GSoC 2020 Proposal

Project : Analytics Support

Organisation : Oppia Foundation

About You

Hi Reviewers !

My name is Sarthak Agarwal and I am in my freshman year at Galgotias University, Greater Noida. I am currently pursuing a Bachelors in Technology degree in Computer Science with specialisation in Artificial Intelligence and Machine Learning. I would like to put forward a proposal for implementing Analytics Support in Oppia's android application. Hope you guys like it.

Prior experience

More than just being a developer, I am also an enthusiast. I love coding and implementing things. When I was in school, I used to see my elder sister code and observe the way things were brought to life through coding. Coding always intrigued me from the beginning and maybe that's why I made my first mobile application when I was in 9th grade.

It was called Mission Talaash. It was basically an application to reunite the missing people with their families. It got selected by Design for Change(DFC) among the Top 100 teams across India for bringing a change in society. We secured 2nd place in the Largest impact category at the DFC awards held in Ahmedabad, India.

I have an experience of about 7-8 months in Android development. I have used both Java and Kotlin to build my applications. I like making a positive change in society and Open Source is the thing that helped me in doing so. I didn't know much about Version Control Systems or Git or Github earlier but participating in NJack Winter of Code (NWoC) organised by IIT Patna, India gave me a lot of hands-on experience of open source. I worked on 4 android projects there and implemented numerous features. I created around 51 issues and put across 41 Pull Requests out of which 34 were merged and unfortunately 7 were closed. I was the 2nd highest contributor on NWoC's Leaderboard when it ended.

After NWoC, I interned for Suno Kitaab Pvt. Ltd. as a Product Development Intern where I was responsible for end-to-end development of two of their Android applications and the Firebase Console for data storage and retrieval. The two applications were -

1. Sunokitaab Learning App (https://play.google.com/store/apps/details?id=com.sunokitaab.sunokitaab)
2. Sunokitaab Studio (Private app)

The SunoKitaab Learning App is an audio streaming application which fetches their audio based lectures through a RSS feed and streams them using ExoPlayer and Android's MediaPlayer. The app can also download the files and play from internal storage. The audio streams playback can be handled from the notifications tab. I also implemented Firebase Analytics in this application to track which content is getting viewed the most.

SunoKitaab Studio is their private application for recording audio lectures. Using this application the recorder records the audio and provides suitable details like the grade, the book and the topic, on which the recording is based on. Then this information is put up in their firebase storage as a file from where their editors pick it up and put them on the website after editing.

My Oppia Contributions -

Merged PRs -

- Removal of UserAppHistory controller- https://github.com/oppia/oppia-android/pull/685
- Landscape onBoarding workflow - https://github.com/oppia/oppia-android/pull/698
- Landscape Topic practice - https://github.com/oppia/oppia-android/pull/719
- Removal of Thumbnail Code - https://github.com/oppia/oppia-android/pull/982
- RecyclerView via BindableAdapter - https://github.com/oppia/oppia-android/pull/955

Closed PRs -

- ProfileChooser Different Screen Scenarios - https://github.com/oppia/oppia-android/pull/732
- Landscape Profile List - https://github.com/oppia/oppia-android/pull/831
- HighFi StoryTextSizeActivity - https://github.com/oppia/oppia-android/pull/845

PR in Pipeline -

- Exploration back flow - https://github.com/oppia/oppia-android/pull/1036

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

There are multiple reasons because of which I am interested in working with Oppia. First, it's an educational platform and nothing gives more happiness to a developer than knowing that his code might help someone in his/her educational life, that someone might actually find it easy to study because of his code. Secondly, it is an open sourced application and I love contributing to

open sourced projects because they have a higher potential of success and a wider scope of development. Different people having different opinions come together in open source which results in the best possible outcome. Thirdly, I liked how the mentors responded to the contributors at Oppia. They were very gentle, respectful and informative.

I chose this project mainly because it has Firebase Implementation in it and I like implementing Firebase and its awesome tools in applications. I also chose this project because it will help a great deal in the future of this application. It will help gain insights and thereby help the end-user get quality content for learning.

Contact info and timezone(s)

Name - Sarthak Agarwal
Country - India
University - Galgotias University, Greater Noida
Email - agarwal.sarthak262012@gmail.com
Github - https://github.com/Sarthak2601
Timezone - Indian Standard Time (IST) / +5:30 GMT
Preferred method of communication - Hangouts, Mail and Gitter

Time commitment

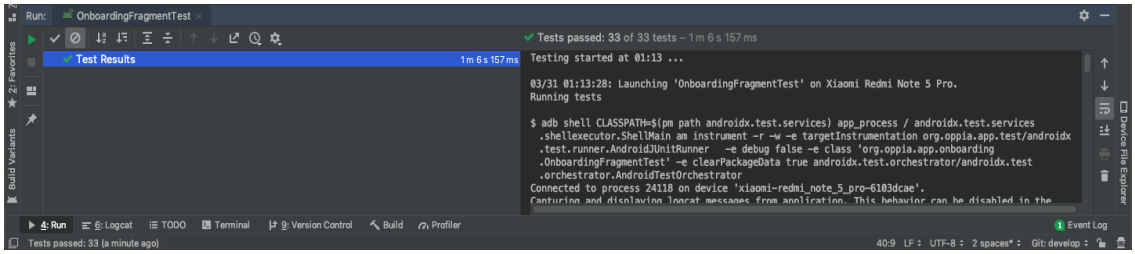
My time commitment to this project will be as follows -

- 7 hours a day (Monday - Saturday)
- Time dedication on Sunday will be subject to work requirements.
- Total : 40-50 hours a week.

The main advantage of being an enthusiast is that you never get bored with it. I never get tired of coding. So if there is work, you won't find me not doing it. My time dedication to a project is particularly based on what the project demands. So bringing in long hours to get the job done won't be an issue with me.

Essential Prerequisites

- Yes, I am able to run tests on my device.



Other summer obligations

I strongly feel that full commitment is necessary for any project to be completed within a fixed timeline. Apart from GSoC I don't have any other obligations during the summer. I would be fully committed to building this project and leading it to completion.

Communication channels

I am comfortable in communicating through any communication channel that my mentor chooses. Currently I use Gitter, Hangouts and Mail for communication. I have also used Slack in the past. The mentor can expect a reply from me in less than an hour from his/her message.

Project Details

Product - Analytics Support

Product Design

The project 'Analytics Support' aims to provide Team Oppia deep insights about their product. Through this project, it aims to make the lives of its developers and strategy makers a lot easier. In my opinion, adding Analytics Support to Oppia Android this summer is a really constructive and well planned idea as it will strengthen the application and help in enhancing its features, whose implementation will be backed by solid facts and customer response gathered via analytics.

The product will have a large variety of users. So, I have categorised them into 3 parts -

1. Product Development Team
2. Product Analysis Team
3. End-User

The integration of Analytics support will give them deep insights about the application and its usage across all its users.

Product Development Team -

This team mainly consists of the code developers, code testors and the content creators.

For code developers and testers, with the crashing reports and app health reports they can appropriately manage, develop, optimize and enhance the code of the application so that it remains stable across all devices, giving all its users a smooth and bug-free experience.

For the content creators, they will get to know about the type of content which is getting viewed the most. This will help them in future production of content and thereby increase the application's content quality and assure that the end-user is getting the content he/she wants.

Product Analysis Team -

Here, I am talking about the team that analyses and manages the product performance of Oppia Android. In my terms of saying, the major tasks of this team are -

1. Performing app's feasibility analysis
2. Keeping an eye on app's scope
3. Enhancing user experience via quality content
4. Researching for newer app features and upgrades

With the integration of Analytics Support, the Product Analysis teams will get to know about the segments of the app that attract most of its users and accordingly set the path of development for the app. The team will also get to know the geographical outreach of their product via Analytics. They can use this data to determine the scope of the application. The addition of Analytics Support will also help in the app's Feasibility Analysis and in return lead to better strategies, plans and designs that will help in sustaining the application for a longer period of time.

End-User

The end-user becomes one of the biggest beneficiaries in this scenario. The analytics support will help in better planning and execution of the work of Product Development and Product Analysis teams, which in return will give out results in the form of better quality content, which is the biggest benefit for the end-user.

Project Implementations

This project consists of various Firebase implementations related to Analytics. Firebase features like -

1. Firebase Analytics
2. Firebase Crashlytics
3. Firebase Performance Monitoring

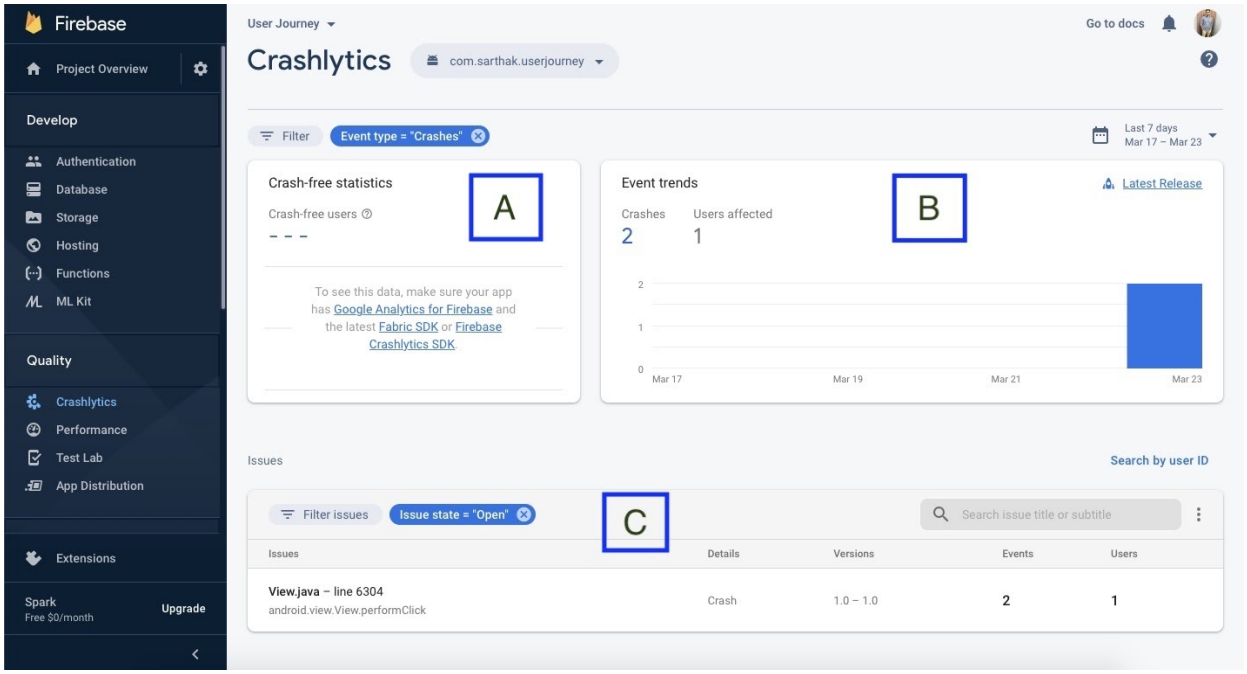
will be thoroughly used throughout the project for features like app-system health tracking, crash reporting and in-app impression tracking.

