GSoC 2021 Proposal

Implement feature flags & platform parameters

Arjun Gupta

About You

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

Oppia is one of the most helping and active organizations I encountered. The procedure of design docs and PR reviews gives insights of professional work. The developer community is so welcoming and maintainers listen to even the smallest doubts we have.

I have a keen interest in knowing the product release cycle which is followed in industry and using Feature Flags to aid the release process is something which even Tech Giants follow. Implementing this feature in Oppia will give me a better insight about this process, and hence I want to propose my idea of implementation for this project.

Prior experience

I have been learning and working in Android App development for more than one year now, and within this short period, I have managed to complete three internships in small scale startups. A little overview about the technicalities of my job while working in them is as follows -

1. GGLabs-
    a. Developed an Android App (Param), entirely written in Java and using Firebase as a backend service.
    b. Learnt using Cloud Functions, Push Notifications, Cloud Database and Cloud Storage
2. Kohbee-
    a. Developed a Cross-platform application (KohbeeLive) in Flutter while using their custom backend setup
    b. Learnt using MVVM architecture and State Management

3. Esaathi-
    a. Developed a Voice Assistant Library(SDK) in Java. It just needs JSON coded instructions and must be used as a dependency in Client Android Apps
    b. Learnt about working with Android File Storage System, Threading, using multiple modules and exporting the code in the form of AAR

Due to these internships, I have a good amount of experience in

- Promising correct deadlines and working according to them
- Good communication skills so that I can convey my problems clearly
- Timely share my progress via meetings to get feedback and improvise

I have been contributing to Oppia-Android for almost 6-months now. Back then, I started my Open Source journey with Oppia by solving good first issues related to changing file names. After completing Two successful PRs, I attended Team meetings and got to know the other contributors and maintainers.

I have worked closely with Rajat Talesra to make Oppia-Android more Accessible so that privileged audiences can easily use it. In this process, I filed many Issues (almost 25) and prepared a Google Sheet to take review from @Chantel Chan. I was also assigned to review Pull Requests related to Accessibility or the ones that just needed an extra eye. I am also a member of the Oppia CLAM Team (Core Learning and Mastery) and have helped some new contributors.

Few Achievements -

- Developed and published an Android Application (Param) from scratch, which has crossed 5000+ downloads on PlayStore
- Participated in SLOP-2020 and was the second-highest contributor among the other participants contributing in Oppia. My overall position was 25th among all the participants under SLOP.

Merged PRs -

- #2010 ,#2034 ,#2110 ,#2111 ,#2112 ,#2113 ,#2114 ,#2115 ,#2121 ,#2153 ,#2158 ,#2160
  Renamed All the dimensions inside Oppia-Android in place of contextual names which justify their use
- #2453, #2454, #2456, #2457, #2507, #2508, #2519, #2458, #2626, #2649
  Solved some of the Accessibility Issues following Chantel's suggestion
- #2929 - Fixed Reading Test Size Ui for increasing and Accessibility and changed the related tests (Will get merged soon)

Contact info and timezone(s)

Email - arjupta.90@gmail.com
University - Indian Institute of Technology (BHU) Varanasi
Gitter - @ARJUPTA
Mobile - +91 6306951818
LinkedIn - Arjun Gupta
Time Zone - Indian Standard Time (Lucknow, Uttar Pradesh (GMT+5:30))

Preferred Communication - Hangouts for Essential Features rest can be explained on Email

Time commitment

I have given a brief description about my time commitment for the project. I will try to keep the major portions of my project to be in the early days of the complete timeline so that milestones are completed on time.

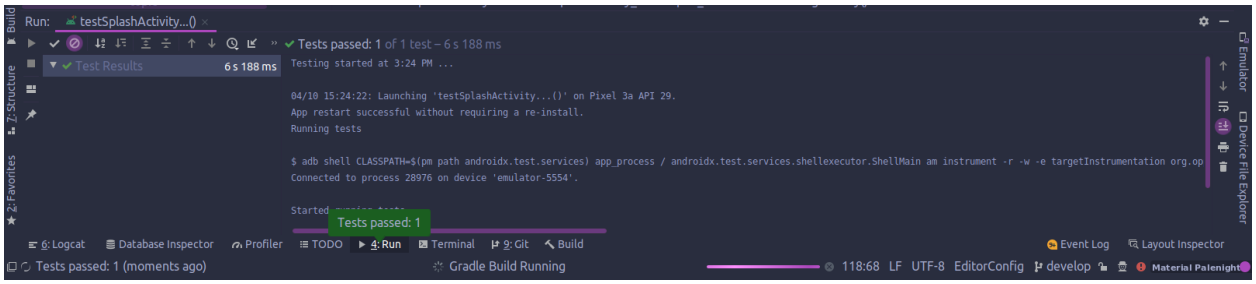
- 7 June to 22 July (Till the end of first evaluation)
    - (Monday - Saturday) 6-7 hours per day
    - Sunday will be spent reviewing or fixing my code which might be hindering other contributors or vice versa.
    - 36 - 42 hours per week

- 23 July to 23 August (For the rest of the time period)
    - (Monday - Friday) 4-5 hours per day (Might be lesser due to college lectures)
    - (Saturday - Sunday) 6-7 hours per day
    - 30 - 37 hours per week

Essential Prerequisites

- I am able to run all the Instrumentation tests of a test suite

- I am able to run single Instrumentation Test from a test suite



Other summer obligations

My final exams will be over by 12 May, and after that, Summer vacations will start. So I will be fully obligated to the organisation.

