About You

Why are you interested in working with Oppia, and on your chosen project?

- *Why **Oppia**?*

When I asked my seniors to suggest some open source organisations to contribute to, I got a few names. I looked at each of the organisations and **Oppia** stood out among them because of its mission/vision to provide education. Oppia's mission as stated on its website reads, *to help anyone learn anything they want in an effective and enjoyable way*. In this era where education is one of the most if not the most important things for anyone in the world, there are many people who are deprived of it due to certain reasons. But Oppia has taken the responsibility to provide free and quality education and learning to everyone in any part of the world through an online medium in the most effective way possible. I was fascinated by their beautiful mission and decided to contribute to this noble cause with my skills as much as I can. Therefore, I am interested to work with Oppia and be part of their generous mission.

- *Why the project - **Developer Options Menu**?*

Being a developer, I can feel how time-consuming it can be to do trivial tasks again and again to test a particular feature. I have also come across such situations while contributing to Oppia. Through this project, I want to reduce the time and effort of fellow developers in the Oppia team and make their lives a bit easier. Also, I always enjoy it when someone appreciates the UI that I have worked on, it makes me feel like my work is being noticed by the users :). Since this project contains work related to UI, I wanted to work on it. Also, I was excited by the fact that this project requires code with clean separation which I thought would be pretty amazing to explore. To write code with such separation could take time but in the long run, it will be very essential as it provides a clear understanding to future developers who will be working on the codebase. So, I decided to get hands-on experience with it. In short, this project suites my desires and skills and allows me to help fellow developers. Thus, I decided to take up this project.

Prior experience

Hi, I am Yash raj, an undergraduate student at the Indian Institute of Technology, Roorkee in the Department of Electronics and Communication Engineering. I didn't have much coding experience before my college but as soon as I joined my college, I started to explore various fields, one of which was *Android Development*. It has been around a year since I started doing it, and I never looked back. I am also part of the *Web Development* cell in my campus group Kshitij,

which is a group of literary enthusiasts trying to spread literature. Being a part of Kshitij, I was a major part of the team that worked on the development of our website to publish e-magazines. To explore the field of Android Development further, I also did an internship in my last semester where I was tasked to work with *Agora* and *Firebase* APIs and some UI components such as RecyclerViews and *ViewPagers* in their app. Later I got introduced to the world of open-source which I was amazed to realise that it contributed to such a large extent to the technology industry. I wanted to give my share of contributions to these organisations. Since then, I have been contributing to **Oppia** and have gained familiarity with the codebase and its architecture by working on several issues.

My **GitHub** profile can be found here.

My open-source contributions are as follows:

1. **Oppia:**
   PRs:
   a. #2948: Clickable text summary for inactive cards
   b. #2966: Create BUILD.bazel for threading package in utility module
   c. #2926: Add label for FAQSingleActivity
   d. #2653: Add Profile Flow
   e. #2700: Remove exploration_java_proto_lite from the model library

   Issues filed:
   a. #3039: No picture shown in Practice Mode

2. **Mifos Initiative:**
   PRs:
   a. #1658: Redesigned UI of the login screen
   b. #1614: Dismiss soft keyboard on background tap
   c. #1629: Fixed incorrect icon on no recent transactions found

   Issues filed:
   a. #198: Convert null check to kotlin style

Contact info and timezone(s)

Email: yashrajprime@gmail.com
Mobile No./Whatsapp No.: +91-9560137963
Gitter/GitHub: yashraj-iitr
Slack Email: yash_r@ec.iitr.ac.in
Timezone: Indian Standard Time (IST - +5:30 GMT)
Preferred Mode of Communication: Email, Gitter, Slack and Whatsapp

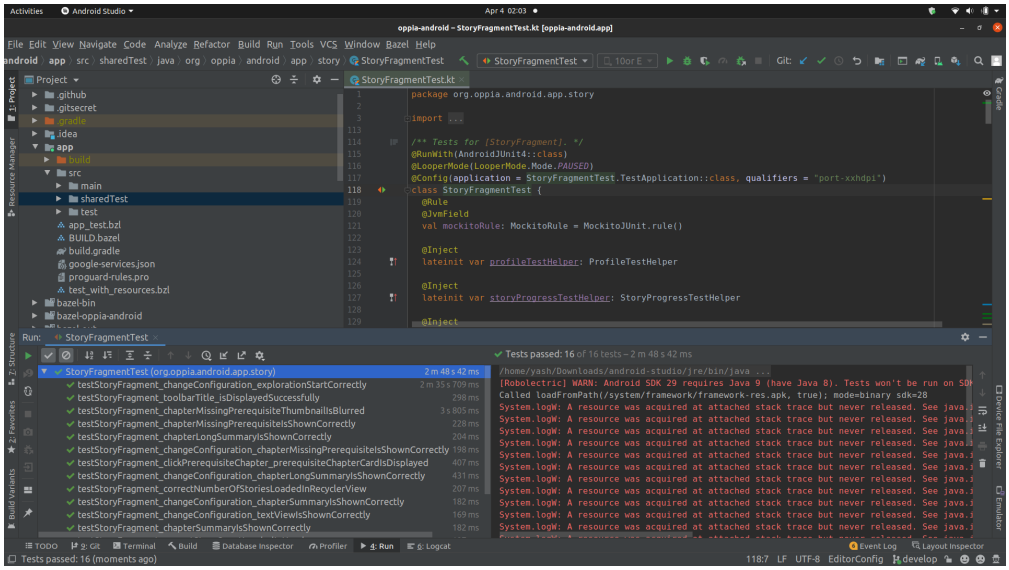
Time commitment

- I will be working on the project throughout the GSoC 10-week coding period from June 7, 2021 to August 16, 2021.
- I will be able to commit at least 4-5 hours per day during the coding period. I may go beyond my committed time if necessary.

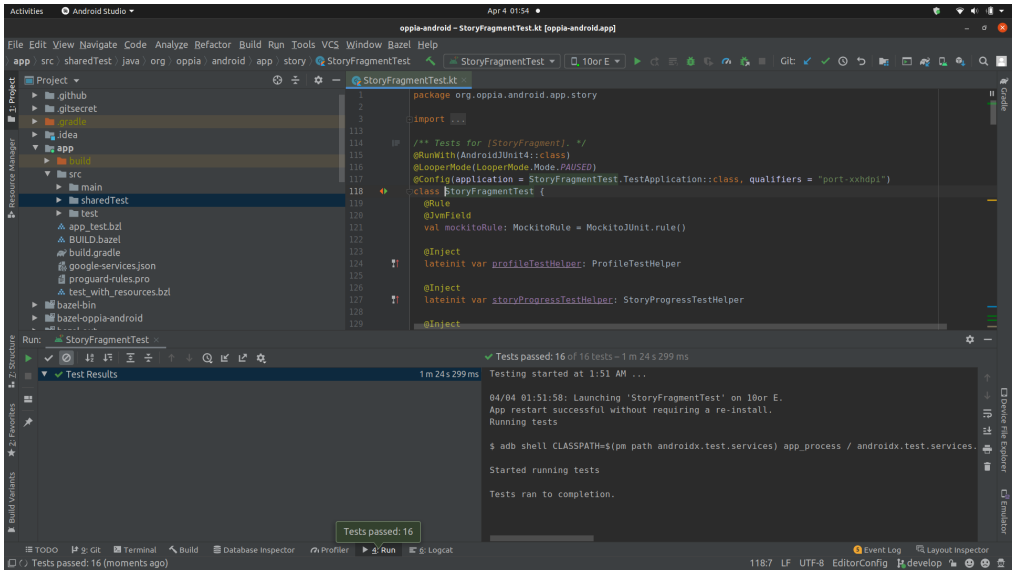
Essential Prerequisites

Answer the following questions (for Oppia Android GSoC students):

- *I am able to run a single Robolectric test target on my machine via Android Studio. (Show a screenshot of a successful test.)*



- *I am able to run a single Espresso emulator test target on my machine via Android Studio. (Show a screenshot of a successful test.)*



Other summer obligations

I have no other commitments during the summer except for an exam period of 10 days which will fall in the community bonding period. I will be able to work full-time on my GSoC project through my entire summer break.

Communication channels

I am reachable through various communication channels such as **email, gitter, slack, whatsapp** or any other proposed channel. We can also communicate through a planned video call if need be.

Application to multiple orgs

I am not applying to any other organizations other than **Oppia.**

Project Details

Product Design

What is this project?

- ➔ As the name of the project says, this project provides various useful options for developers to test and debug the app.
- ➔ It will allow the user to alter app settings or stored data in real time.
- ➔ Almost all major apps have such kinds of options and so should Oppia Android have them as well.
- ➔ All the features of this project are explained in later sections. These features will be very helpful during PR reviews, testing and debugging.

Who are the targeted users? How will it benefit?