Firebase Crashlytics Implementation

Firebase Crashlytics is a realtime crash reporter that helps you track, prioritize, and fix stability issues that raise questions on your app quality. Crashlytics saves you troubleshooting time by intelligently grouping crashes and highlighting the circumstances that lead up to them.

For instance, the image on the right is of an application that will crash on button click. The developer of this app would never get to know about the crash if it doesn't happen on his/her testing device or emulator. Therefore having a crash reporter that can keep track of these crashes across all the application users becomes a necessity.

For this crash reporting, Firebase provides its Crashlytics services, which keep track of the crashes.



Crashlytics Dashboard

Once a crash happens for an application, it is duly reported in the Crashlytics Dashboard. The above image depicts the Dashboard showcasing the crash that happened in the gif. I have divided the Dashboard into 3 parts - A, B and C for explaining them efficiently.

A - Crash Free Statistics

As the name suggests, this section provides the user with statistics about the number of end-users having smooth, crash free experience. In the current case, there is no crash free user of the demo app.

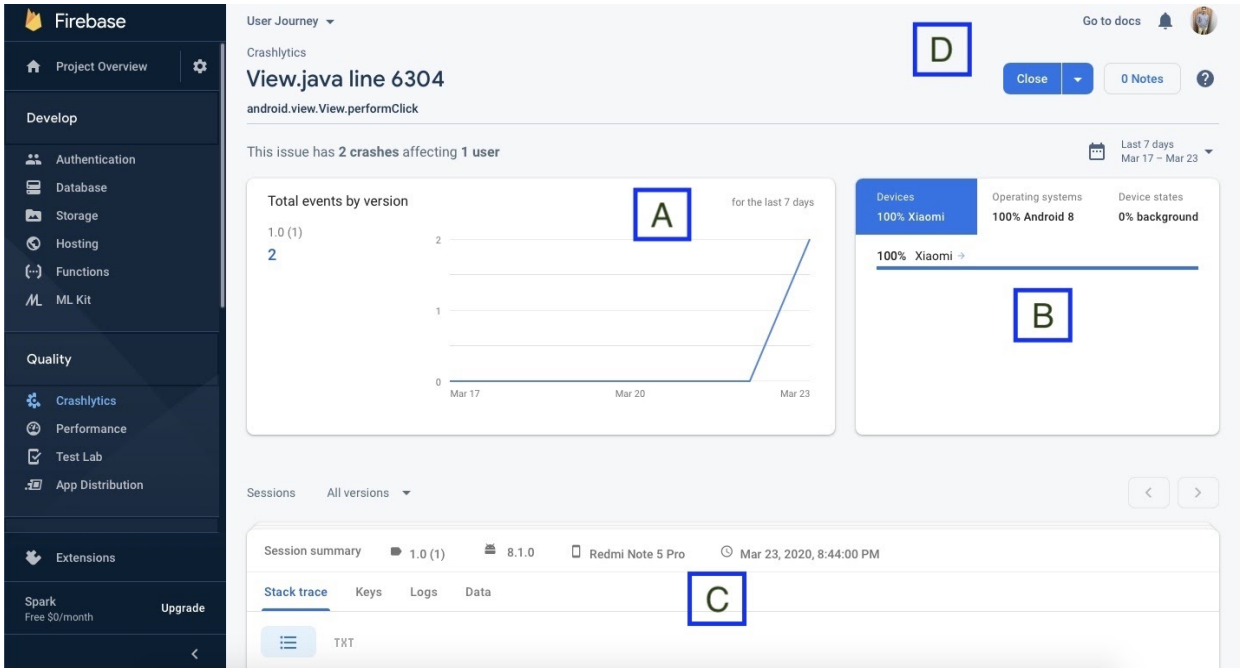
B - Event Trends

This section lists the total number of crashes that have occurred across all the end-users and how many users are affected by those crashes. All this information is also represented in the form of a bar chart.

C - Issues

Whenever a crash occurs, there is a possibility that the crash occurs across all the end-users due to the same problem. Crashlytics clubs those crashes and forms an issue which can be managed by the developers. This section lists out those issues and gives details about how many times that issue has caused the app to crash and how many users were affected by it. In the current case, we have one issue that has occurred twice for the same user.

Whenever we click on the Issues section, we reach the following segment -



Issue Segment

The issue segment gives a deeper insight of the issue that is contributing to the erosion of our app quality. Again, for providing an efficient description I have divided this segment into 4 parts.

A - Graph

Here, the total number of crashes related to that issue are put against their crashing dates in the form of a line chart. These events can be filtered according to the various app versions. Currently in the demo app, there is only one version and the line chart depicts 2 crashes occurring on March 23.

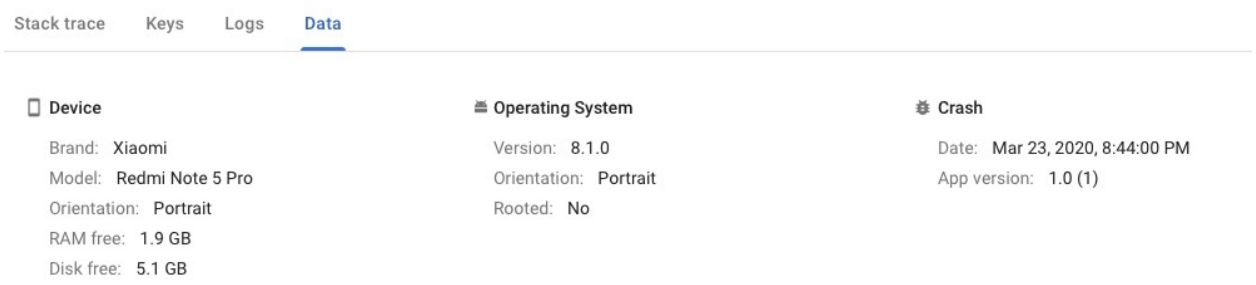
B - Device Details

This section gives cumulative details of the devices on which the app crash has occurred. As you can see in the picture above, device company, operating system and device states are the basis for information that gets displayed. According to the shown data the demo app faces crashes in Xiaomi mobiles having Android 8.0 as its operating system and the app doesn't go in the background.

C - Session Summary

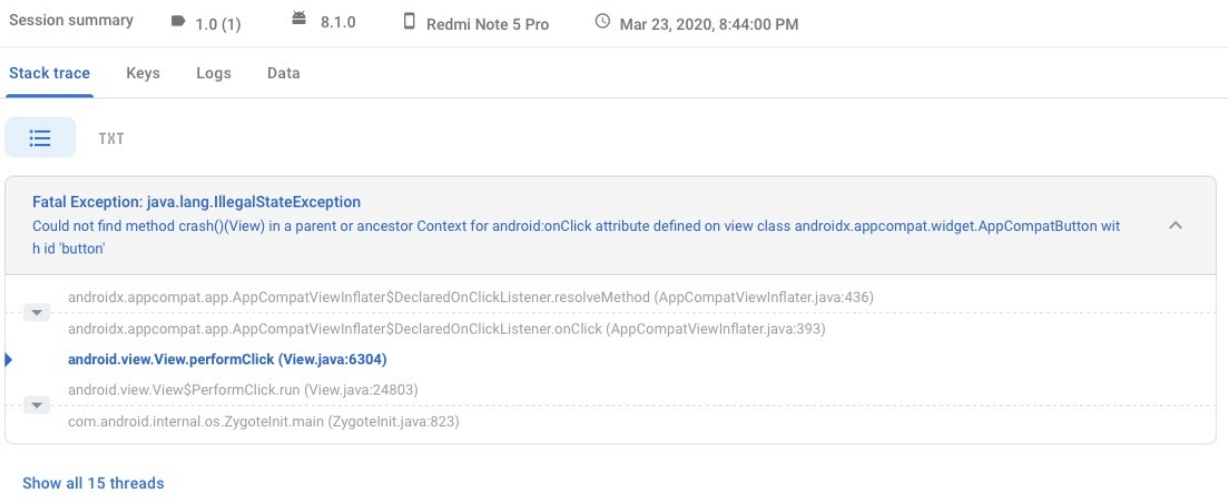
It gives us the actual crash reports. It is one of the most important sections of the whole Crashlytics panel. It consists of 4 sub-parts -

1. Stack Trace -- which gives the actual stack trace of the crash along with the events that happened before the crash. The user can choose between formatted or plain text description of the stack trace.
2. Keys - they are custom made keys that help the user get the specific state of the app leading up to a crash.
3. Logs - they are custom made data that is sent intentionally by the developer to the crash reporter for quick identification and resolution of the problem that may occur. These logs can also be used in identifying the end-user for which the crash occurs, as we can send end-user identifiers using them.
4. Data - which provides the user with all the device information he/she needs for reproducing and resolving the problem.



