating layouts that include it.
- The component view(s) can find any children defined in the layout file from which it was instantiated by invoking this.findViewById(...), which can be manipulated by the view when component state changes and/or have their event listeners bound to the component controller.

The application's MVC components utilize several key design patterns which were taught in this course to implement the functionality outlined above. The observer design pattern is used in views to subscribe to a model instance, and in models to publish state updates to all subscribed views. The delegation design pattern is used in views and controllers by having the controller implement the necessary event handler interface, and having the view delegate the event handling behaviour to the controller. Finally, the state design pattern is often implemented in the models, with model functionality changing based on the user's account type and the third party

service data that they're interacting with (for example, a user cannot fork/star a GitHub repository if they did not authenticate with GitHub and thus possess a GitHub API OAuth token).

Application components follow specific class and package naming conventions. Each component is its own subpackage of the com.example.githubtrailblazer.components package consisting of at least 3 classes. The component class naming conventions and currently implemented MVC components are detailed below.

Table 4. Implementation details of MVC component.

package: *com.example.githubtrailblazer.components.<somecomponentname>*	
Class	**Description**
Model	The component model which manages component state, invokes application side effects, and broadcasts updated state to subscribed instances of SomeComponentName.
Controller	The component controller which invokes changes in an instance of Model based on received events.
SomeComponentName	A component view (user interface) which subscribes to an instance of Model and sets Controller as the event handler. **Note**: The component view is typically not named View to avoid clashing with the android.view.View class which is used heavily in the codebase. Also, a component may consist of multiple different component views (which share the same model and/or controller).

Table 5. Summary of current MVC components that are supported by Git Trailblazer.

Component Package	**Description**
repocard	The component responsible for displaying a repository's summary in card form.
searchbar	The component responsible for performing application search functionality.
toggle	The component responsible for selecting between different dropdown options.

MVC components enforce low coupling in the application by decoupling the component design, state management, and event handling from the class in which the component is inflated. MVC also enforces a high degree of cohesion within the component itself, as the view, model, and controller classes are highly specialized in the functionalities that they perform. As future requirements arise, existing parts of the application can be refactored into MVC components for

easy reuse if necessary. The view, model, and/or controller classes are modified as necessary when additional complexity is introduced at the component level. Due to the high degree of decoupling, component complexity can easily be scaled without having to worry much about breaking the rest of the application. An example of such a system evolution would be a new requirement where the search bar must show autocomplete suggestions for current trending searches - the bulk of the changes taking place in searchbar.Model.

There were two other design alternatives with regards to components that the team considered, but ultimately chose not to pursue. The first was to combine the component user interface, state management, and event handling into one all-encompassing class that extended a descendant of the android.view.View class. The main issue with this approach was that it exhibited poor separation of concerns, rendering the component incohesive and shifting the external application-specific side effect coupling from what would have been the component model into the component view + model + controller. The second option was to move the model and controller logic into the class that inflates, manages, and uses the high-level component state information. This component design ultimately resulted in implementation-dependent components that were not scalable or easy to extend due to high coupling with the class in which the component is created, managed, and displayed. This design would exhibit poor cohesiveness and add significant overhead to classes in the application that create, manage, and display the component, but are out of the component's scope.

Our application utilizes a variation of MVC known as MVVM (model, view, viewmodel) for the implementation of our application's navigation. An example of MVVM is the RepoDetailsActivity, which serves the purpose of displaying the full repository details. In the context of our application, MVVM and MVC can easily be swapped for each other.