Communication channels

I prefer to meet (google meet) two times a week, one meet at Monday and other one to be at any day as me and my mentor finalise, but it must be before the weekend so that I can take review over any major problem I may face and improvise any further plans for the coming week. Rest of the times we will be communicating over PR comments and hangouts if needed.

Application to multiple orgs

I will apply only for Oppia-Android Project

Project Details

Product Design

This Project aims to help the developers in the Oppia Android team build features, solve issues and commit them into the develop branch without affecting the releasable code. In other words, with the help of these parameters, developers will be able to keep their code up-to-date with the develop branch even if the feature that they are working on isn't completed yet and shouldn't get merged into the release-ready code. These parameters will also help us have more granular control over different parts of the codebase. We will have a medium to tweak the App behaviour remotely without the need for a new release.

What are Feature Flags or Platform Parameters?

These Parameters or Flags, in general, are configurations for the app used to tweak the behaviour of systems. One such configuration that's noteworthy is using them for feature gating, meaning simple values that direct the if-else statements to enable or disable certain parts of the code. These values can be a boolean, string or integer type which we can control to toggle between the blocks of code we want to run.

```
boolean FirstTime = true;

if(FirstTime){
    Log.d(TAG,"Hello First Time User");
} else {
    Log.d(TAG,"Welcome Again User");
}
```

Feature Gating

In this example, we used a boolean parameter as a flag, but we could have also used a string/integer value and compare it to another string/integer to work as a flag. This example gives us an idea of using Parameters to gate Features.

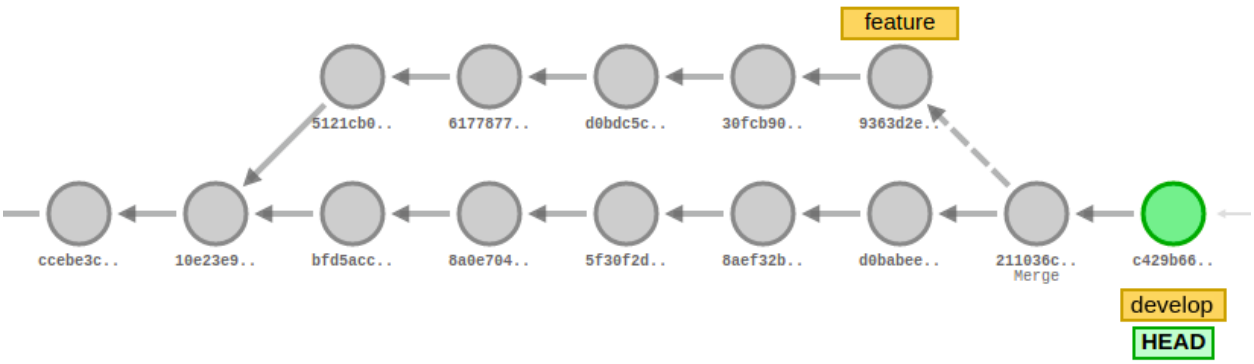
```
String OperationalEndpoint = "https://Oppia.org/..";

FetchValue( ){
  makeRequest (OperationalEndpoint);
  ...
}
```

Another functionality can also be derived from these parameters to provide different configurations to the App. Here we can change the Operational Endpoint Parameter remotely and hence can control the App's behaviour.

How will these Parameters Help Us?

(1) Division of Implementation into smaller chunks and Faster Releases

Whenever we are working on implementing a new feature, we usually make a feature branch out of the develop branch and then make all the code changes required on this new branch. After completing and testing the implementation, we merge this branch into the develop to include the code changes we did in the feature branch. But this merging isn't always easy because it creates merge conflicts as the develop branch has moved on until we were working on the feature branch, and now there are commits from other branches that have already merged. Hence this will slow down development and create a lag in releasing the essential feature.

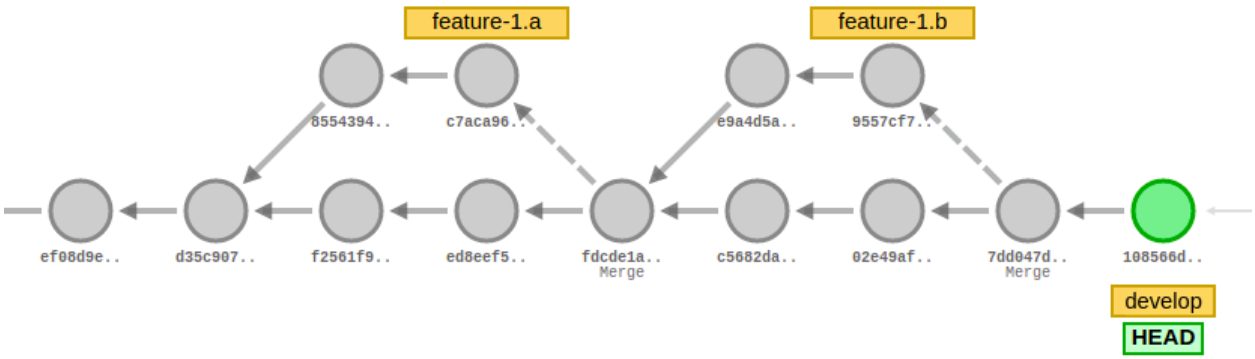


(This is the git graph we get while operating in the traditional way of developing a feature in a separate feature branch without dividing it into smaller chunks.)