Data sub-section of Session summary in Issues segment

The above image shows the device details of the device on which the demo app crashed twice. It recorded the device model, free disk space, free ram, OS, orientation and crash date of the device.



Stack trace sub-section of session summary in Issues segment

The above image shows that the app crashed on the press of a button. It also consists of the app functionalities that led up to the crash.

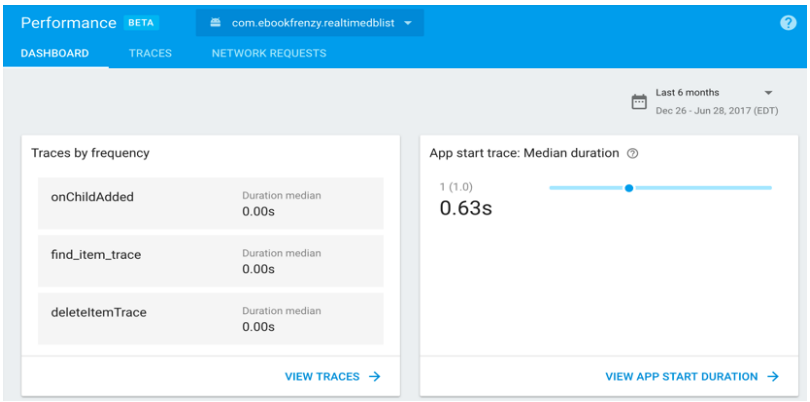
D - Issue management

This section consists of 2 options - close the issue or add notes for the current issue. The users(developers) can add notes about what they think of the issue and its solution. The close button should be clicked whenever the issue has been resolved.

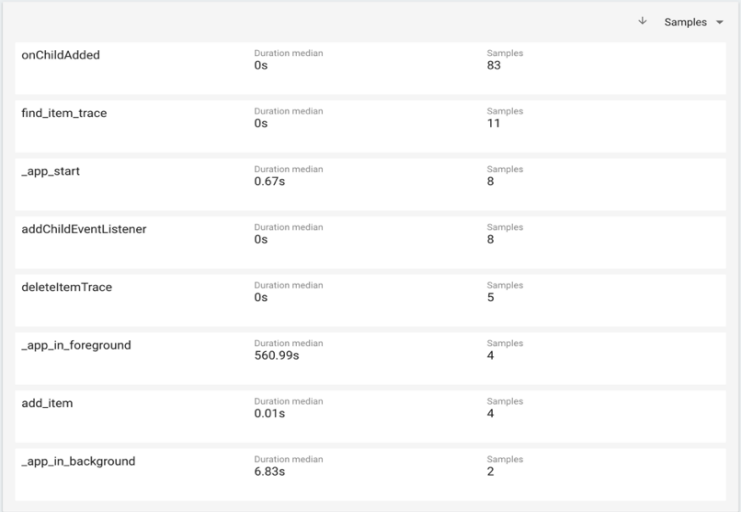
Firebase Performance Monitoring Implementation

Firebase Performance Monitoring is a service that helps us gain insight into the performance characteristics of our applications. The Performance Monitoring SDK collects performance data from our app, reviews it and analyzes that data in the Firebase console. Performance Monitoring helps us to understand where and when the performance of our app can be improved so that we can use that information to fix performance issues.

The image on the right shows the Performance Monitoring Dashboard. It gives the summary of the data collected from the application.

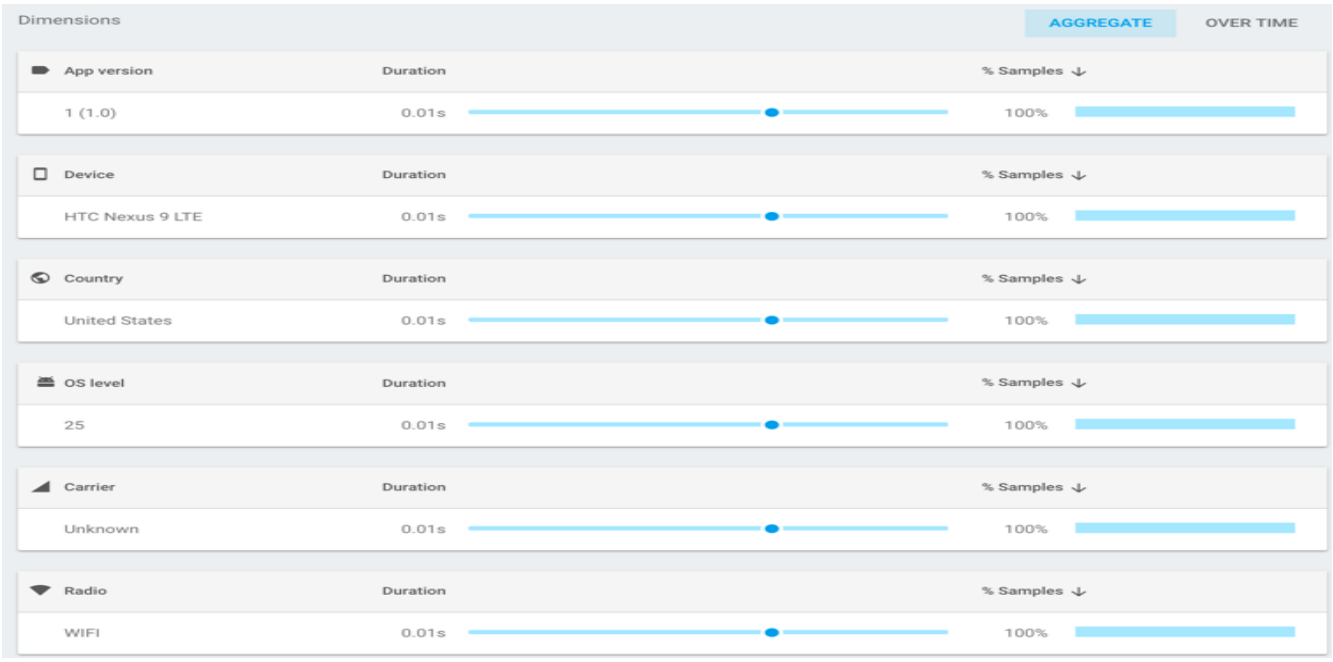


If we see closely, a summary of traces is placed on the left, while a summary of the App start trace is put on the right. The dashboard also provides options for navigating to the Traces and Network Requests sections of the console.



The image on the left displays detailed information about the traces set up by the developer to track app performance. This section provides details like the average time it takes for the app to start. It also shows the number of samples, based on which the data is prepared for a specific trace.

On most occasions, the user needs to know deeper insight than just knowing the duration and number of samples. To view additional performance details for a trace, select it from the list to display the trace detail screen.



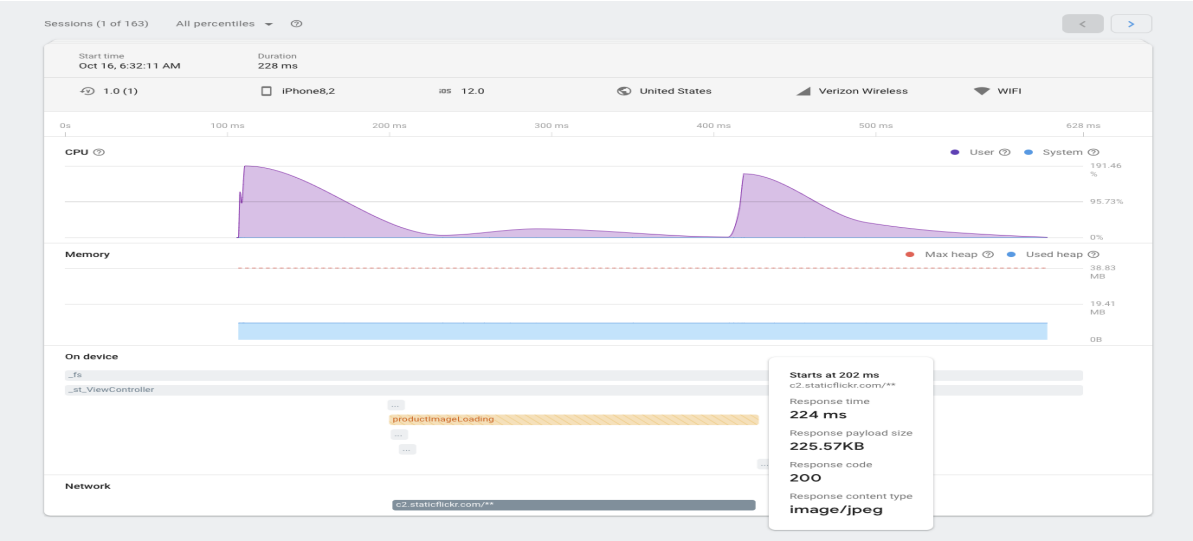
Detailed Trace section

For example, the image above shows data for the add_item trace including the trace counter. Clicking on the View More link will display information based on the device type, app version, operating system, country and network characteristics. Performance Monitoring *automatically* provides the following types of duration traces for iOS and Android apps:

- *App start* trace — Measures the time between when the user opens the app and when the app is responsive
- *App in background* trace — Measures the time when the app is running in the background
- *App in foreground* trace — Measures the time when the app is running in the foreground and available to the user
- *Screen* trace — Spans the lifetime of a screen and measures slow and frozen frames

Performance metrics that will be implemented are -

- ➢ **CPU:** How much user time and system time your app consumes.
- ➢ **Memory:** How much heap memory your app uses. Heap memory is the memory used for dynamic allocations, including objects created, objects deallocated, and objects that the app is actively using.
- ➢ **Individual information:** Detailed information about a single instance of a trace or network request, including start time, end time, duration, request size, and response size.
- ➢ **Concurrent instances:** Information about traces or network requests that happened at the same time.
- ➢ **Device attributes:** Information about the device, including app version, model, OS version, radio, and custom attributes.