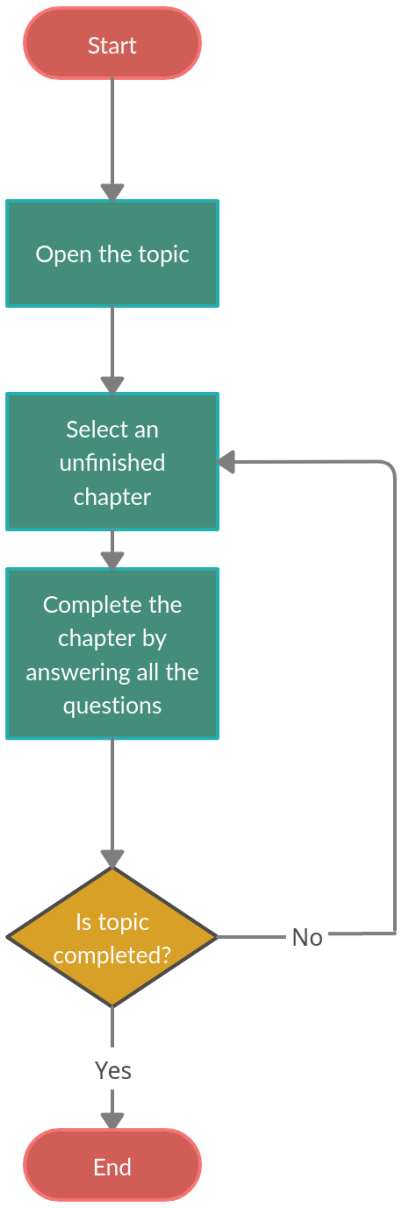
- ➔ This feature is targeted towards the developers on the Oppia team and contributors as well. It will reduce the time and effort of the development and testing process.

- ➔ There are many instances where we need to do certain database-related changes, like marking progress of a topic/story/chapter complete, and other things like crashing the app, while testing it. Doing these things manually takes up a lot of time and sometimes is even not possible to do so.

➔ This is where this feature will come in handy. With this, the developers can perform these tasks with just a click or two. ***Detailed user flow and mockups are shown later.***

Consider an example where we need to mark a topic as completed:

Without this feature:



If we follow this process for the **Fractions topic**, it will take us around **8-10 minutes** to complete it.

With this feature:

On the other hand, if we follow this approach, it will take us around **1 min** to mark the **Fractions topic** as completed.

We can see how useful this feature can be for developers and testers. What required a lot of time (**8-10 minutes**) can be achieved in a short amount of time (**around 1 min**). There are several more features which we will see later. With this example, I wanted to demonstrate the importance of this project in making the lives of developers easier.

What features are to be implemented?

- ➔ The ability to mark each of the following completed with all versions of each as available options
    1. Topics (*Mock is available here*)
    2. Stories (*Mock is available here*)
    3. Chapters (*Mock is available here*)

- ➔ View analytic event logs inside our app. (*Mock is available here*)

- ➔ App-wide behaviour changes
    1. Force all hints/solutions on by default (as a toggle) (Mock is available here)
    2. Force app to run in wifi/cellular/no network cases irrespective of the actual state on the phone (excluding impossible cases such as forcing wifi/cellular when there is no connectivity at all) (Mock is available here)

- ➔ An action to crash the app (for investigation & logging purposes) (Mock is available here)

Now, this project is required to be implemented only on the Bazel versions of the app, not on Gradle versions. So, first, we understand what Bazel is.

Bazel is an open-source build and test tool similar to Make, Maven, and Gradle. So, what makes Bazel different? Bazel has more structured configuration files than any other build format. It uses a human-readable, high-level build language. It supports large codebases across multiple repositories and large numbers of users. This is very essential when it comes to scalability.

Concepts and Terminologies related to Bazel

- ➔ **WORKSPACE:** The WORKSPACE file identifies the directory and its contents as a Bazel workspace, being present at the root of the project's directory structure. It has references for external dependency downloads needed to build the project.

- ➔ **BUILD:** The BUILD file has instructions/rules on how to run or build or test the project. It is equivalent to the build.gradle file in Gradle build tool.

- ➔ **Packages:** Any folder/directory which consists of the BUILD file is known as a Bazel package and the subfolders known as sub-packages.

- ➔ **android_library rules:** These rules help Bazel to understand how to build an Android Library Module from the given source file and its dependencies.

- ➔ **android_binary rules:** These rules help to build Android packages (.apk files)

Advantages of using Bazel over Gradle

- ➔ **High-level build language.** Bazel uses an abstract, human-readable language to describe the build properties of our project at a high semantical level. Unlike other tools, Bazel operates on the *concepts* of libraries, binaries, scripts, and data sets, shielding you from the complexity of writing individual calls to tools such as compilers and linkers.

- ➔ **Bazel is fast and reliable.** Bazel caches all previously done work and tracks changes to both file content and build commands. This way, Bazel knows when something needs to be rebuilt, and rebuilds only that. To further speed up our builds, we can set up our project to build in a highly parallel and incremental fashion.

- ➔ **Bazel is multi-platform.** Bazel runs on Linux, macOS, and Windows. Bazel can build binaries and deployable packages for multiple platforms, including desktop, server, and mobile, from the same project.

- ➔ **Bazel scales.** Bazel maintains agility while handling builds with a large number of source files. It works with multiple repositories and user bases in the tens of thousands.

- ➔ **Bazel is extensible.** Many languages are supported, and we can extend Bazel to support any other language or framework.

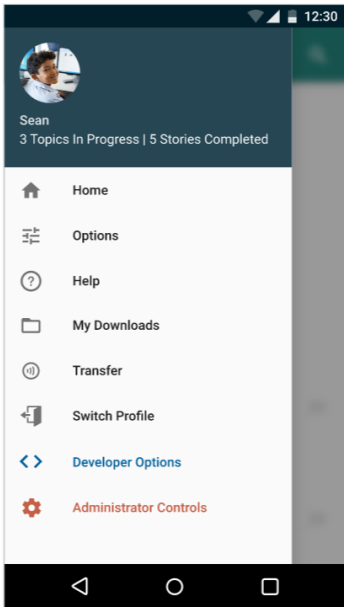
Overview of the implementation of the features:

Note: *All the mocks used here are just for* ***demonstration purposes*** *only. The mocks in this document might not match with the mocks provided in the links. This is because the mocks were updated after the completion of this proposal.*

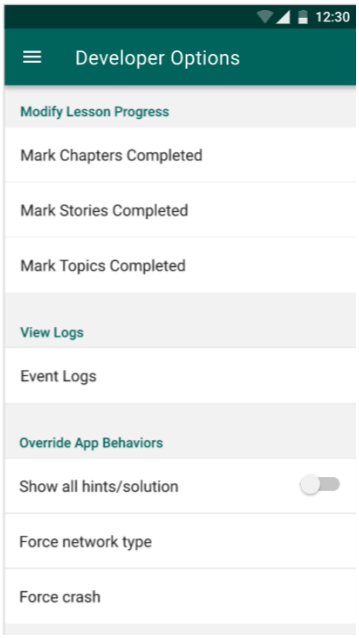
1. The first requirement of this project is that the feature should not be available/accessible to the end-user. It should be accessible only when the user is using a debug flavor of the app. The implementation of product flavors is beyond the scope of this project and will be implemented in future separately.
2. What we are currently trying to achieve here is that this feature should be accessible only when we are using the debug flavor of the app. In other words, we have to implement all these features in a separate module and then, including the module in the

binary will enable the features while not including it will disable the features. This feature will be accessible to all the users regardless of the fact that the user is an admin or not.