We see that till the time our feature branch got merged, the develop branch has code changes from other branches which are not included in the feature branch and hence these changes will create conflicts on time of merging our branch, which is not up to date with develop branch for a long time.

We can reduce this problem if we break our implementation of features into smaller blocks and merge these blocks individually as soon as possible. Doing so will help us decrease the number of conflicts, but it means that we will merge parts of a feature before even developing and testing it thoroughly. Hence our develop branch will not be release-ready as it will contain incomplete implementations of a feature. So even in this case, we create a lag in the release process.

If we use a simple flag to hide our incomplete implementations inside a typical if-else block, then we will be able to divide our feature into smaller chunks and merge it to develop. In this way we will get lesser merge conflicts, and we won't affect the release-ready code, thus preventing any further lag.



(This is the git graph if we use flags to hide our code-under-development and merge it timely into develop branch.)

In this way, we also save ourselves from accidentally shipping the code that hasn't been completed yet. For example, in the graph above, if the commit fdcde includes a fork to release branch, then feature-1.a would have got shipped to users before even getting completed. Hence using a Flag to prevent this scenario will be helpful

(2) Granular Control Over Both Under-Development and Released code

There are times when we need to control the App remotely without making a new release (maybe because the change required is too minuscule or there is a requirement of instant action). For example -

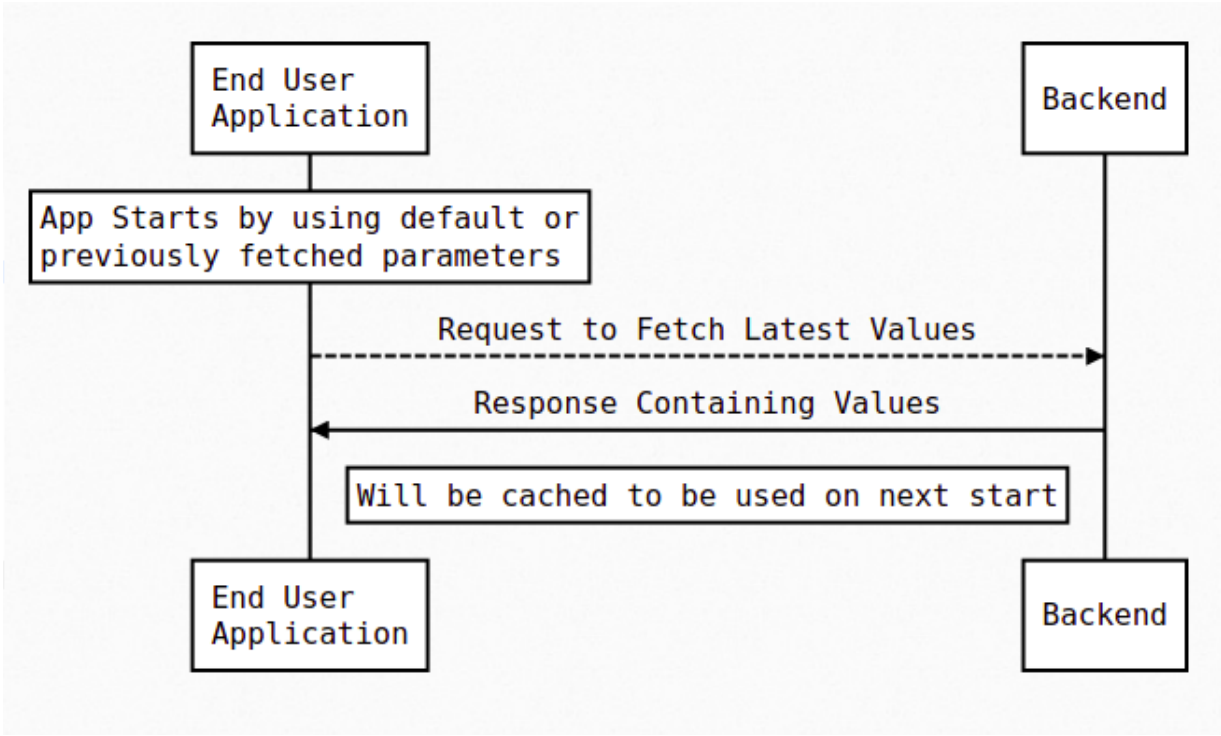
A. When we want to change the behaviour of certain feature that is currently being used

- We may we plan to enable one specific feature only after a particular release version so we will be needing to block its code from running
- We may intend to rollback a newly developed feature from getting used further based on reviews by a particular group of users or because of any bug found later on.

B. When we want to change some configuration value which was set during compile time

- If there is a need of reducing the app's QPS to certain APIs in cases when the backend is being overloaded then we can change the url provided to the app, remotely as per the requirement.
- If there is a change of endpoints for the storage buckets the App currently use, we will need to provide the new endpoint values to every App as soon as possible.