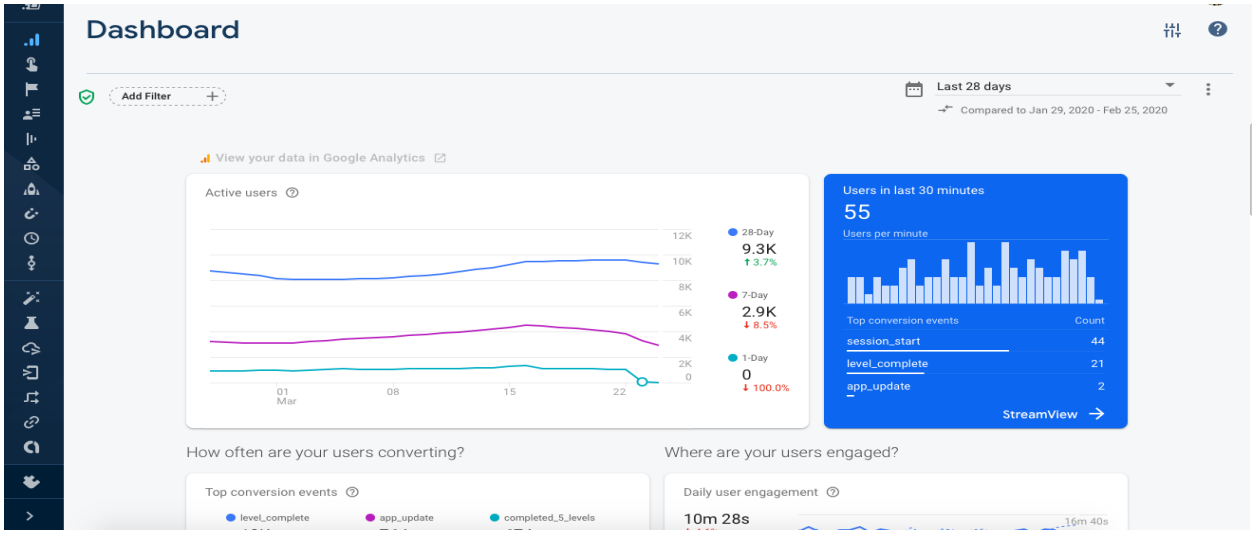


Firebase Performance Metrics

Firebase Analytics Implementation

At the heart of Firebase is Google Analytics, a free and unlimited analytics solution. Analytics integrates across Firebase features and provides unlimited reporting for up to 500 distinct events that can be defined using the Firebase SDK. We get a better understanding of our user behaviour via Analytics reports. This in turn enables us to make informed decisions regarding app marketing and performance optimizations.

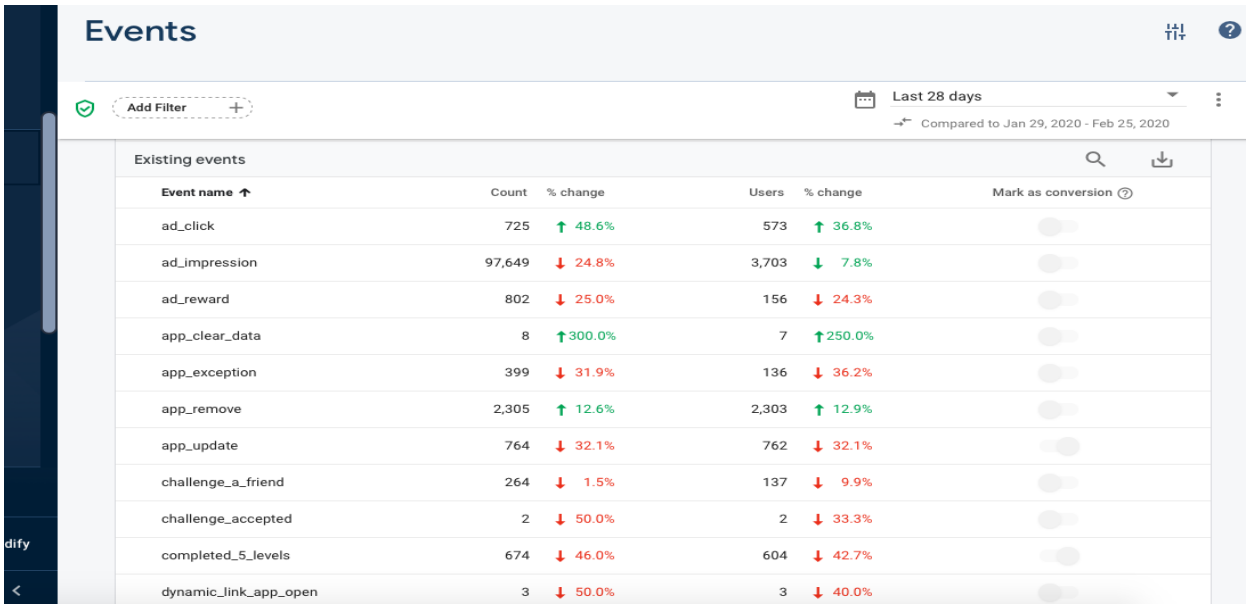
The SDK captures a number of events and user properties automatically. It also allows us to define our own custom events to measure the things that uniquely matter to our business. Once the data is captured, it's available in a dashboard through the Firebase console.



Firebase Analytics Dashboard

This dashboard provides detailed insights about our data — from summary data such as active users and demographics, to more detailed data such as identifying our most purchased items. Currently for this project we can use the existing dashboard. However, we can move on to BigQuery and then further make our own customised Dashboards using Data Studio. We can further use this for our decisions.

Now in this project, we are mainly going to focus on the Events section of Analytics. Whatever happens in our application, from a user action to a system event or maybe even an error, everything is an Event.



Event name	Count	% change	Users	% change	Mark as conversion
ad_click	725	↑ 48.6%	573	↑ 36.8%	
ad_impression	97,649	↓ 24.8%	3,703	↓ 7.8%	
ad_reward	802	↓ 25.0%	156	↓ 24.3%	
app_clear_data	8	↑ 300.0%	7	↑ 250.0%	
app_exception	399	↓ 31.9%	136	↓ 36.2%	
app_remove	2,305	↑ 12.6%	2,303	↑ 12.9%	
app_update	764	↓ 32.1%	762	↓ 32.1%	
challenge_a_friend	264	↓ 1.5%	137	↓ 9.9%	
challenge_accepted	2	↓ 50.0%	2	↓ 33.3%	
completed_5_levels	674	↓ 46.0%	604	↓ 42.7%	
dynamic_link_app_open	3	↓ 50.0%	3	↓ 40.0%	

Events Section of Analytics

The above image is a demonstration of the collected data of events recorded for a demo application. Through this section, we get to know -

1. Count - the number of times that event has occurred
2. %change (count-wise) - the percentage change in the occurrence of that event
3. Users - the number of users for whom that event has taken place
4. %change (user-wise) - the increase or decrease in the number of users going through that event.

The recorded events are majorly of two types. On one hand we have the events that are automatically collected by Analytics. While on the other hand we have custom events which are made by the developer himself according to the demand.

Automatically collected events include -

1. Screen Tracking
    - Analytics automatically tracks some information about screens in your application, such as the class name of the Activity that is currently in focus. When a screen transition occurs, Analytics logs a `screen_view` event that identifies the new screen. Events that occur on these screens are automatically tagged with the parameter `firebase_screen_class`

2. Event Tracking
    - Analytics automatically logs some events for us and we don't need to add any code to receive them. The following parameters are collected by default with every event, including the custom events we implement ourselves -
        a. Language

    b. Page_location
    c. Page_referrer
    d. Page_title
    e. Screen_resolution
- Few automatically recorded events include -
    a. add_click - when a user clicks on an add
    b. Add_exposure - when atleast one add is on the screen
    c. Add_impression - when a user sees an add impression

The events that I plan on implementing in this project are -

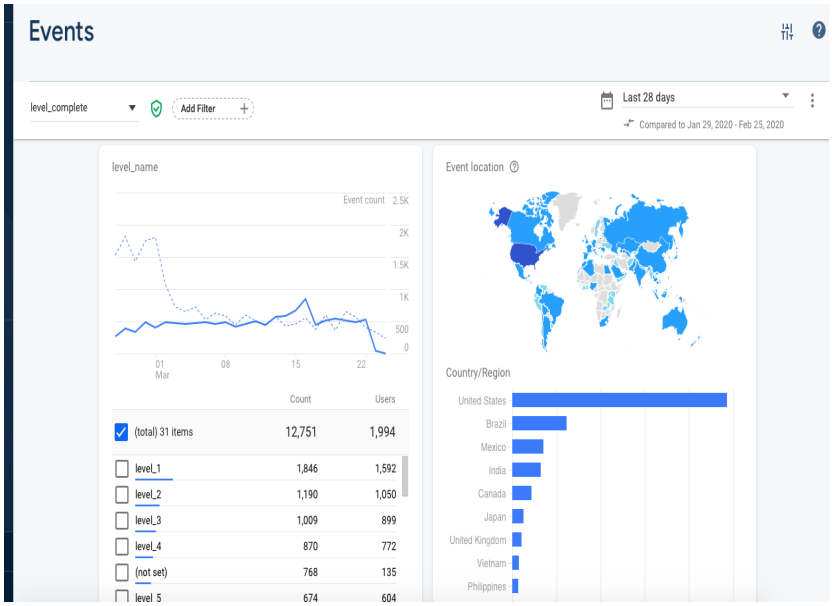
1. Custom screen tracking -
    - Although Analytics provides its screen tracking which tracks every screen but due to a lot of ambiguity in the screen names and views, the user(developer) can't analyse properly.
    - So to make the analysis easier and accurate, we can implement custom screen tracking wherein we can give custom names to the screens.
    - Using this i think we can track the views on the different screens of our application and , some of which are -
        a. Home screen
        b. Topic screen (Topic wise analysis)
        c. Downloads screen
        d. Topic page (where lessons and questions load)

2. User impression tracking -
    - This will be done by using custom events implementation.
    - Following is a list of a few of many user impressions that i wish to track -
        a. Opening of dialog boxes
            - When the user presses back from topic page
            - When the user tries to switch profile
            - When the user tries to delete a profile
            - When the user tries to log out

        b. Clicking of a button - almost all the impressions can be tracked using it
            - When the user opens a topic
            - When the user selects a particular button from the nav bar (like Options, Downloads, Help)
            - When the user enters Admin controls panel
            - When the user enters a certain menu (Options/Downloads/Help/Admin controls) and uses a subsection
    - More user impression tracking events can be made as the application progresses and integrates newer features and functionalities.