3. We can access the ***Developer Options*** menu from the *Navigation Drawer as shown below:*



4. We can see that there are several options present inside *Developer Options.* We will go through them one by one:

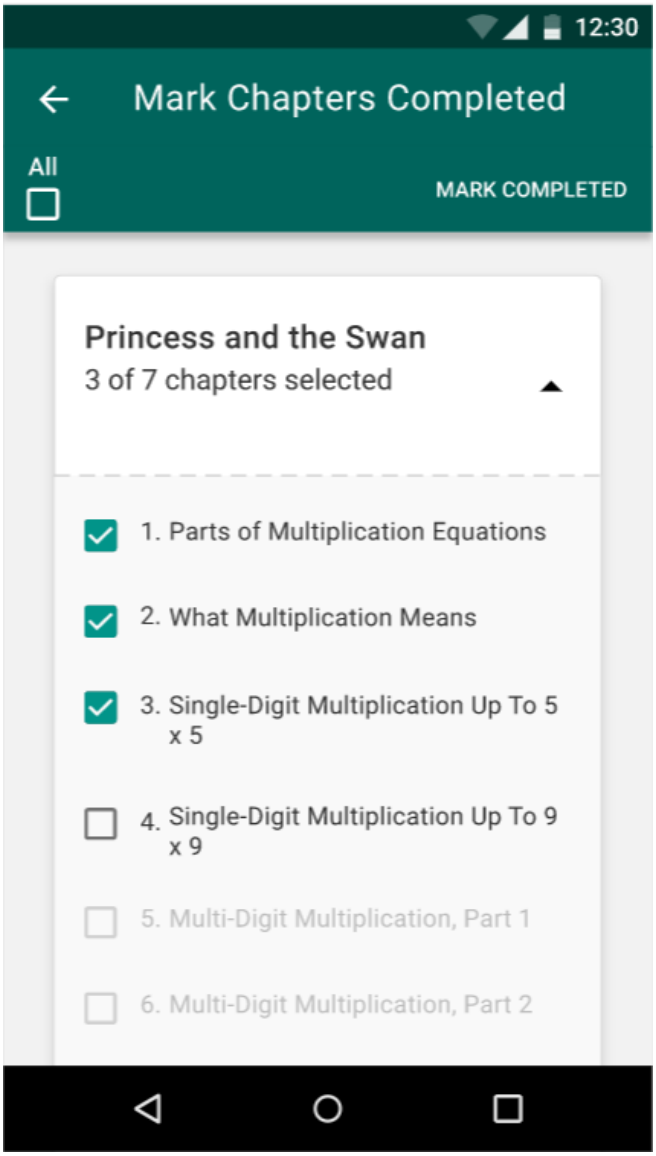


- **Modify Lesson Progress:**

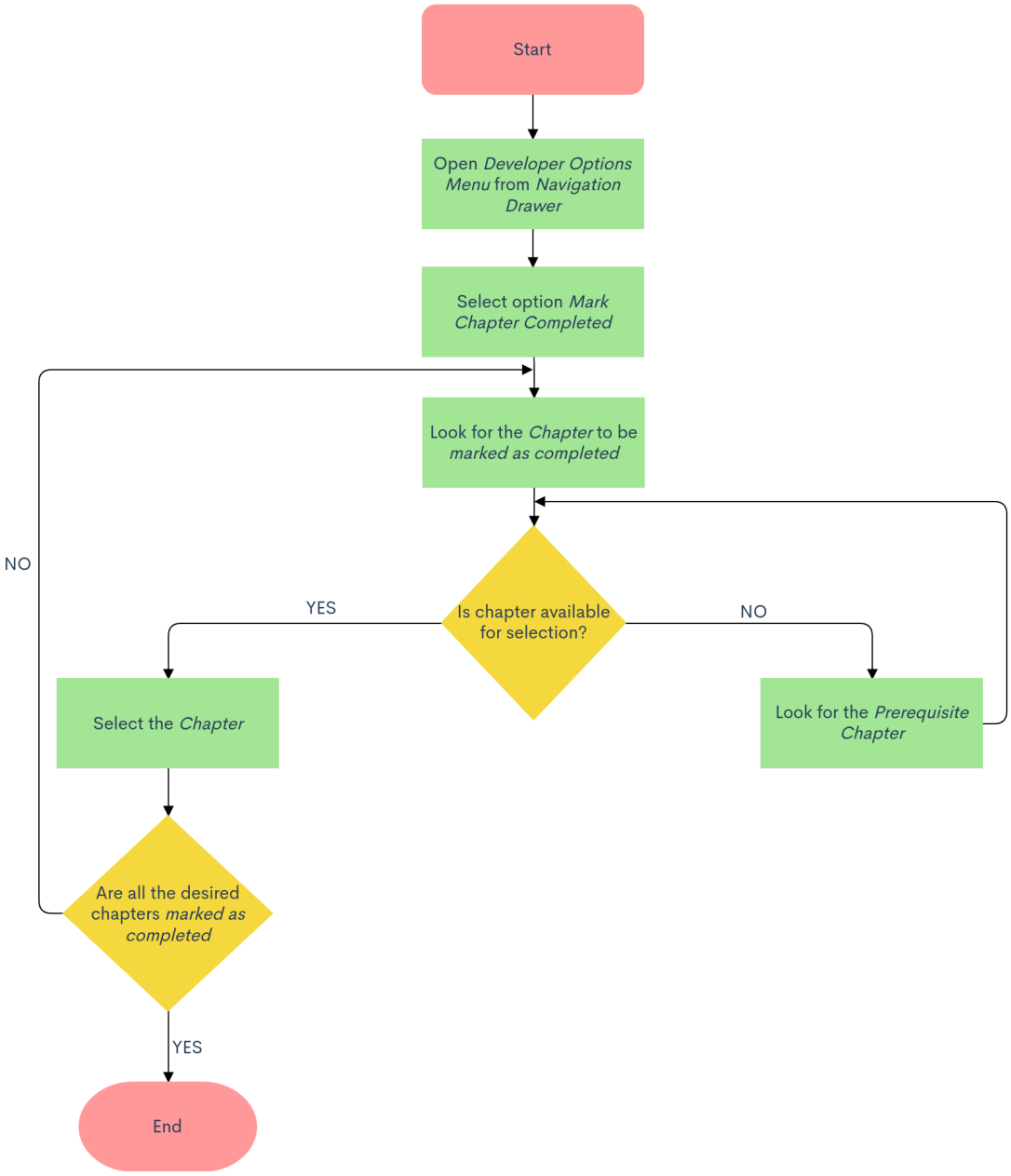
  As discussed earlier in an example, these features are very useful for developers and testers. **Issue:** **#2630** is a great example where these features would turn out to be very handy. It can reduce time and effort to a great extent. Now let's see each of the options in more detail:

  - **Ability to mark *Chapters* as completed:**

    This feature allows us to mark any chapter as completed. It can come handy when we need to check the app state after a particular chapter is completed. The mocks for the same are shown below:



USER FLOW:



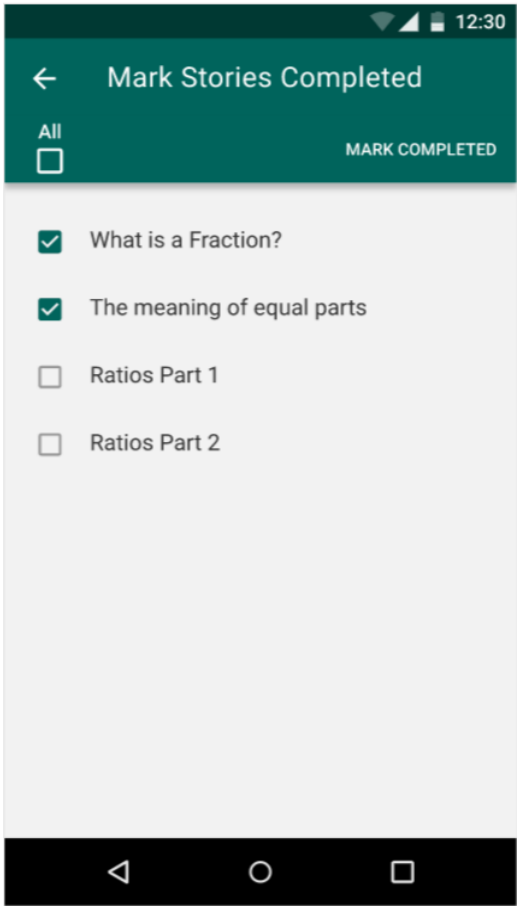
Note:

1. We must note that we **cannot** select a *Chapter* unless we have already selected all its *Prerequisite Chapters* or all the *Prerequisite Chapters* are already completed.
2. Also, the chapters in the list which are already completed before entering this screen/menu will be checked by default and will not respond to user interactions.

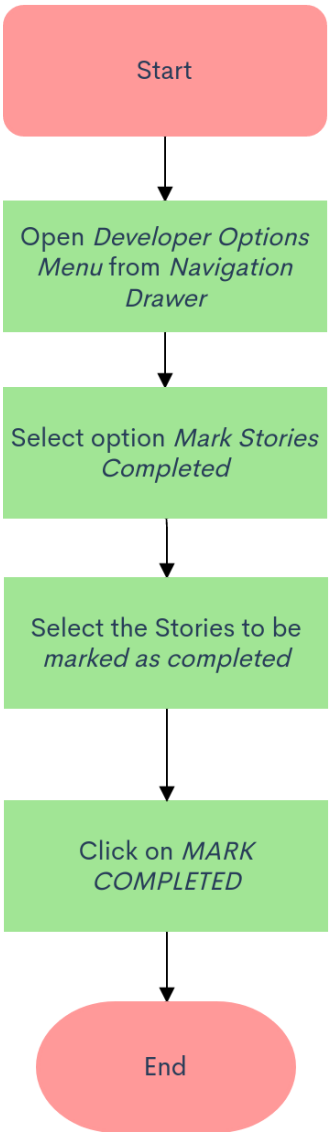
3. All the default UI possibilities based on current progress is shown below:
    a. **Chapter is checked:** It means the chapter is already completed.
    b. **Chapter is checked and active:** It means the chapter is not completed and is available for selection.
    c. **Chapter is inactive:** It means that the chapter is not completed and is not available for selection. We will first need to select its prerequisite chapter to make it available for selection.