These things are impossible until we include platform parameters in our implementations and let our app depend on them completely. As the app will fetch these parameter values from the backend, we would be able to control the behaviour of App by changing these values remotely



Cycle of refreshing the Parameters values which can be changed from backend

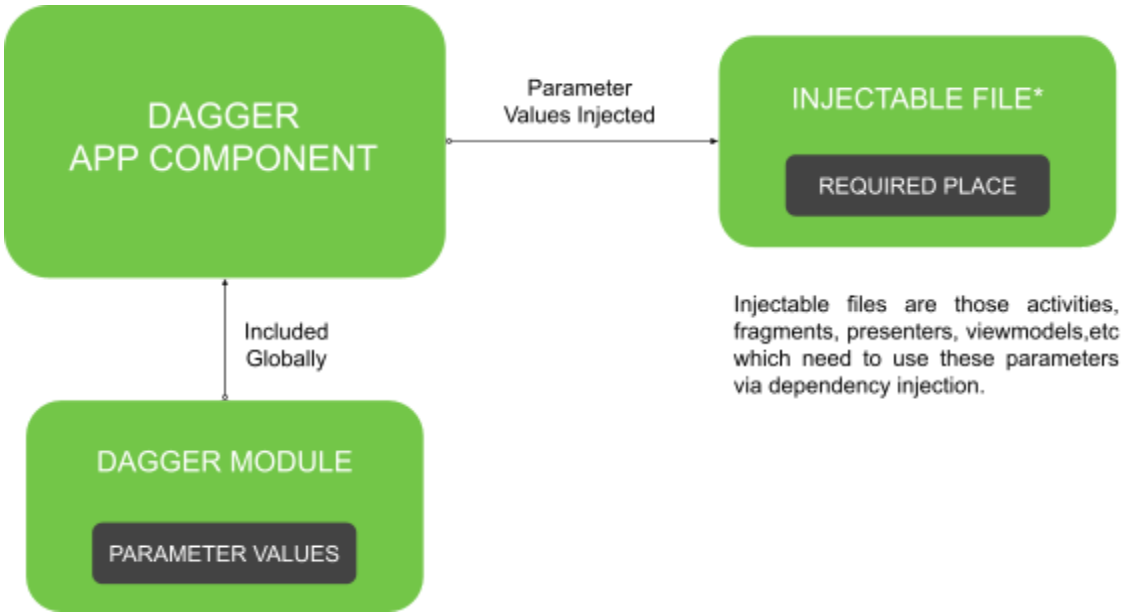
Walkthrough of Final Implementation

After completing this project, developers of Oppia-Android will have a scalable code architecture that will enable them to use Platform Parameters in the codebase.

The deliverables for this project will be-

1. Utilisable Parameter Values -

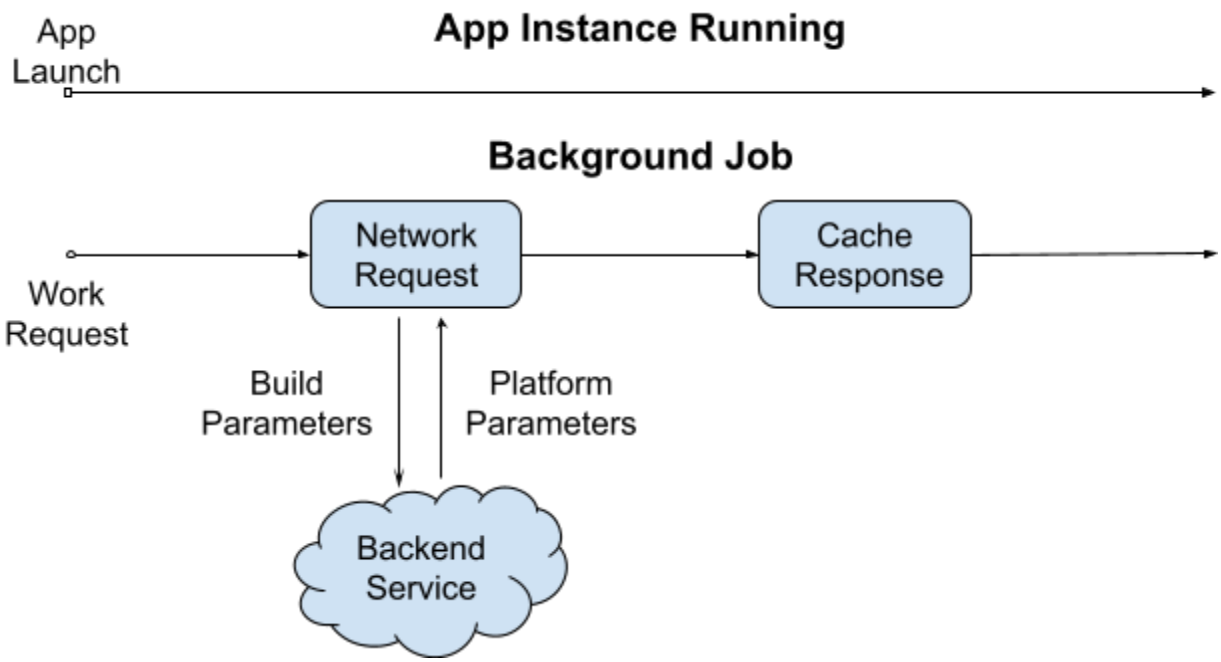
   As the Codebase already uses Dagger for Dependency Injection, we will use it to provide the Parameter Values where needed. We will do so by generating a dagger module that will exist for the complete lifetime of the App, which means these parameter values are globally available and can be injected at any point in time while the App is running.