We can also view data according to our preferences by adding filters from the top left corner and by setting the range-wise date of the data displayed from the top right.

We can get additional event information by selecting it from the list. Upon selection we are directed towards its subsection which will be entirely dedicated to that event.



For example, if we select the 'completed_5_levels' from the above list, we will be directed towards the subsection on the right.
The subsection contains a lot of data related to that event like -

- Recent occurrences
- Event count
- Count per user
- Event location
- Event demographics
- Events per session

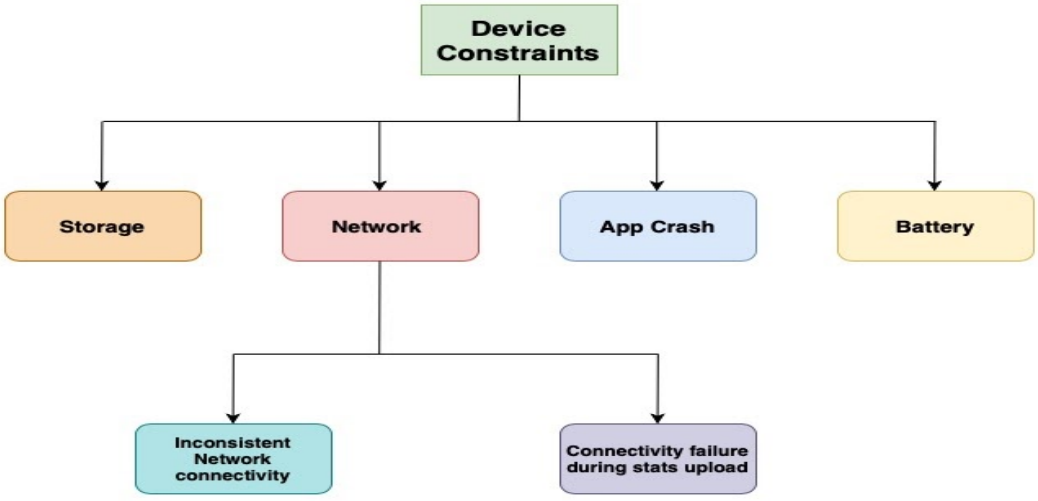
All this data can also be downloaded as a CSV file from the console itself. The CSV file can be further linked to other analytical platforms like Google's BigQuery for getting more out of the recorded data. This data can be analysed and important informed decisions can be made according to it. By tracking events, the user(developer) can determine the areas that the end-users like and further build on that.

Necessity of Offline Support

The necessity of having an Offline Support for Oppia-android is majorly because of two factors which are very much inter-related. The two factors are -

1. To handle the various device constraints efficiently.
2. To prevent loss of highly valuable analytics data collected via user impressions.

Let's first talk about the various device constraints that need to be taken care of. For the purpose of better understanding, I am dividing these into 4 categories.



Storage :

- The storage consumption is one of the major concerns for any application. More storage consumption by a particular app doesn't go well with the end-users.
- So an application must focus on consuming lesser storage space. Keeping the focus on Oppia-Android, the implementation of the Firebase Performance SDK will help in keeping the storage consumption in check.

Network :

- Network connectivity is an issue that is directly linked to the upload of user analytics' stats.
- Inconsistent network connectivity and connectivity failure during stats upload cause trouble when we try to upload data online to the console.
- So for the sake of analytics' accuracy, the network connectivity issue should be handled with proper planning.
- The implementation of Offline Support will mainly help in overcoming the network constraints and ensuring that each and every user statistic is uploaded when connectivity is regained.

App Crash :

- Sometimes, there is very less time available for the uploading of statistics to the console. This is mainly because there are times when the application is killed by the system.
- So, to prevent app-crashes the application's codebase must be properly optimised.
- The implementation of Firebase Crashlytics will help a long way in eradicating this problem.

Battery :

- The battery consumption by an application is also a big factor that decides the user footfall of an application. An app must consume a standard amount of battery only and shouldn't go overboard.
- Analysing and working upon the battery consumption of our application can be done using the data obtained via Firebase Performance SDK.

Inter-Linking of Factors :

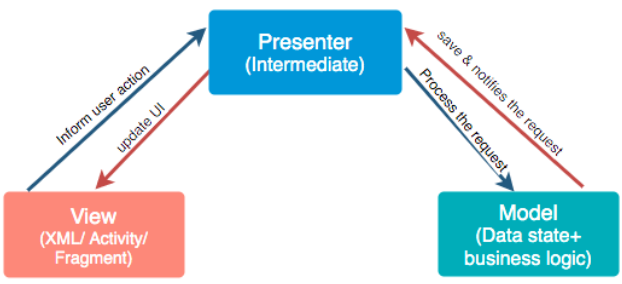
- The connection between device constraints and the availability of the user statistics is deep rooted but simple.
- If we don't handle the device constraints efficiently, we might end up losing the important stats.
    1. Storage might fill up leaving no space for new stats to store
    2. Network connectivity can go while uploading the stats or while using the application.
    3. The app may crash while uploading stats.
- On the other hand, if we handle the constraints well enough, we will have a lot of analytics' data, which will help in the future development of the application.

Technical Design

Architectural Overview

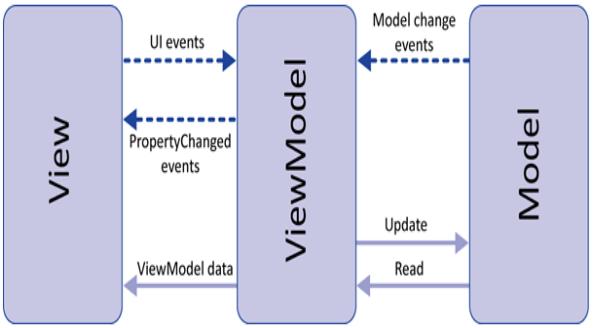
Application Architecture

The Oppia application's architecture is a mix of two well known application architectures, MVP (Model-View-Presenter) and MVVM (Model-View-ViewModel). It takes advantage of their combination to fulfill the requirements of each piece.



The image on the left is a flowchart of **MVP app architecture**. The Model consists of the Business Logic and Data. The View contains the UI elements and the Presenter layer presents Data from Model and controls the Display. The flow of MVP can be understood by studying the image.

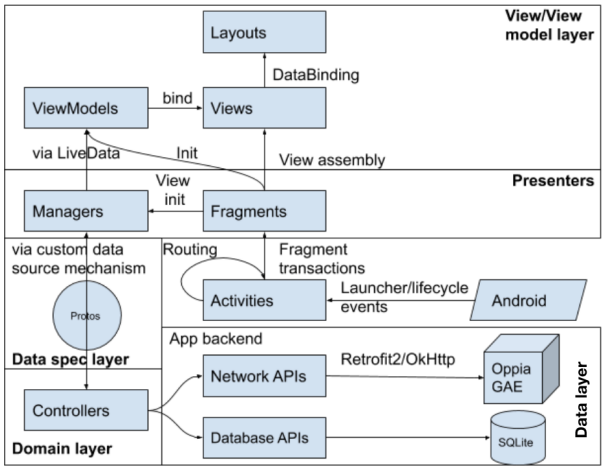
The image on the right is a flowchart of **MVVM app architecture**. Here as well, the Model layer consists of the Business Logic and Data. The View layer contains the UI elements and does not consist of any logic. The ViewModel layer consists of the UI logic. It doesn't know which view is going to use it because of no direct referencing. It uses Observables to update changes to the UI. The image shows the event flow of MVVM architecture.



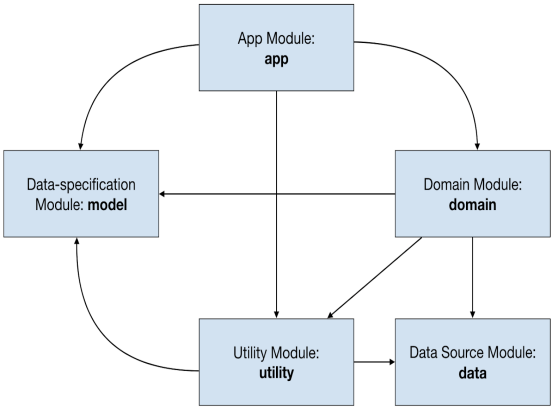
MVVM uses data binding and is therefore a more event driven architecture. MVP typically has a one to one mapping between the presenter and the view, while MVVM can map many views to one view model. In MVVM the view model has no reference to the view, while in MVP the view knows the presenter.