- **Ability to mark *Stories* as completed:**

  This feature allows us to mark any story as completed. It can come handy when we need to check the app state after a particular story is completed. The mocks for the same are shown below:



USER FLOW:

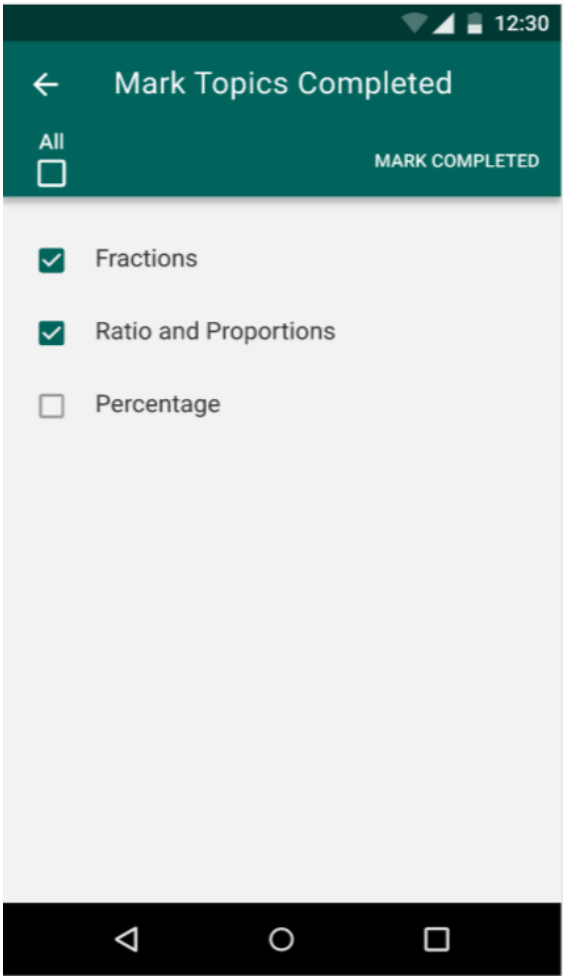


Note:

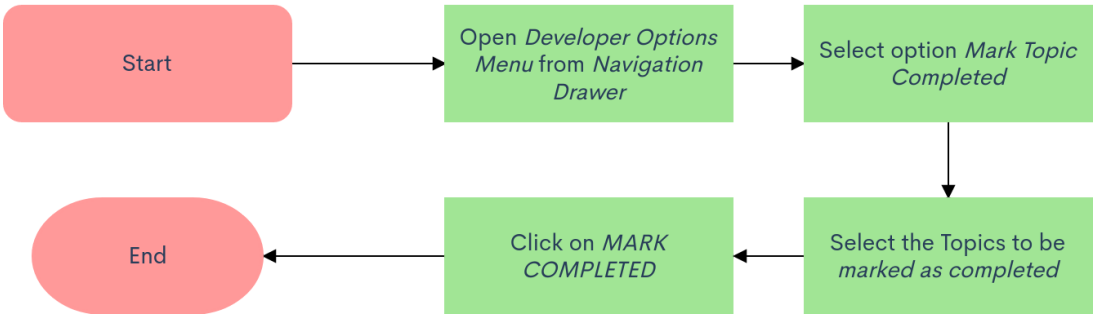
1. This feature works by traversing through all the chapters in the story and marking each of them completed, which in turn also marks the story as completed.
2. Also, the stories in the list which are already completed before entering this screen/menu will be already checked and will not respond to user interactions.
3. Unlike in the case of chapters, here all the stories will be available for selection.

- **Ability to mark *Topics* as completed:**

  This feature allows us to mark any topic as completed. It can come handy when we need to check the app state after a particular topic is completed. The mocks for the same are shown below:



USER FLOW:

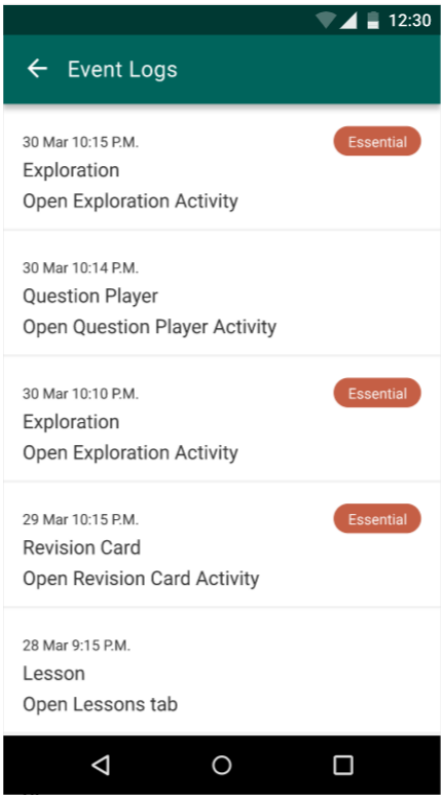


Note:

1. This feature works by first traversing through all the stories in the topic. Then, we traverse through all the chapters in each of the stories and mark each of them completed. This in turn will mark the topic as completed.
2. Also, the topics in the list which are already completed before entering this screen/menu will be already checked and will not respond to user interactions.
3. Similar to the case of stories, here also all the topics will be available for selection.

- **View analytic event logs:**

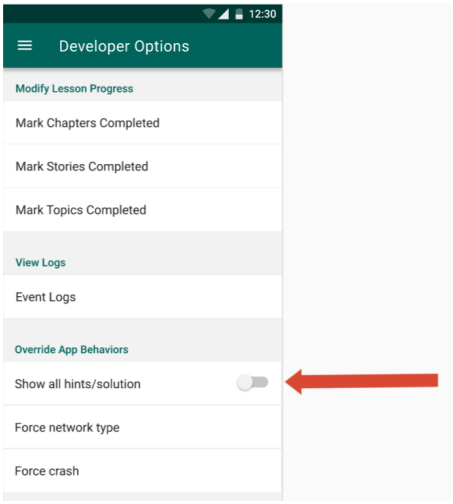
  Analytic event logs are the custom log reports that are sent to the remote service (in our case--Firebase as of now) for analysing our app usage by the users. These log reports are collected throughout the application process and provide us with vital stats for understanding and developing our product. The content of these reports depends upon the part of the codebase from where they are collected (eg- content collected from topics will be different from that collected from the exploration).



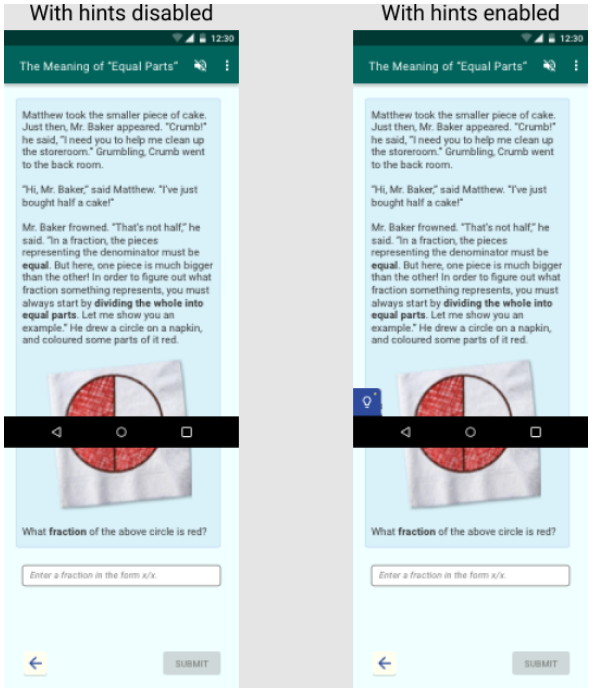
From the above mock we can observe that the logs are shown in the form of a list containing desired information. The information includes a tag/name of the log, date and time of the log, activity from where it is reported and a label showing whether the log is essential or not.

- **Show all hints/solutions:**

  This option provides us with a toggle to force enable all hints and solutions throughout the app. The following mock shows the option to enable this feature:



The following mock shows the desired outcome after enabling hints/solutions:



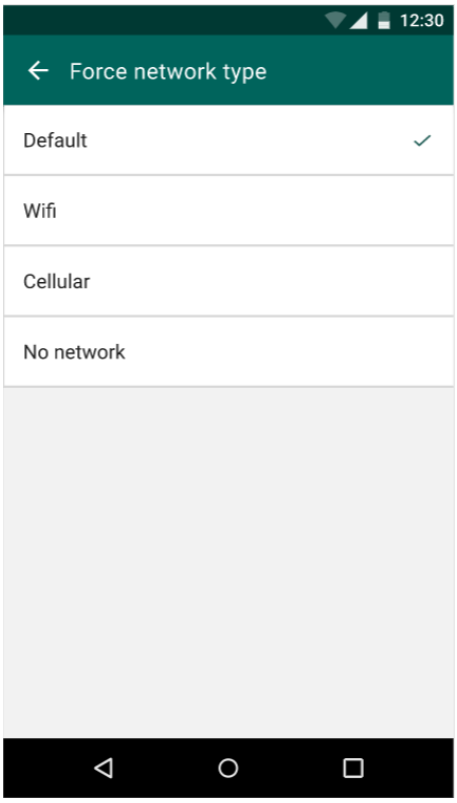
We can observe that here the hints are on by default, without the user giving any wrong answers. We should also note that the availability of the hints is subject to the condition that hints are available for that question i.e, if a question has no hints to it then we won't be able to see the hints. Also, we will be able to see all the hints and solutions based on availability in the *Hint Dialog* by clicking on the *Hint Bulb.*

We will see something like the below mock when we click the *Hint Bulb* (assuming this feature is enabled):

Note: The number of hints and availability of solutions may vary for different explorations.