   The Parameter Values can be of any data type, be it boolean, integer or a string. Classes that are registered to be injectable by Dagger can use these values, which are globally available in the App. Dagger will get these values by the backend's response that will get cached on the User's device via Persistent Cache Store. We already use Cache-Store in the Codebase for storing any data locally on the User's device with an asynchronous mechanism.

2. Refreshing Mechanism -

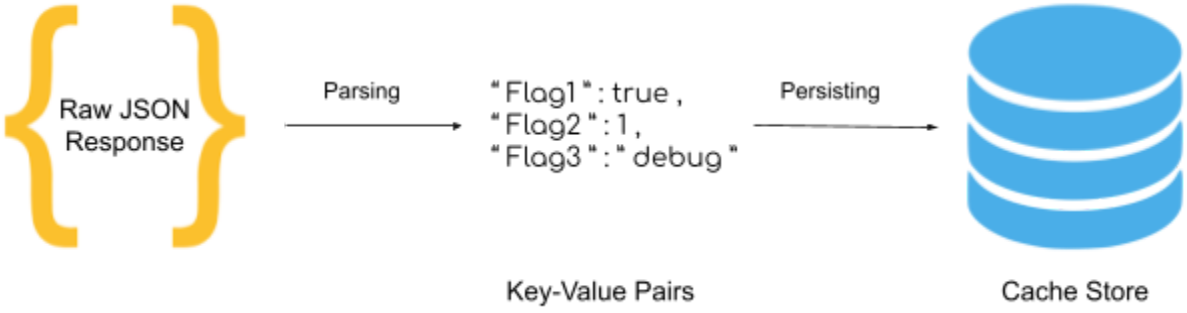
   We will be able to remotely control these Parameter Values even for the App which is in Production. For this purpose, we will let our WorkManager trigger a network request periodically (This job will run in the background with a predefined time gap). This network request will fetch the latest Parameter Values from the backend-



Making a network request is a time-consuming process; therefore, we will perform it asynchronously (which means it will not block App's UI on completing this network request) in the background. We won't be using these fetched results in this same App run to avoid mixing and using two different versions of parameter values. Thus we will depend upon the previously fetched/default values to be used in this run. Afterwards, when we receive the response, it will be cached by the WorkManager to be used on the following App start. Thus we get a lightweight syncing mechanism as we only make a single network request, which also does not block our App from working fine.

3. Parsed and Persisted Response -

Whenever we request parameters from the backend, it will return us a JSON file as a response. This file will contain an array of parameters along with their property and values (Authorized people maintain these flags with the help of the Feature Gating Console). After we fetch the response and Parse it with Moshi Converter's help, we can store it inside the Cache-Store via Persistent Cache-Store Classes that we already have. Data Structure for storing these values can be a simple map of key and value pairs.



Parsing the response just after a successful fetch will save us from the cost of storing the raw JSON file. So on the next start of the App, Dagger Modules can use this previously cached response and provide it without much delay or blocking of UI.

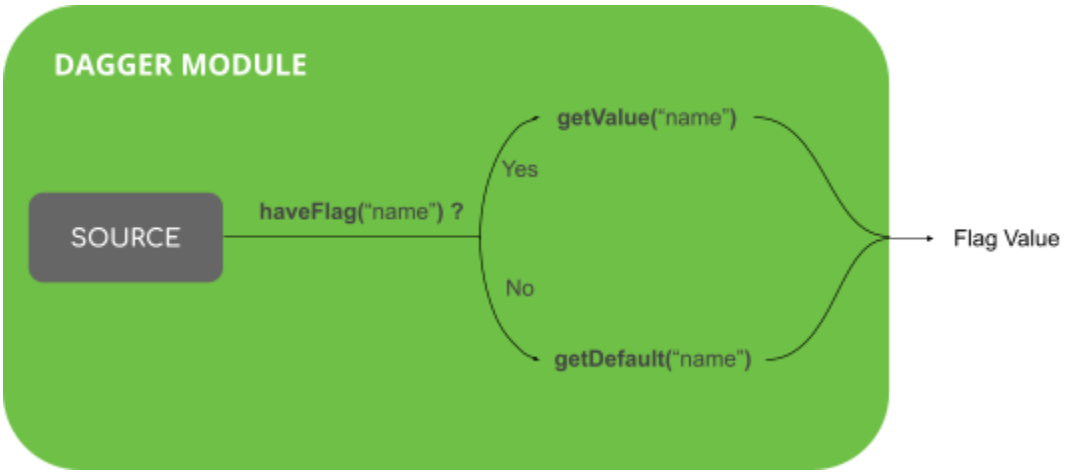
4. Fallback Mechanism -

There can be cases when we would need a mechanism to handle any failures while providing Parameter values. It can happen because of

- Cancelled network request or no internet connectivity
- Exception in Parsing because of invalid response
- Error while caching the values due to any Low Storage exception
- Failure in retrieving the values from Cache-Store
- Data wipe off by the User or any other program etc.
- The first time opening of App, which won't have any previously cached values

All these cases leave the App to

a. Operate without any parameter values
   In this case, we will rely on the parameter's default values, which are getting exported with the App on every release. Dagger will use them only if it cannot find the specific parameter in the source, which can be because of any cases mentioned above or anything else.