Thus, the combination of MVVM and MVP fits perfectly to fulfill the requirements of every section of the app. The image on the right displays the **App Architecture of Oppia**. If we see, Oppia follows a 5-layered architecture pattern where the 5 layers are -

- View/ViewModel Layer
- Presenters Layer
- Model/Data spec Layer
- Controller/Domain Layer
- Data Layer



The Views in the View/ViewModel layer are classic views and only have the function of displaying data from ViewModel. **The ViewModel** in the View/ViewModel layer are Android ViewModels that listen for and expose changes from the Model layer. **The Model** layer consists of protobuf (Protocol Buffer) and Kotlin data objects that are provided by the Controller layer via LiveData. Using protobuf we can define how we want our data to be structured once, then we can use the special generated source code to easily write and read our structured data to and from a variety of data streams. **The Controller layer** communicates with the Data layer and has access to the app's Database and Network (through their respective APIs). The role of **Presenters** is given to Fragments, they contain the business logic and are basically responsible for binding ViewModels to Views and arranging the UI layout.

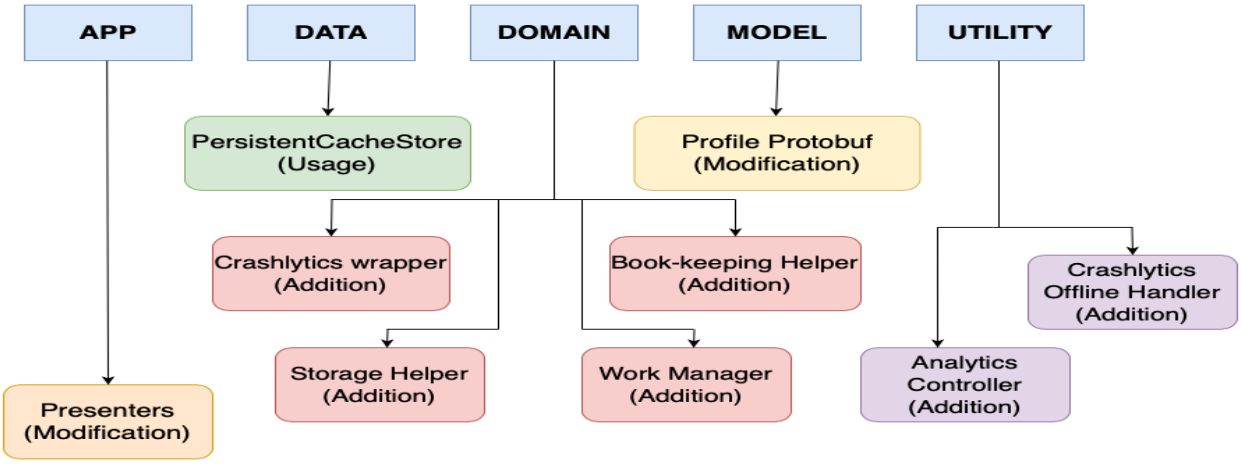


The image on the left shows the Directory structure of the application. The application follows a 5 module project layout. Each module has its own significance -

1. App - contains Fragments, Activities, Views and ViewModel.
2. Model - contains Protobuf used in app
3. Data - provides data to the app via backend or local offline cache storage.
4. Domain - contains business logic of app.
5. Utility - contains utilities that all other modules may depend on.

To implement Analytics Support we might need the help of most of the layers/modules present in the architecture. Module-wise implementation -

1. Domain module -
    a. For creating a controller that handles data collection, recording and storing.
2. Data module -
    a. For creating and retrieving cache storage.
3. Model module -
    a. Analytics protobuf - for storage/retrieval of data from cache memory
4. App module -
    a. ViewModels - Handling UI impressions
    b. Presenters - Logic behind that handling of UI impressions.
5. Utility Module -
    a. For integrating Crashlytics and Analytics logging in the Logger module.
    b. For using OppiaClock for timestamps.
    c. For creating a work manager class.
    d. For creating the Testing Infrastructure class.

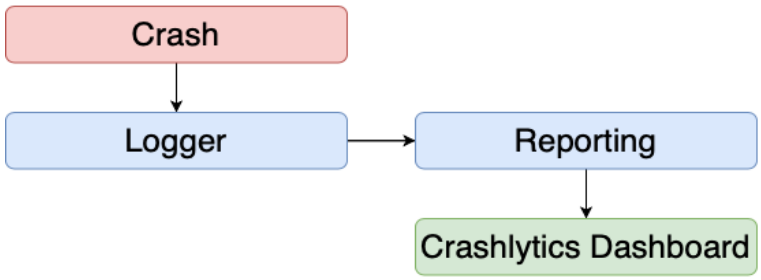


Implementation Approach

Firebase SDK Integration

1. Add Firebase to the app by creating a project in Firebase console and give our application's credentials like package name.
2. Add the google-services.json file.
3. To your project level build.gradle add -
   classpath 'com.google.gms:google-services:4.3.3'
4. Add the plugin using apply plugin: 'com.google.gms.google-services to your module level build.gradle.

Crashlytics Implementation



Crashlytics Mechanism

SDK Integration -

1. Select Crashlytics from the left nav panel of the console and select the app from the dropdown at the top.
2. Click Set up Crashlytics and further select This app is new to Crashlytics (it doesn't have any version of the SDK)
3. In the project level build.gradle add
   - maven {url 'https://maven.fabric.io/public'} [Maven repository]
   - classpath 'io.fabric.tools:gradle:1.31.2' [Crashlytics plugin]
4. In the app-level, domain-level and utility-level build.gradle files add
   - apply plugin: 'io.fabric [Plugin]
   - com.crashlytics.sdk.android:crashlytics:2.10.1 [Dependency]

We can customise the crash reports via custom logging of data. We can -

a. Set user identifiers
b. Add custom keys
c. Add custom logs

d. Log non-fatal exceptions.

However in this project we will only be setting the unique profile Id of the end-user as the user identifier. For Crashlytics' implementation we can set up its custom logging in our Logger class of the Utility module.

We can add a function in the Logger class that sets the unique profile Id of the current user to the crash report using `Crashlytics.setUserIdentifier({{profileId}})`. So when the crash occurs the Crashlytics report has this identifier through which we can further analyse our app's performance according to the user. However, I don't think that logging user IDs will be a nice idea because crash reports are not user specific, they are more on the generic side of things.

Also, there is one feature we can implement to report failure cases in service code and to report the cases where data is requested but it fails to load.

- Crashlytics Wrapper [Domain Module - Addition]
  - This wrapper will enable custom logging to the Crashlytics Dashboard.
  - In this, we will create a method and report the exception using `Crashlytics.logException({{exception}})`
  - Then we can call this method wherever we want to report a non-fatal exception, specially in case of manual uploading of crashes that are stored offline. The Crashlytics Dashboard labels these exceptions as Non-Fatal for the easy viewing of the user.

Optional Crashlytics Implementations -

- **Unique Profile Id**
  a. Adding an element for unique profile id in the ProfileId data structure of profile protobuf in the model module.
  b. Adding a method to generate the unique profile id for each user in the ProfileManagementController of the Domain module.
  c. We can use `var uniqueID = UUID.randomUUID().toString()` to create globally unique identifiers. We can use this uniqueID in combination with the internal_id to provide globally unique IDs to every profile in an installed instance of the app.

Performance Monitoring Implementation

To monitor the App-Health of your application we are going to use Firebase Performance Monitoring. This service can be implemented by simply adding the Performance Monitoring SDK to our app by -

a. In the app's module level build.gradle we have to add -

- implementation 'com.google.firebase:firebase-perf:19.0.5' [Dependency]
- apply plugin: 'com.google.firebase.firebase-perf' [Plugin]

b. In the app's project level build.gradle we can add -
- `jcenter()` [Bintray repository]
- classpath 'com.google.firebase:perf-plugin:1.3.1' [Classpath for plugin]

Optional Performance Monitoring Implementation -

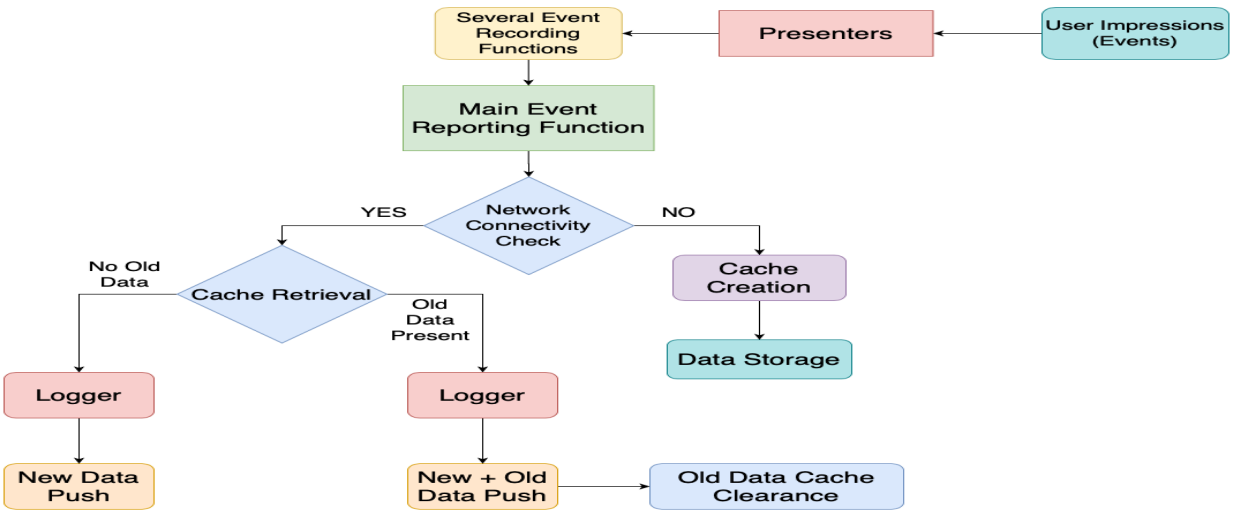
- **Custom Traces**
  a. We can add custom traces to our application from one point to another and measure its performance by tracking its duration. They can be implemented by-
  Val myTrace = FirebasePerformance.getInstance().newTrace("test_trace)
  myTrace.start()
  // code that you want to trace
  myTrace.stop()

  b. We can also add up to 5 attributes along with the traces using -
  myTrace.putAttribute("experiment", "A")
  We can also read a particular scenario, remove a scenario and read all the attributes at once.