- **Force Network Type:**

  This option allows us to force the app to run on a specific network state irrespective of the actual state of the device such as forcing the app to run on cellular state even if the device is connected to wifi. Though we need to take care to make sure impossible cases don't happen, such as forcing wifi/cellular when there is no connectivity at all. The below mock shows this option:



We can see that there are four options, each corresponding to forcing the app to run in that state, out of which only one can be selected. The availability of each option is subject to the actual state of the phone. For example, we won't be able to see the options *Wifi* and *Cellular* if there is no network connectivity on the device or we can show a toast/dialog stating that the feature can't be implemented when clicked on it in case of an impossible situation. Now, we will see what these network states mean:

1. **Default:** This option will be selected by default. It means that the app will work on the actual state of the device. For example, if the device is working on wifi, then the app will also work on *wifi network state*.
2. **Wifi:** This option will force the app to work on *wifi network state* irrespective of the actual state of the device. But this will not work if there is no network connectivity on the device. We can either not show this option or show a toast stating the error when clicked on it.
3. **Cellular:** This option will force the app to work on *cellular network state* irrespective of the actual state of the device. But this will not work if there is no network connectivity on the device. We can either not show this option or show a toast stating the error when clicked on it.
4. **No Network:** This option will force the app to work on *no network state* irrespective of the actual state of the device.

References:

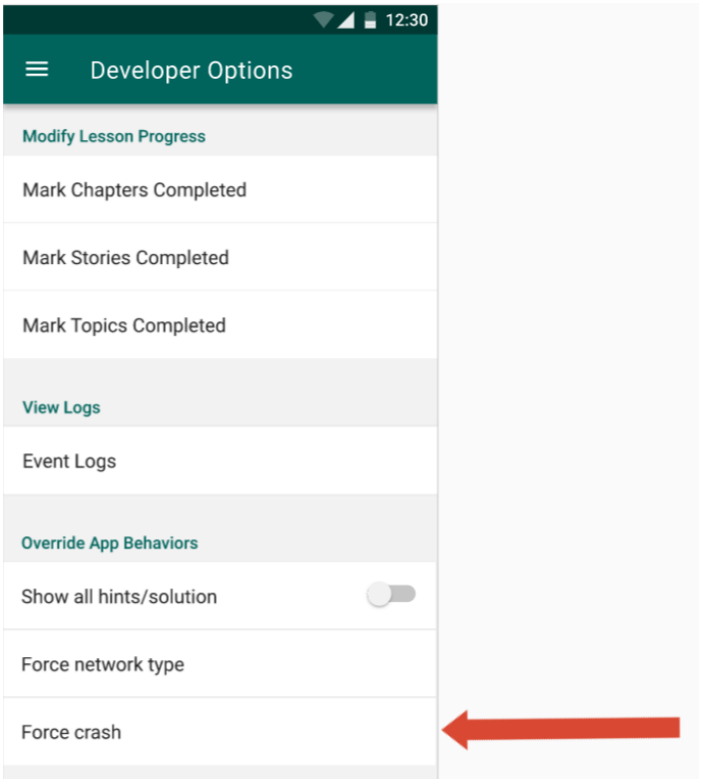
1. https://developer.android.com/reference/android/net/ConnectivityManager
2. https://developer.android.com/reference/android/net/NetworkCapabilities

- **Force crash app:**

  This option provides us with the feature to forcefully crash the app when needed. This feature is very handy for investigation and logging purposes which helps us to debug our app quickly and smoothly. This feature will come in handy when we have to check the app behaviour on a crash i.e how all the controllers handle a crash. There are several methods to crash the app. I have provided one here:

  Reference: https://stackoverflow.com/a/11807360

  The below mock shows the option to force crash the app:



We have seen the product design and overview of the implementation. Now, we will go into the technical details of this project to understand the idea more clearly.

Technical Design

Architectural Overview

This project will affect the app architecture to some extent although it won't modify the existing code too much. Currently, there are five modules in the app architecture each corresponding to their specific functions.

1. **app:** This module contains all the activities and fragments, as well as the view, view model, and presenter layers. It also contains Robolectric test cases and integration/hermetic end-to-end tests using Espresso.

2. **data:** This module provides data to the application by fetching data from the Oppia backend, or by fetching data from an offline PersistenceCacheStore. This module is unit-tested with a combination of JUnit and Robolectric.

3. **domain:** This module contains the business logic of the application, including both frontend controller and business service logic. It is a Java/Kotlin library without Android components, and it is unit-tested using raw JUnit tests.

4. **model:** This library contains all protos used in the app. It only contains data definitions, so no tests are included.

5. **utility:** This is a Java/Kotlin module without Android dependencies. It contains utilities that all other modules may depend on. It also includes JUnit test cases.

This project will add another module to this architecture, lets call it ***dev*** *module*. This module will contain all the logic, activities, fragments, views, view models and presenter layers which corresponds to the *Developer Options*.

The reason that we need to add another module is that we don't want the code meant for developer options to mess up with actual code of the app. Also, we can easily include or exclude the module at build time depending on whether we want the feature in our app or not. This allows us to easily and effectively separate the code.

Now, we will need to have several subdirectories to further organise the features. Let's take a look at the proposed subdirectories:

1. **dev/developermenu:** it will contain the activity, fragment, presenters, view models and adapters for the *Developer Options* screen.

Potential list of files that will be added:

- ➔ **DeveloperOptionsActivity:** Activity file for developer options menu
- ➔ **DeveloperOptionsActivityPresenter:** Presenter for DeveloperOptionsActivity
- ➔ **DeveloperoptionsFragment:** Fragment for developer options menu
- ➔ **DeveloperOptionsFragmentPresenter:** Presenter for DeveloperOptionsFragment. Contains most of the UI logics
- ➔ **DeveloperOptionsViewModel:** ViewModel for developer options menu. Prepare and manage data
- ➔ **RouteToMarkChaptersCompletedListener:** Listener to route to MarkCompletedChaptersActvity
- ➔ **RouteToMarkStoriesCompletedListener:** Listener to route to MarkCompletedStoriesActvity
- ➔ **RouteToMarkTopicsCompletedListener:** Listener to route to MarkTopicsCompletedActvity
- ➔ **RouteToEventLogsListener:** Listener to route to EventLogsActvity
- ➔ **RouteToForceNetworkTypeListener:** Listener to route to ForceNetworkTypeActvity
- ➔ **ShowHintsAndSolutionListener:** Listener to enable/disable hints and solution
- ➔ **ForceCrashListener:** Listener to force crash the app

2. **dev/topic:** It will contain the activity, fragment, presenters, view models and adapters for the *Mark Topics Completed* screen.

   Potential list of files that will be added:

   - ➔ **MarkTopicsCompletedActivity:** Activity file for mark topics completed screen
   - ➔ **MarkTopicsCompletedActivityPresenter:** Presenter for MarkTopicsCompletedActivity
   - ➔ **MarkTopicsCompletedFragment:** Fragment for mark topics completed screen
   - ➔ **MarkTopicsCompletedFragmentPresenter:** Presenter for MarkTopicsCompletedFragment. Contains most of the UI logics
   - ➔ **MarkTopicsCompletedViewModel:** ViewModel for mark topics completed screen. Prepare and manage data
   - ➔ **TopicAdapter:** Adapter for recycler view containing list of topics
   - ➔ **TopicItemViewModel:** ViewModel for topic items in the list. Prepare and manage data

3. **dev/story:** It will contain the activity, fragment, presenters, view models and adapters for the *Mark Stories Completed* screen.

   Potential list of files that will be added:

   - ➔ **MarkStoriesCompletedActivity:** Activity file for mark stories completed screen
   - ➔ **MarkStoriesCompletedActivityPresenter:** Presenter for MarkStoriesCompletedActivity

- ➔ **MarkStoriesCompletedFragment:** Fragment for mark stories completed screen
- ➔ **MarkStoriesCompletedFragmentPresenter:** Presenter for MarkStoriesCompletedFragment. Contains most of the UI logics
- ➔ **MarkStoriesCompletedViewModel:** ViewModel for mark stories completed screen. Prepare and manage data
- ➔ **StoryAdapter:** Adapter for recycler view containing list of stories
- ➔ **StoryItemViewModel:** ViewModel for story items in the list. Prepare and manage data

4. **dev/chapter:** It will contain the activity, fragment, presenters, view models and adapters for the *Mark Chapters Completed* screen.