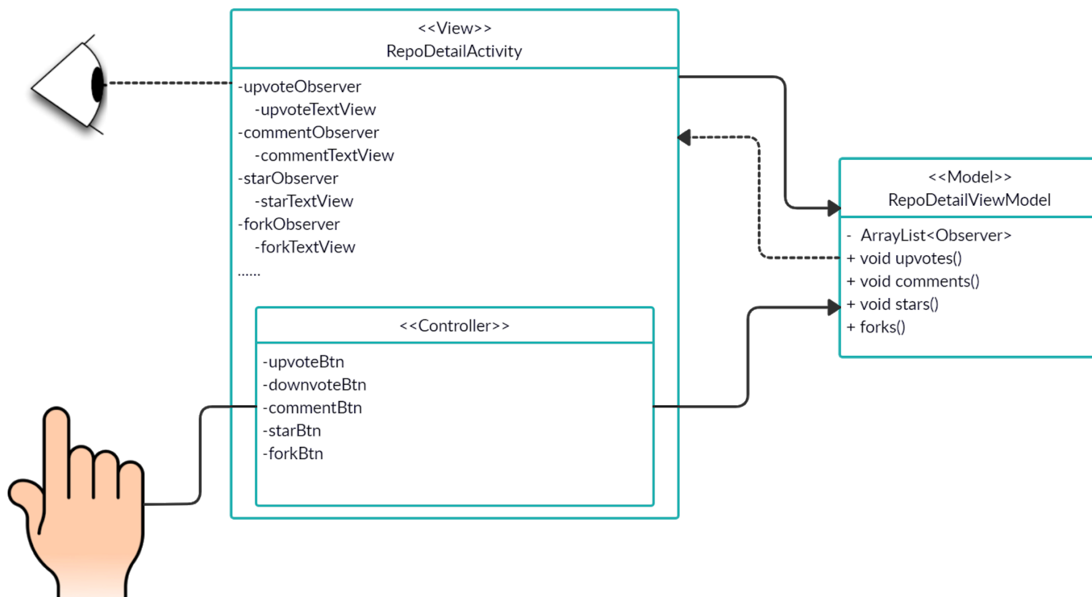

Figure 6. An MVVM pattern example of Repo Card. Users interact with the controller to modify the model and the change will be reflected on the card.