   We will keep a check on every parameter that we provide to have a default value; the set of default values that we hardcode must be compatible with each other.

b. Operate on the previous version of parameter values
   In this case, we do not need to use default values, as the previous version of values were also a valid configuration. And we know that the App will always try to fetch new values periodically with WorkManager, thereby getting a new set of values whenever the fetching is successful.

5. Scalable Architecture -

   While working on a local machine, adding new Platform Parameters will be as simple as creating a new constant for the parameter name and its default value. But if we want to deploy these new parameters globally, we will also have to make them on the Feature Gating Console

   > Similarly, for Removing Parameters while working locally, we will delete these same snippets. But to make this change globally, we will have to remove the parameters from the backend via the same Console.

   While developing locally, we might need to turn off the fetching of parameters by the App because the changes we perform may not be in sync with the parameters stored in the backend; hence this can create an unexpected issue. For this, we will create one more flag for deciding when to refresh a value and when to not. (Part of Future works now)

Technical Design

Architectural Overview

The implementation for this project will require changes in all the modules that we have in Oppia-Android.

Overview of the need of changes needed in each module -

1. **app** -
   a. Trigger the load of Parameter values from cache store
   b. Consume the Parameters values that are provided.

2. **data** -
   a. Introduce new API for retrieving Platform Parameters
   b. Models and parsers to handle the network response from Oppia Backend

3. **domain** -
   a. New Controller for reading and writing to Platform Parameter Cache
   b. A Dagger SIngleton which can store all the Parameters at runtime
   c. New Coroutine Worker and Factory class which are able to trigger a network request and cache the response
   d. A WorkManager configuration different than the one used for Analytics

4. **model** -
   a. New proto classes for storing with Persistent Cache Store.

5. **utility** -
    a. Dagger Module which provides all the individual parameters.
    b. Constants File containing all Parameter names and default values
    c. Utility classes to help with Dagger injection and WorkManager

Overview of the file changes that are required is as follows -

All these files changes are based on the Implementation Approach. Also look at the Flow of Logic section to get a clear Idea of how things are actually working.

1. app/src/main/java/org/oppia/android/app/
    a. application/ApplicationComponent.kt
        - Include the PlatformParameterWorkerModule for helping with initialising WorkManager for it
        - Include ParameterModule for providing the Map of Parameters from the Source (Dagger Singleton).
    b. splash/SplashActivityPresenter.kt
        - Inject ParameterDatabaseController to trigger a fetch of parameter values from the Cache-Store.
        - Observe the process of Parameter Data Fetching and block the App UI until it is completed.
2. data/src/main/java/org/oppia/android/data/
    a. backends/gae/api/PlatformParameterService.kt
        - New endpoint for fetching Platform Parameters from the backend
    b. backends/gae/model/GaePlatformParameter.kt
        - New model class for representing a single Platform Parameter in the received response.
    c. backends/gae/model/GaePlatformParameters.kt
        - New model class for representing a List of GaePlatformParameter in the received response.
    d. backends/gae/NetworkModule.kt
        - Include a new @provides annotated function for injecting the new PlatformParameterService API with dagger.
3. domain/src/main/java/org/oppia/android/domain
    a. platformparameter/database/ParameterDatabaseController.kt
        - New DatabaseController to read and write ParameterDatabase from the cache with PersistentCacheStore
        - Provide a DataProvider in order to let the SplashActivityPresenter observe the completion of reading of Parameter Values.
        - Inject the Dagger Singleton to store data the fetched from the Cache-Store
    b. platformparameter/ParameterSingleton.kt

        - New simple Singleton class which can be injected by Dagger.
        - It will store the Parameter Map at runtime so as to act like a source for the ParameterModule
    c. oppialogger/loguploader/WorkManagerConfigurationModule.kt
        - Change its location to the outermost directory in the domain module. For eg - domain/src/main/java/org/oppia/android/domain/workmanager
        - Include new PlatformParameterWorkerFactory in the Configuration of WorkManager
    d. platformparameter/worker/PlatformParameterWorkManagerInitializer.kt
        - New Class which is responsible to enqueue a PeriodicWorkRequest to the WorkManager for refreshing the Platform Parameters
        - This class will implement AppStartupStateListener that means it will enqueue a work request as soon as the App Starts.
    e. platformparameter/worker/PlatformParameterWorker.kt
        - New Coroutine Worker implementation for overriding the doWork method to start a network request inside it.
        - Transform and Cache the response from the network with the help ParameterDatabaseController
    f. platformparameter/worker/PlatformParameterWorkerFactory.kt
        - New Custom Worker Factory to initialize the PlatformParameterWorker
    g. platformparameter/worker/PlatformParameterWorkerModule.kt
        - New Dagger Module to return PlatformParameterWorkManagerInitializer as an implementation of AppStartupState Listener
        - It will be included in the ApplicationComponent
4. model/src/main/proto/
    a. platform_parameter.proto
        - New proto file for generating the classes which will be used for storing the Platform Parameters
        - Classes like ParameterDatabase, Parameter, ParameterMap and ParameterWithName will be generated
5. utility/src/main/java/org/oppia/android/util
    a. platformparameter/ParameterModule.kt
        - New Dagger Module that will provide a ParameterMap by extracting it from the ParameterSingleton.
        - It will also provide a DefaultMap which will be hardcoded in the ParameterConstants file.
        - This model will be included in the App Component to enable parameter injection globally
    b. platformparameter/ParameterHelper.kt
        - A new helper class which will take the name of Parameter as an input and checks for its corresponding Parameter first in ParameterMap and then in DefaultMap.