   Potential list of files that will be added:
   - ➔ **MarkChaptersCompletedActivity:** Activity file for mark chapters completed screen
   - ➔ **MarkChaptersCompletedActivityPresenter:** Presenter for MarkChaptersCompletedActivity
   - ➔ **MarkChaptersCompletedFragment:** Fragment for mark chapters completed screen
   - ➔ **MarkChaptersCompletedFragmentPresenter:** Presenter for MarkChaptersCompletedFragment. Contains most of the UI logics
   - ➔ **MarkChaptersCompletedViewModel:** ViewModel for mark chapters completed screen. Prepare and manage data
   - ➔ **StorySummaryAdapter:** Adapter for recycler view containing list of chapters
   - ➔ **StorySummaryViewModel:** ViewModel for story summary items in the list. Prepare and manage data
   - ➔ **ChapterSummaryViewModel:** ViewModel for chapter items inside the story summary item. Prepare and manage data

5. **dev/logs:** It will contain the activity, fragment, presenters, view models and adapters for the *Event Logs* screen.

   Potential list of files that will be added:
   - ➔ **EventLogsActivity:** Activity file for event logs screen
   - ➔ **EventLogsActivityPresenter:** Presenter for EventLogsActivity
   - ➔ **EventLogsFragment:** Fragment for event logs screen
   - ➔ **EventLogsFragmentPresenter:** Presenter for EventLogsFragment. Contains most of the UI logics
   - ➔ **EventLogsViewModel:** ViewModel for mark chapters completed screen. Prepare and manage data
   - ➔ **EventLogAdapter:** Adapter for recycler view containing list of event logs
   - ➔ **EventLogViewModel:** ViewModel for event log items in the list. Prepare and manage data

6. **dev/network:** It will contain the activity, fragment, presenters, view models and adapters for the *Force network type* screen.

   Potential list of files that will be added:
   - ➔ **ForceNetworkTypeActivity:** Activity file for force network type screen
   - ➔ **ForceNetworkTypeActivityPresenter:** Presenter for ForceNetworkTypeActivity
   - ➔ **ForceNetworkTypeFragment:** Fragment for force network screen
   - ➔ **ForceNetworkTypeFragmentPresenter:** Presenter for ForceNetworkTypeFragment. Contains most of the UI logics
   - ➔ **ForceNetworkTypeViewModel:** ViewModel for force network type screen. Prepare and manage data.
   - ➔ **ForceNetworkAdapter:** Adapter for recycler view containing list of network states
   - ➔ **ForceNetworkViewModel:** ViewModel for network state items in the list. Prepare and manage data

This module (***dev***) will most likely depend on ***app***, ***utility*** and ***domain*** modules.

1. **app module:**

   a. **app/activity:** As this module also contains activities, it will depend on the activity subdirectory of the app module.

   b. **app/fragment:** As this module also contains fragments, it will depend on the fragment subdirectory of the app module.

   c. **app/viewmodel:** We will need to handle Ui impressions for the activities and fragments mentioned above, so it will depend on the viewmodel subdirectory of the app module.

   d. **app/application:** This dependency is needed to implement the feature to *show all hints and solution.* We will see in the *Implementation Approach* section on how it depends on it.

2. **utility module:**

   a. **utility/networking:** It will depend on this module for implementing the *Force network type* feature as we will be using ***NetworkConnectionUtil***.

3. **domain module:**

   a. **domain/oppialogger:** As we have to work with the event logs, we will need ***AnalyticsController***. So, it will depend on this subdirectory of the domain module.

b. **domain/topic:** As we have to modify lesson progress, we will need ***StoryProgressController.*** So, it will depend on this subdirectory of the domain module.

Potential list of files that will be modified:

- ➔ **HintsViewModel**
- ➔ **HintsAndSolutionDialogFragmentPresenter**
- ➔ **StateFragmentPresenter**
- ➔ **QuestionPlayerFragmentPresenter**
- ➔ **AnalyticsController**

We will see in more detail on how these files are modified in the *Implementation Approach* section.

We have seen an overview of the app architecture of this project. Now, we will see in detail how all of these can be implemented.

Implementation Approach

The first and the foremost feature of this project is to separate all the code and logic pertaining to *Developer Options* from the rest of the code. This allows us to easily include/exclude the module which in turn enables/disables this feature. There are a few approaches to achieve this. I'll mention the other ones in the **Appendix** section at the end as instructed. Now, the approach that I have come up with to implement it in Bazel version of the app is as follows:

Reference: **Configurable build attributes - Bazel 4.0.0**

- First, we will create a module (*say* ***dev***). Now, we need to include or exclude this module at build time. For this, we will use ***Configurable Build Attributes*** provided by Bazel.

- Now, in the **app/BUILD.bazel** file, we will define a ***config_setting*** as follows:

```
config_setting(
    name = "dev",
    define_values = {"is_dev": "true"},
)
```

Here, we have assigned the name as ***dev.*** Also, we have named the flag as ***is_dev*** and its value as ***true.***

Note: All these names can be changed to any appropriate names, if needed. We just have to refactor the rest of the code that follows accordingly.

- Now, we will use the ***select()*** function provided by bazel. The code is shown below:

```
select({
   "dev": ["//dev"],
   "//conditions:default": [],
})
```

We will add this statement in **deps** argument of the kt_android_library labeled as **app**. The code is shown below:

```
deps = [
      ...
      ...
] + select({
   "dev": ["//dev"],
   "//conditions:default": [],
}),
```

The dots represent all other dependencies which are already added. Notice that we use a plus (**+**) sign to add the select statement to the existing dependencies.

- All the steps mentioned above will allow us to include or exclude the **dev** module at build time. We can do this by using this command to build the app:
    - ***dev*** *module excluded:*

      ```
      bazel build //:oppia
      ```

      We can note that there is no change in the build command from the current one. This is because there is no change in the app in this version.

    - ***dev*** *module included:*

      ```
      bazel build //:oppia --define is_dev=true
      ```

      We can notice that in this case we need to add build flags in our build command.

- Now lets see how these commands work.

    - When we add the build parameter **is_dev=true**, it checks the build files for the **config_setting** which matches these parameters. We can see that in our case the **config_setting** named ***dev*** contains the same parameter and value pair. This means that this config_setting returns true when used in a ***select()*** function.

    - Now we go to the select statement in the **app** library. We can see that it contains a condition with label **dev** and a default condition. Since, we had specified the build parameters this select statement will return the value with the condition **dev** which is a label with name **//dev.** This means that the **app** library now also depends on **//dev** module.

    - If we do not specify the build parameters the **dev** config_setting will return false when used in a ***select()*** function. Since our ***select()*** function contains only one condition and a default condition, it will select the ***//conditions:default*** condition and return its value which in our case is empty. Thus, the **app** library will not depend on the **//dev** module in this case.

We have seen how to include or exclude the **dev** module at build time. Now, we will see how this checks on whether the user accesses the *Developer Options* or not.

- First, we will create a clickable layout in the ***drawer_fragment.xml*** for the option of *Developer Options* similar to the one for *Administrator Controls*. We will set the default visibility to ***GONE.***

- Now, inside the **subscribeToProfileLiveData()** function inside **NavigationDrawerFragmentPresenter.kt** we will add a *try and catch* block similar to the one below:

```
try {
  Class.forName("org.oppia.android.dev.DeveloperOptionsActivity")
  Log.d("DeveloperOptionsActivity", "AVAILABLE")
} catch (e: Exception) {
  Log.d("DeveloperOptionsActivity", "NOT AVAILABLE")
}
```

- What this try and catch block does is that it checks whether the **OptionsActivity** in the dev module is present or not. It will throw an exception when we don't build the app with the specified build parameters.

- Now, when the **OptionsActivity** is present, we will set the visibility of the *Developer Options* option to ***VISIBLE.*** Otherwise, it will remain ***GONE.***

- Now, in the same **subscribeToProfileLiveData()** function inside **NavigationDrawerFragmentPresenter.kt,** we will set the **onClickListener()** for *Developer Options* option. We will again use the *try and catch* block for the sake of preventing any exception and app crash which shouldn't happen under normal circumstances. The *try and catch* block will contain the code to start the **OptionsActivity** of the **dev** module. The demo code is shown below:

```
try {
  val intent = this.context?.let { Intent().setClassName(it,
"org.oppia.android.dev.DeveloperOptionsActivity") }
  startActivity(intent)
  Log.d("DeveloperOptionsActivity", "Successfully Opened")
} catch (e: Exception) {
  Log.d("DeveloperOptionsActivity", "Failed to open")
}
```

- This way we can make sure that the *Developer Options* is only accessible when we are using the debug version of the app.

Now, we will look inside the **dev** module and go through the main components one by one.

- **dev/developermenu:** It will contain the activity, fragment, their presenters, view models and adapters for *Developer Options* screen. The layout will be defined in the **res/layout** directory. We will use a ***recyclerView*** similar to the *AdministratorControls* screen to create the layout.