Analytics and Offline Storage Implementation

Overall Approach : It is divided into two parts.

1. Part 1 -

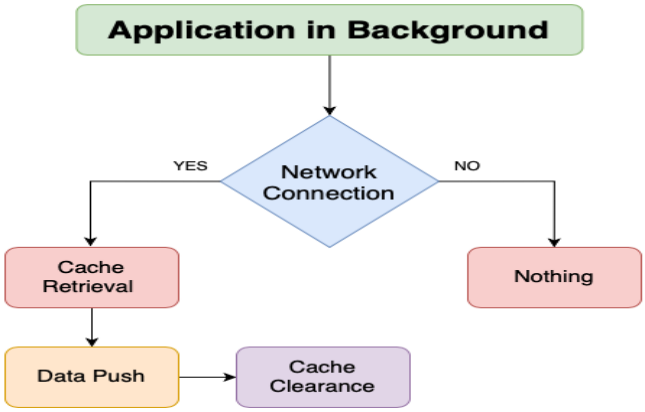


The above approach is used when a user impression is recorded and data is logged. Let's understand this approach using an example. Suppose, the user clicks on a button. The button click is handled at the Presenter layer wherein the timestamp of the event is recorded using a Custom Timestamp and is sent as a parameter to the underlying Event Recording function.

The event recording function accepts the timestamp, adds the event name(say "button_click") and forwards it to the Main Event Reporting function. This function then checks for network availability. If available, a check for old cached data is made and upon its retrieval, the data (old + new) is pushed to the online console using the function defined in the Logger Class.

Upon successful push, the old data is cleared off from the memory. But if there is no network connectivity, the same data is stored in the cache memory.

2. Part 2 -



The above approach comes into play when the app is no longer in the foreground. This approach is handled via a Work Manager class which will work as a Foreground Service. Through this approach we will be able to check network availability periodically and once connectivity is established, we can retrieve the data and push it to online. Upon successful upload the data will be cleared off the cache memory.

- **SDK Integration**

    - Add `com.google.firebase:firebase-analytics:17.2.3` dependency to your app-level, domain-level and utility-level build.gradle files.

- If the Firebase project doesn't have Google Analytics enabled, it can be implemented from the Integrations tab of your Settings > *Project settings*.

- **Logger class modification** [Utility Module]

  - An event logger function to log events to Analytics.
  - `firebaseAnalytics.logEvent(String name, Bundle params)` can be used to log the event to Analytics.
  - `firebaseAnalytics.setCurrentScreen(this, screenName, null /* class override */)` can be used to provide customised names to the Screens of our application.

- **Presenters** [App Module - Modification]

  - To record the data of a UI component, we will have to modify the Presenter Layer of those fragments which handle that component.
  - The Presenters will be used to link the Event Recording functions (defined in the Analytics Controller) to their respective UI components by setting up their onClick listeners.
  - The timestamp (obtained from OppiaClock) of the occurrence of that event will be passed on as the parameter to those functions.

- **OppiaClock** [Utility Module - Usage]

  - Using this class we can provide the required timestamp for event recording.
  - `oppiaClock.getCurrentCalendar().timeInMillis` does the job for us here.

- **Storage Helper** [Utility Module - Addition]

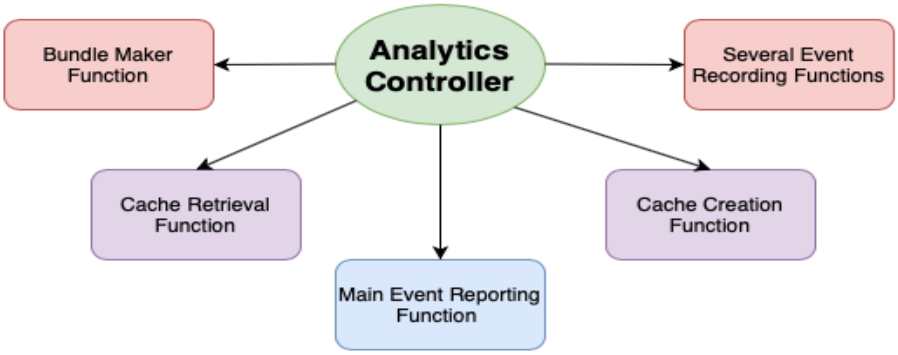
  - This helper class will be more of a Storage wrapper for offline events.
  - It will consist of methods to read event/ crash records, delete event/crash records and to delete all records.

- **Book-keeping Helper** [Utility Module - Addition]

  - This class will help us in gaining an extended control of what gets stored in our cache storage.
  - It will consist of methods to check if the size of our existing storage is under a specified limit. If the limit is exceeded, it will make use of Storage helper class to remove certain records based on their priority and recency.

- **Analytics Protobuf** [Model Module - Addition]
  - The addition of this protobuf will allow us to efficiently store and retrieve our stored records. It will also help us in the book-keeping of records.

- **Analytics Controller** [Domain Module - Addition]



  - The above image shows the kind of functions that the Analytics Controller will contain to get the job done. So let me explain to you the exact working of this.
  - The Event Recording functions get timestamp as an input and have the Event name stored with them in the form of a variable. These values are passed on to the Main Event Reporting function.
  - The Main Event Reporting function is the heart of the Analytics Controller. It takes the timestamp and event name as input and serves them as an input to the Bundle Maker function which returns a bundle containing those two values.
  - Then using `networkConnectionUtil.getCurrentConnectionStatus()` it checks for network connectivity.
  - Based on its outcome the cache retrieval function or the cache storage function is called.
  - The cache retrieval function and the cache storage function use the `PersistentCacheStore` to retrieve and store our data.
  - The retrieved data is pushed along with the freshly recorded data to the online server if there is network connectivity and upon successful retrieval, the cache data is removed from the memory. On the other hand, in the absence of network connectivity the data is stored using the cache storage function.

```
private fun bundleMaker(string: String, long: Long): Bundle {
  var bundle = Bundle()
  bundle.putString("event_name", string)
  bundle.putLong("event_timestamp", long)
  return bundle
}

private fun mainEventReporting(eventName: String, timestamp: Long){
  var bundle = bundleMaker(eventName ,timestamp)
  when(networkConnectionUtil.getCurrentConnectionStatus()){
    NetworkConnectionUtil.ConnectionStatus.NONE -> cacheStorage(bundle)
    NetworkConnectionUtil.ConnectionStatus.LOCAL -> cacheRetrieval()
    NetworkConnectionUtil.ConnectionStatus.CELLULAR -> cacheRetrieval()
  }
}

fun button_click_event_recording(timestamp:Long){
  val eventName = "sign up button click"
  mainEventReporting(eventName,timestamp)
}
```

Sample Implementation

- **Work Manager** [Utility - Addition]

    - I would like to implement the Work Manager Class in Oppia-Android. It would help us in achieving the part 2 of our approach, which is running the application in background to upload the stats whenever network connectivity is regained.

    - WorkManager is an Android library that gracefully runs deferrable background work when the work's conditions (like network availability and power) are satisfied. It is often best suited to work in combination with Foreground Service.
    - Using a foreground service tells the system that the app is doing something important and it shouldn't be killed. Foreground services are visible to users via a non-dismissible notification in the notification tray.

    - All work must be done in a ListenableWorker class. There are two types of work supported by WorkManager: OneTimeWorkRequest and PeriodicWorkRequest.

    - In here, we will use the PeriodicWorkRequest to look for network connectivity at regular intervals in the background. The work executes multiple times until it is cancelled, with the first execution happening immediately or as soon as the given constraints are met.

```
val constraints = Constraints.Builder()
                  .setRequiresCharging(true)
                  .build()

val work = PeriodicWorkRequestBuilder<MyWorker>(1, TimeUnit.HOURS)
                  .setConstraints(constraints)
                  .build()

val workManager = WorkManager.getInstance(context)
workManager.enqueuePeriodicWork(work)
```

Sample Implementation

    - We can use OneTimeWorkRequest if we want to perform the task at a specific instance.
    - Now, we will like to initiate this once the app starts. But there might be consequences when we do that as we might end up creating duplicate work requests. So to deal with that we will enqueue our work request using enqueueUniquePeriodicWork(), which will make our work requests unique.

```
    WorkManager.getInstance(this).enqueueUniquePeriodicWork(
        "MyUniqueWorkName",
        ExistingPeriodicWorkPolicy.KEEP,
        myWork)
```

Sample Implementation

Handling Storage Consumption -

We can handle the storage consumption of stats in the following possible ways -

1. **Cache Clearance**
    - Upon successful retrieval and upload, the cache memory can be cleared off using the clearCacheAsync() function of the PersistentCacheStore.

2. **Time Limit**
    - We can put up a time limit (say about a month or two) from the time the event has been recorded and remove the stats after that time using the OneTimeWorkRequest of the WorkManager class.

3. **Size Limit**
    - We can put up a size limit on our stored event records. If the size of the directory storing all records exceeds that limit then we can proceed to deletion of a few records.

4. **Priority check**
    - We can set custom priority status to all the events and then if our storage size reaches its limit then we can delete the records having the least priority.

5. **File Compression (Zip/Unzip)**
    - We can compress the data before we store it using the zip technique and then unzip it upon retrieval.