Section 2.3: Sequence Diagram

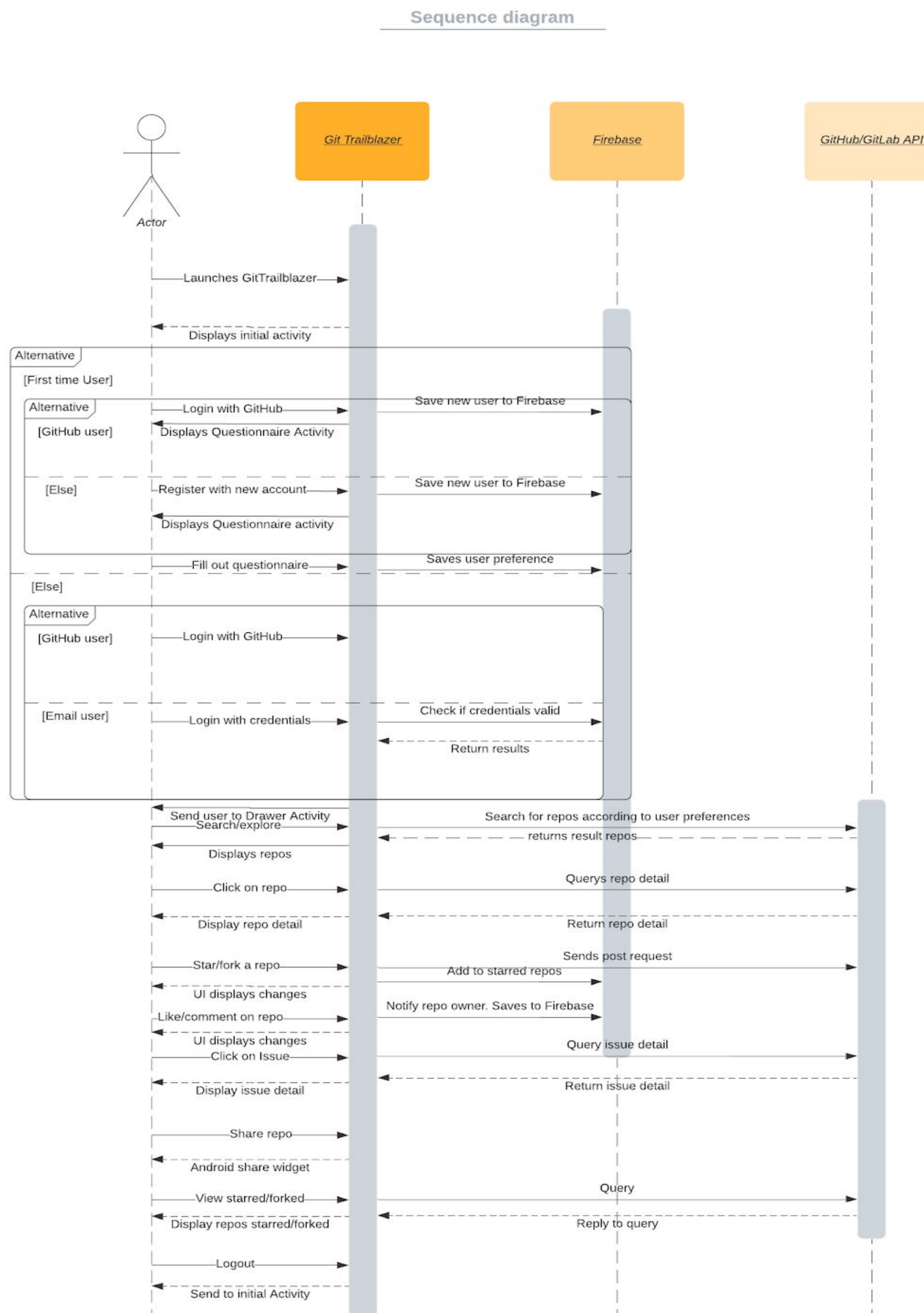

Figure 7. Sequence diagram that captures both user scenarios from the project proposal. Key functionalities that are displayed include: (i) GitHub authentication and sign in, (ii) the questionnaire activity, (iii) searching for and browsing through repositories/issues, (iv) clicking on repositories/issues to display further details, (v) starring and forking repositories, (vi) sharing repositories with Android Sharesheet, (vii) filtering repositories based on total stars/forks, and (vii) logging out.

Contributions

To-Date Contributions

1. Alexander Lipianu contributed by:
    a. implementing the Connector data queries, help design and create the Connector (Singleton pattern and Client-Server architecture)
    b. implementing the repository feed (MVVM pattern)
    c. implementing the search bar, repository card, and toggle components (MVC pattern)
    d. creating mockups for repo feed, issue feed, application navigation, and issue details
    e. implementing application navigation and toolbar
    f. helping create the notification feed UI
2. Alex Pawelczyk contributed by:
    a. implementing UI and backend for GitHub and email/password account creation, sign in, and authentication with Firebase (client-server)
    b. helping set up Cloud Firestore for storing detailed data of user accounts (client-server)
    c. helping implement the menu-options UI of the repository card (i.e., displaying drop-down list of menu options on a repo card) and implementing the backend of of the Android Sharesheet feature (i.e., the backend of the 'share' menu-option) (MVC design pattern)
    d. implementing the UI of the project pivot using mock data (i.e., showing mock contributors/contributions + historical data)
3. Kent Zhu contributed by:
    a. Help designing the structure of the user data stored in the Firestore.
    b. implemented the register process to link new github users to Firebase data store.
    c. Help implemented the module that query user information from Github api
4. Sanketh Menda contributed by:
    a. helping design and create the Connector (which follows the Singleton pattern and has a Client-Server architecture); and
    b. helping with the GitLab integration (this is also Client-Server).
5. William Chen contributed by:
    a. Mockup design iterations
    b. Implementing the front and backend of the repo details screen (Observer pattern)
6. Zhengyuan Gao contributed by:
    a. Implementing the questionnaire page's frontend and backend (interact with Firebase) (shown in D3).
    b. Implementing part of notification feed UI (bell button with count and recyclerview)
    c. Implementing notification feed backend [MVVM pattern and Adapter pattern] (synchronization with Realtime Database).

d. Helping design structure of user data in Firestore and Realtime Database.

To-Be-Done Contributions

1. Alexander Lipianu will contribute by:
    a. implementing the issue cards (MVC)
    b. implementing the issue feed (MVVM)
    c. implementing the issue details (MVVM)
2. Alex Pawelczyk and William Chen will contribute by:
    a. Implementing the backend of the project pivot (contributors and contributions for a specific repo) (client-server)
    b. Implementing UI and backend for displaying a user's contribution graph (time permitting) (client-server)
3. Kent Zhu will contribute by:
    a. Implement both the backend and UI of comment feature for Repo cards.
4. Sanketh Menda will contribute by:
    a. helping update the questionnaire to take into account the user's past contributions; and
    b. helping design and implement the issue feed
5. Zhengyuan Gao will contribute by:
    a. Implementing backend of upvote/downvote feature on repos. (including notifications for repo owner)
    b. Implement backend of starring/forking feature on repos.