  The important point to note here is that we will implement the logic for *showing all hints and solution* and *force crash app* here only. We will need to define interfaces corresponding to each of the features which we will override in the **DeveloperOptionsActivity**. We will call these interfaces inside the onClick function defined in the **DeveloperOptionsViewModel** corresponding to their respective views.

  - **dev/developermenu/ShowHintsAndSolutionListener:** This is a listener to enable all hints and solution. We will override this listener in the **DeveloperOptionsActivity.** The logic for showing all hints and solution by default is as follows:

- To implement this feature, we will use dagger to create an *injectable boolean* which determines whether the hints and solution are enabled or not. The default value of this boolean will be ***false.***

- We will also need a module (say ***HintsAndSolutionEnabledModule***) which will provide dependencies corresponding to this feature.

  Note:The above approach is inspired by the PR #2986.

- Now, when we will toggle **on** the *Show all hints and solution* option in *Developer Options* we will set this boolean as ***true***.

- Now, we will have to add some conditions in ***StateFragmentPresenter.kt*** and ***QuestionPlayerFragmentPresenter.kt.*** The purpose of this condition is to check whether the *injectable boolean* is *true* or *false* and perform the necessary actions depending on the condition. The demo code is shown below:

```
if (hintsAndSolutionEnabled) {
 if (currentState.interaction.hintCount > 0) {
   viewModel.setHintBulbVisibility(true)
   viewModel.newAvailableHintIndex = 0
 }
}
```

- In all the following points we will assume that the condition specified above is true.

- Now, we will also have to create two functions inside ***HintsViewModel.kt*** to show all the hints and solution. The demo code is shown below:

```
fun processCompleteList(): List<HintsAndSolutionItemViewModel> {
 itemList.clear()
 allHintsExhausted.set(true)
 for (index in hintList.indices) {
   addHintToList(hintList[index])
   if (solution.hasExplanation() && hintList.size * 2 == itemList.size) {
     addSolutionToList(solution)
   }
 }
```

```
 return itemList
}
```

- The above function adds all the hints and solution to the ***itemList.***
- Now, we will also have to check for the same condition as ***StateFragmentPresenter.kt*** in ***HintsAndSolutionDialogFragment.kt*** in the function ***loadHintsAndSolution().***
- Here we will use the function specified above to create the list instead of the default function.
- Apart from the boilerplate code, this is all the code we will need to make this feature work.

Note: The detailed information regarding separate implementation of the hints and solution part for dev and prod mode can be found in the **Appendix** at the end.

- **dev/developermenu/ForceCrashListener:** This is a listener to force crash the app. We will override this listener in the **DeveloperOptionsActivity.** The logic to force crash the app is to simply throw an uncaught exception. The demo code is shown below:

```
throw RuntimeException("This is a crash")
```

- **dev/chapter:** In this module we will put the activity, fragment, their presenters, view models and adapters for the *Mark Chapters Completed* screen. The layout will be defined in the **res/layout** directory. Here also we will use **recyclerView** to display the list of chapters.

  The logic for marking the chapters completed is as follows:

  ➔ We will use the function **recordCompletedChapter()** inside **StoryprogressController.kt** to mark the chapters completed.

  ➔ To get the list of chapters, we will use the **TopicListController.kt** to get the list of all the topics using the function **getTopicList()** and from these topics we will extract all the stories using ***topic.storyList.*** From this list of stories we will extract all the chapters using ***story.chapterList.*** We will then compile all the chapters in a list and use it.

➔ The default interaction and visibility of chapters as described in the *Product Design* section can be achieved with the following pseudo code:

```
for (chapter in chapterList)
{
   if (chapter.chapterPlayState == ChapterPlayState.COMPLETED)
   {
      chapter.checked = true
      chapter.enabled = false
   }
   else if (chapter.chapterPlayState != ChapterPlayState.COMPLETED)
   {
      if (chapter.hasMissingPrerequsiteChapter)
      {
         chapter.enabled = false
      }
      else
      {
         chapter.enabled = true
      }

   }
}
```

➔ The chapters in the list will not be accessible to select unless its prerequisite chapters are selected first. We can achieve this by allowing only **serialised selection** of chapters of a particular story. The pseudo code for the *onClickListener* could be as follows:

```
clickedChapter.checked = !clickedChapter.checked
if (clickedChapter.index != chapterList.size - 1 &&
    clickedChapter.checked = true)
{
    chapterList[clickedChapter.index + 1].enabled = true
}
else if (clickedChapter.index != chapterList.size - 1 &&
    clickedChapter.checked = false)
```

```
{
    for (i = (clickedChapter.index + 1) to chapterList.end)
    {
        chapterList[i].checked = false
        chapterList[i].enabled = false
    }
}
```

➔ After selecting all the desired chapters from the list, the *MARK COMPLETED* button will be clicked. Here, we will store the *topicId, storyId* and *explorationId* corresponding to each of the chapters in the form of a list.

➔ We will then traverse through this list and call the **recordCompletedChapter()** function for each of them.

➔ Thus, all the selected chapters will be marked as completed.

- **dev/story:** In this module we will put the activity, fragment, their presenters, view models and adapters for the *Mark Stories Completed* screen. The layout will be defined in the **res/layout** directory. Here also we will use **recyclerView** to display the list of stories.

  The logic for marking the stories completed is as follows:

  ➔ Unlike in the case of chapters, here all the stories will be accessible for selection.

  ➔ To get the list of stories, we will use the **TopicListController.kt** to get the list of all the topics using the function **getTopicList()** and from these topics we will extract all the stories using ***topic.storyList.*** We will then compile all the stories in a list and use it.

  ➔ After selecting all the desired stories from the list, the *MARK COMPLETED* button will be clicked. Here, we will store the *topicid* and *storyId* corresponding to each of the stories in the form of a list.

  ➔ We will then traverse through this list and for every story in the list, we will traverse through all the chapters of that story and call **recordCompletedChapter()** function for each of them sequentially.

  ➔ Thus, all the selected stories will be marked as completed.

- **dev/topic:** In this module we will put the activity, fragment, their presenters, view models and adapters for the *Mark Topics Completed* screen. The layout will be defined in the **res/layout** directory. Here also we will use **recyclerView** to display the list of topics.

  The logic for marking the stories completed is as follows:

  - ➔ Similar to the case of chapters, here also all the topics will be accessible for selection.

  - ➔ To get the list of topics, we will use the **TopicListController.kt** to get the list of all the topics using the function **getTopicList()** and then compile all the topics in a list and use it.

  - ➔ After selecting all the desired topics from the list, the *MARK COMPLETED* button will be clicked. Here, we will store the *topicid* corresponding to each of the topics in the form of a list.

  - ➔ We will then traverse through this list and for every topic in the list, we will traverse through all the stories of that topic. Further, for each of the stories of a particular topic, we will traverse through all the chapters of that story and call **recordCompletedChapter()** function for each of them sequentially.

  - ➔ Thus, all the selected topics will be marked as completed.

- **dev/logs:** In this module we will put the activity, fragment, their presenters, view models and adapters for the *Event Logs* screen. The layout will be defined in the **res/layout** directory. Here also we will use **recyclerView** to display the list of event logs.

  The important thing here is to get the list of analytic event logs. We will see the logic below:

  - ➔ Here, we do not need to store the logs in the cache as we will be deleting them when the app is closed. Instead, we will be using a singleton class which will be globally accessible.

  - ➔ Using this, we will store the logs temporarily only for the current session of the app and as the app is closed, all the stored data will be cleared automatically.

  - ➔ Inside this class, we will have a *MutableList* of *EventLog.* We will add the logs to this list as they are created in the *AnalyticsController*.

➔ Then, inside the *EventLogsActivity* we will fetch the logs from this list and extract the required information i.e, ***actionName, context, priority*** and ***timestamp.*** We will then pass these data to the adapter which in turn will show it in the UI.

➔ Thus, on following these steps we can easily view the event logs reported in the current session of the app.

- **dev/network:** In this module we will put the activity, fragment, their presenters, view models and adapters for the *Force network type* screen. The layout will be defined in the **res/layout** directory. Here also we will use **recyclerView** to display the list of network states.

  Now, we will see how the logic works here:

  ➔ We will use **NetworkConnectionUtil** to get and set the network type.

  ➔ First, we will get the current state of the device by setting **testConnectionStatus** to ***null*** and then calling **getCurrentConnectionStatus().**

  ➔ When the network state is selected as ***Default***, we will set the connection status to the actual state of the device by passing the value obtained above in the **setCurrentConnectionStatus()** function.

  ➔ For options other than *Default,* we will pass corresponding values to the **setCurrentConnectionStatus()** function.

  - **Wifi:**
    setCurrentConnectionStatus(ConnectionStatus.LOCAL)
  - **Cellular:**
    setCurrentConnectionStatus(ConnectionStatus.CELLULAR)
  - **None:**
    setCurrentConnectionStatus(ConnectionStatus.NIONE)

  ➔ There are a few conditions here on how to forcefully set the connection status of the app. We will see them below:

  - If the actual connection status is ***LOCAL*** or ***CELLULAR:*** In this case, all the options will work as expected.