Zip Function

```
class ZipManager {
    private val bufferSize = 6 * 1024
    @Throws(IOException::class)
    fun zip(files: Array<String>, zipFile: String?) {
        var origin: BufferedInputStream? = null
        val out =
            ZipOutputStream(BufferedOutputStream(FileOutputStream(zipFile)))
        try {
            val data = ByteArray(bufferSize)
            for (i in files.indices) {
                val fi = FileInputStream(files[i])
                origin = BufferedInputStream(fi, bufferSize)
                try {
                    val entry = ZipEntry(files[i].substring( startIndex: files[i].lastIndexOf( string: "/") + 1))
                    out.putNextEntry(entry)
                    var count: Int
                    while (origin.read(data,  off: 0, bufferSize).also { it: Int
                            count = it
                        } != -1) {
                        out.write(data,  off: 0, count)
                    }
                } finally {
                    origin.close()
                }
            }
        } finally {
            out.close()
        }
    }
```

UnZip Function

```
fun unzip(zipFile: String, location: String) {
    try {
        val f = File(location)
        if (!f.isDirectory) {
            f.mkdirs()
        }
        val zin = ZipInputStream(FileInputStream(zipFile))
        try {
            var ze: ZipEntry? = null
            while (zin.nextEntry.also { ze = it } != null) {
                val path = location + File.separator + ze?.name
                if (ze?.isDirectory!!) {
                    val unzipFile = File(path)
                    if (!unzipFile.isDirectory) {
                        unzipFile.mkdirs()
                    }
                } else {
                    val fout = FileOutputStream(path,  append: false)
                    try {
                        var c: Int = zin.read()
                        while (c != -1) {
                            fout.write(c)
                            c = zin.read()
                        }
                        zin.closeEntry()
                    } finally {
                        fout.close()
                    }
                }
            }
        } finally {
            zin.close()
        }
    } catch (e: Exception) {
        e.printStackTrace()
        Log.e( tag: "TAG",  msg: "Unzip exception", e)
    }
}
```

Sample Implementation

Hooking up of data to server in future -

- We can hook up the data to the server in the future using the Network API via OkHttp or Retrofit2 (As used in Oppia-Android's codebase)
- We can take the bundles of the events and use the POST request call to upload the bundles to the server using the API link.

```
String doPostRequest(String url, String json) throws IOException {
    RequestBody body = RequestBody.create(JSON, json);
      Request request = new Request.Builder()
          .url(url)
          .post(body)
          .build();
      Response response = client.newCall(request).execute();
      return response.body().string();
}
```

Example code for a POST request

- **Note**

1. The Firebase Analytics rejects data which was recorded more than 72 hours before its reporting. But we can bypass that requirement by logging the event only when the Network is present and otherwise storing it.
    - ✓ The major advantage of this approach will be that there will be no loss of analytical event recordings and we can get more accuracy in our data.
    - ✓ The only disadvantage is that the timestamp of data recording will be hampered. However, **we can manage** this by recording the timestamp of every event while capturing them. This recorded timestamp can then be stored in the bundle as an attribute and the data can be visualised in the Analytics dashboard according to it using the filter options. For further analysis (maybe using BigQuery) a CSV file of the filtered data can be downloaded from the event panel.

2. As the profile id is not needed in the uploaded stats but can be used for offline storage, we can initially store them in the bundles but then later unpack and repack the bundle while getting rid of the profile ID in between. This repacked bundle which doesn't have a profile ID stored can then be uploaded to Analytics.
3. I don't think there is a need to provide offline support to stats other than of Analytics. This is because Crashlytics can store data locally and upload crashes whenever the internet is back up and running. Similar is the case for Performance Monitoring SDK. Both of them don't lose data if its storage is 72 hours prior to reporting, it's always reported.

Testing Approach

Testing is a very important part of any application. It confirms that the way the application is behaving is the same as the developer wants before the application is made public. Currently in Oppia we are mainly using JUnit, Robolectric, Espresso, LeakCanary and Performance testing to maintain the correctness of the app.

In this project we'll have to test the following things -

1. Firebase Crashlytics
2. Firebase Analytics
3. Work Manager
4. Storage Helper Class

Firebase Crashlytics Testing :

- We'll unit test this feature using mocks.
- We'll use the Mockito library to mock the class and check if the function was called at the appropriate time.

Firebase Analytics Testing :

- Similar to Firebase Crashlytics testing, we'll unit test this functionality using mocks.

Work Manager Testing :

- Testing the WorkManager class is also pretty straightforward. We can start the work synchronously and match its value using assertThat().

```
  @Test
  fun testRefreshMainDataWork() {
    // Get the ListenableWorker
    val worker = TestListenableWorkerBuilder<SeedDatabaseWorker>
(context).build()

    // Start the work synchronously
    val result = worker.startWork().get()
    assertThat(result, `is`(Result.success()))
  }
```

Sample Implementation

Storage Class Testing :

- Our storage helper class will contain methods for cache creation, retrieval and deletion. It'll be made using the PersistentCacheStore
- We'll create a fake of our storage and then test the class using it.

MILESTONES

Milestone 1

Key Objective:
When a crash or an event arises, the corresponding crash report or event report is uploaded to the Firebase console when there is network connectivity. A full suite of unit tests will be written for both the crashlytics logging wrapper and the event analytics logging wrapper.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Addition of Firebase SDKs		25/05/2020	26/05/2020
1.2	Crashlytics Wrapper	1.1	01/06/2020	03/05/2020
1.3	Analytics Controller (Part 1)	1.1	07/06/2020	09/06/2020
1.4	Analytics Controller (Part 2)	1.3	11/06/2020	12/06/2020
1.5	Linking Presenters	1.4	15/06/2020	17/06/2020

Backlog clearance + Documentation buffer for Milestone 1 ----> 18/06/2020 - 22/06/2020

PR Descriptions -

1. Addition of Firebase SDKs
    - Adding the Firebase SDKs of Crashlytics and Analytics in their respective modules.

2. Crashlytics Wrapper
    - Implementing a Crashlytics Helper class to implement crash reporting in the project.
    - This class will contain a method to log screen names during crashes along with methods to report exceptions.
    - Implementing this function in all the presenters to provide accurate additional logs.

3. Analytics Controller (Part 1)

    - Adding the Event Recording functions to record user impression and adding the bundleMaker function to create bundles using the data.

4. Analytics Controller (Part 2):
    - Adding the Main Event Reporting function to the Analytics Controller class.

5. Linking Presenters
    - Adding the event recording functions to the presenters to link it to the UI elements that need recording.

Milestone 2

Key Objective:
In the absence of network connectivity, event analytics stats are stored offline. Logs will be stored in disk, with the maximum size of the files determined by a constant specified in the code. If the max size is reached, logs will be removed based on their priority and their recency (i.e. timestamp). This functionality will be achieved via a log persistence class. A full suite of unit tests will be written for the log persistence functionality.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Analytics Protobuf		26/06/2020	28/06/2020
	---Exam Week--- [1/07 - 7/07]			
2.2	Storage Helper Class	2.1	10/07/2020	12/07/2020
2.3	Book-keeping Helper Class	2.1, 2.2	19/07/2020	21/07/2020

Backlog clearance + Documentation buffer for Milestone 2 ----> 21/07/2020 - 27/07/2020
PR Description -

1. Analytics Protobuf
    - Adding a protobuf for handling offline storage and retrieval of analytics events.

2. Storage Helper Class
    - Implementing a helper class that will consist of methods for creating, retrieving and deleting cached storage.

3. Book-keeping Helper Class
    - Implementing the book-keeping helper class which will enable us to have control over our storage.

Milestone 3

Key Objective:
In the absence of network connectivity, crash reports are stored offline. In the presence of network connectivity, these offline records of crashes/events are uploaded to the firebase console using a work manager. There will be a full suite of unit tests for this work manager. In addition, event stats for the following 5 views of the oppia application can be analysed via the Firebase analytics dashboard: the home screen, and the 4 tabs of the topic page (i.e. info, lessons, practice, and revision). There is also clear documentation for how to create similar dashboards in the future for other views.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
3.1	Offline storage : Crashlytics		07/08/2020	09/08/2020
3.2	Work Manager	3.1	14/08/2020	16/08/2020
3.3	Analytics Dashboard	1.5	17/08/2020	19/08/2020

Backlog clearance + Documentation buffer for Milestone 3 ----> 19/08/2020 - 24/08/2020

PR Description -

1. Offline Storage : Crashlytics
    - It involves creating a functionality to store exception logs when there is absence of network connectivity.
2. Work Manager Class
    - Adding the Work Manager class to the project which will enable uploading of offline stats in the background whenever there is network connectivity.

Optional Sections

Future Work

There are a few things that aren't in the scope of this project but if implemented, they will provide a lot of help to the using teams. Those things are -

1. Feedback Reporting
    - This functionality was earlier a part of this project but was taken off due to its size and the amount of work it will require. This feature is a really good step forward if we want to develop the app to newer heights. It will increase user engagement in the development process.

2. Room Database Implementation
    - The implementation of the room database will provide a better and optimised approach for offline support.

3. Custom Traces
    - We can implement custom traces for better performance monitoring. By implementing this we can take full advantage of the Firebase Performance Monitoring SDK. Using its data, we will be able to better optimise our application and thus increase end-user experience.

Additional Project-Specific Considerations

Security

No, the feature that will be implemented won't 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. The feature will only provide analytics data of the users to the various teams at Oppia.

Accessibility (if user-facing)

My feature is not user facing but yes it provides analytical capabilities to the application and thereby provides a user interface to the developer teams at Oppia. The scope of the usage of the user interface provided by this feature is limited as it will only be used by Oppia teams.

Documentation Changes

Yes, modifications should be made to the Oppia wiki for the support of this feature and to provide additional guidance to maintainers of Oppia instances. We can add some of the information from the Product design and the Technical Design sections of my proposal. We can also mention the implementation methodology for the maintainers using flowcharts which will enhance developer understanding.