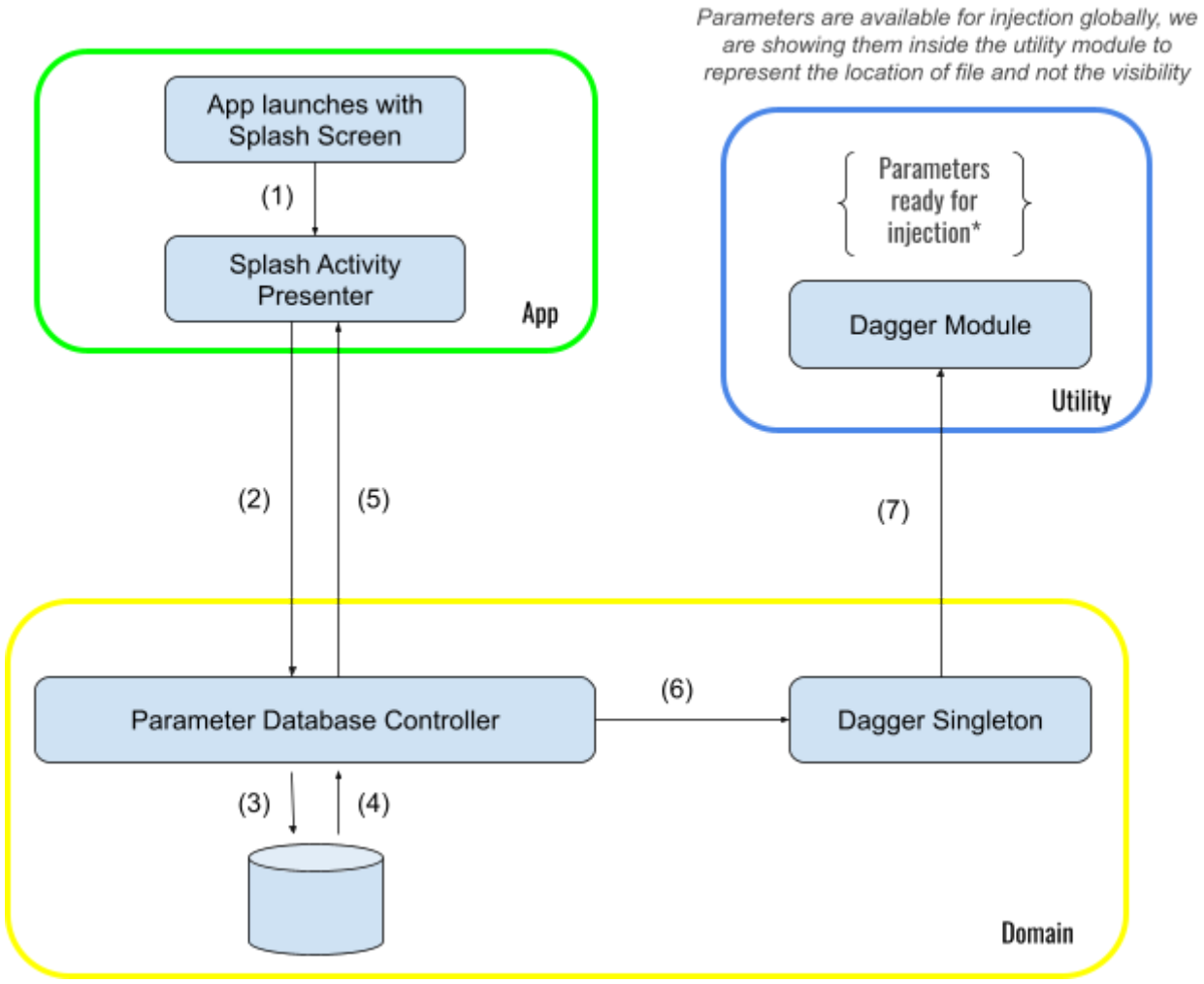
- It will have three methods stringVal( ), boolVal( ), intVal( ) which takes the name of Parameter as parameter and return string, boolean and integer respectively.

c. platformparameter/ParameterConstants.kt
   - New constants file that will contain the names and the default values of each PlatformParameter we define.

Flow of the Logic for this Project

We can see the complete flow of logic in two parts. Diagrams below may use some reference to the files that are going to be discussed in the Implementation Approach.