- If the actual connection status is ***NONE:*** In this case, the *Wifi* and *Cellular* options won't work as it is an impossible case. We will show a *Toast* to the user with the message that *"This action belongs to an impossible case".*

Note: The detailed information regarding separate implementation of the *NetworkConnectionUtil* for dev and prod mode can be found in the **Appendix** at the end.

We have seen the implementation approach of the project. Now, we will look into the *Testing Approac*h.

Third-party Libraries*

This project does not add any new third-party libraries.

Testing Approach

Testing is one of the most important parts of any project. It allows the developer to verify that the behaviour of the application is the same as the developer wants it to be, before the feature is released.

In this project we will test the following things by writing tests in Espresso and Robolectric:

- **Test for checking Modularization:**
  - When the app is built in debug mode, check if the *developer menu* is accessible or not. [Logic]
- **Test for checking correctness of Hints and Solution:**
  - Enabling hints and solution and checking if the *Hint Bulb* is visible by default or not. [UI + Logic]
  - Enabling hints and solution and checking if all the hints and solution are visible by default or not. [UI + Logic]
- **Test for checking correctness of Developer Options menu:**
  - Check if all the options in the recycler view are shown correctly or not. [UI]
  - Check if onClicks of different options are working correctly or not. [UI]
  - Check if *Show hints and solution* toggle is working as expected or not. [Logic]
  - Check if the *Force crash app* option is working as expected or not. [Logic]

- **Test for checking if Mark Chapters Completed feature is working correctly:**

    - Check if all the chapters are displayed correctly or not. [UI]
    - Check if user interaction with the list is working correctly or not. [UI]
    - Check if the chapter selection logic as described in the *Product Design* section is working as expected or not. [Logic]
    - Marking a chapter completed and checking if the chapter is actually marked completed or not. [Logic]

- **Test for checking if Mark Stories Completed feature is working correctly:**

    - Check if all the stories are displayed correctly or not. [UI]
    - Check if user interaction with the list is working correctly or not. [UI]
    - Marking a story completed and checking if the story is actually marked completed or not. [Logic]

- **Test for checking if Mark Topics Completed feature is working correctly:**

    - Check if all the topics are displayed correctly or not. [UI]
    - Check if user interaction with the list is working correctly or not. [UI]
    - Marking a topic completed and checking if the topic is actually marked completed or not. [Logic]

- **Test for checking if Event Logs feature is working or not:**

    - Check if all the event logs are displayed correctly or not. [UI]
    - Storing the logs in singleton and checking if the logs are actually there or not. [Logic]

- **Test for checking if Force Network Type feature is working or not:**

    - Check if all the network options are displayed correctly or not. [UI]
    - Check if onClicks of all the options are working as expected or not. [UI]
    - Changing the network state to a particular state and checking if the network state is actually changed or not. [Logic]

These are the possible tests that we need to implement to check the correct working of this project.

Milestones

Milestone 1

Key Objective:
The *dev* module contains UI for *Developer Options*, *Mark Chapters Completed*, *Mark Stories Completed* and *Mark Topics Completed*. The *Developer Options* and its features are only accessible in developer builds of the app. The app crashes on clicking *Force Crash App* option in *Developer Options* menu. The options to mark chapters, stories and topics completed are working. All these features are backed by *Espresso* and *Robolectric* tests.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Create dev module and introduce initial UI for Developer Options		08/06/2021	11/06/2021
1.2	Introduce UI for Mark Chapters Completed	1.1	15/06/2021	19/06/2021
1.3	Implement logic for marking chapters completed	1.1, 1.2	21/06/2021	25/06/2021
1.4	Introduce UI for Mark Stories Completed and Mark Topics Completed	1.1	29/06/2021	03/07/2021
1.5	Implement logic for marking stories and topics completed	1.1, 1.4	05/07/2021	09/07/2021

Milestone 2

Key Objective:
The *dev* module contains UI for *Event Logs* and *Force Network Type* screens. The user is able to view analytic log events through the *view Event Logs* option in the *Developer Options* menu. All hints and solution can be enabled by default using *Show all hints and solution* toggle option in the *Developer Options* menu. The user can force the *ConnectionStatus* of the app using the *Force Network Type* option in the *Developer Options* menu.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Introduce UI for Event Logs and Force network Type screen	1.1	19/07/2021	23/07/2021
2.2	Implement logic to force network type of app	1.1, 2.1	25/07/2021	28/07/2021
2.3	Implement logic to view analytic events logs	1.1, 2.1	01/08/2021	05/08/2021
2.4	Implement logic to show all hints and solution	1.1	04/08/2021	10/08/2021

Optional Sections

Additional Project-Specific Considerations

Privacy

No, this project does not collect any new user data or change how user data is collected.

Security

No, this feature does not provide any new opportunities for users to gain unauthorized access to user data or otherwise impact other users' experience on the site in a negative way. This feature is only meant for developers so users won't get affected by it.

Accessibility (if user-facing)

This project will introduce some UI components to the app. There are not any complex new screens so it would not offer major accessibility challenges. Also, all the activities introduced will have a label assigned to them.

Documentation Changes*

This project changes the app architecture quite a bit so we will need to add the necessary information in the wiki to help others understand how the new app architecture works.

Ethics*

Having *developer options* in an application is very useful and is practised by many major applications. Oppia-android should definitely have something like this which would reduce the time and effort of developers significantly thus increasing the efficiency of the development process.

Future Work

In this project, we try to separate the code using modularisation. However, to achieve complete separation we should introduce product flavors to the app. This way we can completely separate the debug code from the production code.

Appendix

- **Separate implementation for a dev mode and prod mode for both NetworkConnectionUtil and the "hints and solutions" parts of the project**

  - Here we need to have different versions of the same files for dev mode and prod mode. What we can do is we create a directory (say **dev**) in the **app** module (for having implementations of the *hints and solution* feature) and **util** module (for having different implementations of the *NetworkConnectionUtil)*.

  - We will then follow the same structure as in the ***main*** directory of the respective modules. Below is a demonstration for the same:

```
app
 ├ dev
    ├ java
       ├ org.oppia.android.app
          ├ hintsandsolution
             ├ HintsViewModel.kt
             ├ HintsAndSolutionDialogFragmentPresenter.kt
          ├ player
             ├ state
                ├ StateFragmentPresenter.kt
                ├ QuestionPlayerFragmentPresenter.kt
```

```
util
 ├ dev
    ├ java
       ├ org.oppia.android.util
          ├ networking
             ├ NetworkConnectionUtil.kt
```

- Please note that these files already exist in the ***main*** directory of their respective modules. We have created a different version of it in a separate ***dev*** directory which is shown above.
- Now we have to decide which file to be used and when. For this we will use the **select** statement in the **BUILD.bazel** file to determine dependencies at build time.
- I have already mentioned in the proposal how we are going to use the **select** statement to add dependencies in **BUILD.bazel** by assigning flags in the build command. We can similarly use the **select** statement here to determine which version of the file we will use in a particular build of the app.
- Below is a sample implementation for the same:

```
select({
   "dev":
["src/dev/java/org/oppia/android/app/hintsandsolution/HintsViewModel.kt"],
   "//conditions:default":
["src/main/java/org/oppia/android/app/hintsandsolution/HintsViewModel.kt"]
,
})
```

- Now as for the checker to show/hide hints and solutions, we will create the checker inside the ***dev*** directory alongside *HintsViewModel* and other files. This way we can isolate the checker to only dev mode of the app.
- Thus, by following these steps we can have separate implementations for dev mode and prod mode for both *NetworkConnectionUtil* and *"hints and solution"* parts of the project.

- **Other ways to achieve modularization**

1. ***Using product flavors:*** (*Reference*)
   At first I thought of using product flavors to create different build variants of the app. This way we could have easily created different versions of the ***Navigation Drawer*** *for debug and production versions of the app. It will*

*allow us to completely separate the code, even the code for including the **Developer Options** button in the **Navigation Drawer.***

***Problem faced:** This method works only when the app is using **Gradle** build. Even after a lot of research I could not find a way to achieve the same in the Bazel version of the app. Since the project requirement is to implement this project only on the **Bazel** version of the app as we are migrating from **Gradle** to **Bazel**, I had to reject this approach. Also, introducing product flavors was beyond the scope of this project.*

2. ***Using dynamic feature modules:*** *(Reference)*
   *Since introducing product flavors was not part of this project, I looked for other approaches and found that using **dynamic feature modules** can do the job. In this approach, we would put all the code related to this project in a module and then will dynamically include/exclude it from our app. This would allow us to manage the app size as well by deleting the module from the app if not needed.*

   ***Problem faced:** One problem is that to include this module dynamically the app must be connected to the internet, which is not a good thing. Also, This method works only when the app is using **Gradle** build. Even after a lot of research I could not find a way to achieve the same in the Bazel version of the app. Since the project requirement is to implement this project only on the **Bazel** version of the app as we are migrating from **Gradle** to **Bazel**, I had to reject this approach.*