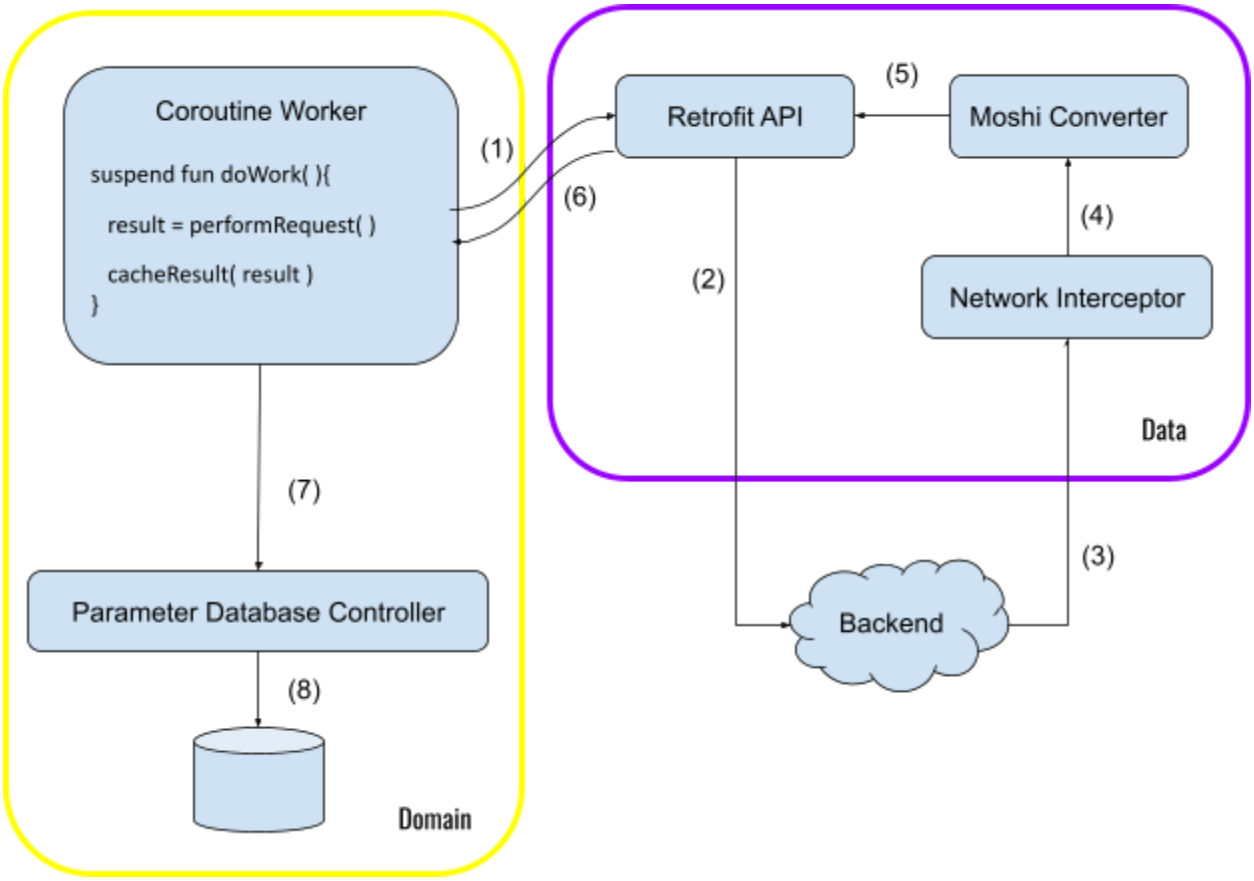
A. Logic for Loading Parameters from Cache



Explanation of each Step -

1. App launches, and by default, the Splash Screen gets opened, which requires Splash Activity to work, which in turn require SplashActivityPresenter to handleOnCreate( )
2. SplashActivityPresenter shows the Splash Screen and blocks the App's UI to fetch the Parameter values from the Cache-Store with ParameterDatabaseController.
3. ParameterDatabaseController reads the "platform_parameter.cache" file with the help of Persistent Cache Store
4. The values are received from the cache-store and then transformed into a ParameterMap to be easily retrieved.
5. Splash Activity is made aware that cache file has been read (successfully/unsuccessfully is not the concern of app module) and it can continue the App's functioning
6. The ParameterMap received from the database is stored inside the Dagger Singleton to act as a source of parameters for the Dagger module..
7. The dagger module will use this ParameterMap as a source for providing individual parameter values.

B. The Flow of logic for a work manager network request



Explanation of each Step -

1. WorkManager starts executing our Scheduled WorkRequest and triggers the Retrofit API to initiate a network request for fetching the latest parameter values.
2. Retrofit used along with a coroutine dispatcher will perform the task asynchronously. It also sends the build parameters in the request.
3. We get the JSON response containing the list of Platform Parameters. This JSON needs to get passed through the Network Interceptor to remove the XSSI prefix from it.
4. Now the Moshi converter can easily transform this JSON into the Models we define.
5. Retrofit request gets completed and the response is parsed and ready to be sent back to the Coroutine worker
6. doWork( ) function gets the response in the result variable and resumes the further execution.
7. This result gets forwarded to the Parameter Database Controller, which converts it to the proto-model objects.
8. Data finally gets stored in the platform_parameter.cache file, and it will be used in the following App start.

Implementation Approach

We can break the implementation into the following categories.

1.) Generating Proto Models

Model Classes are needed so that the parameter values can be stored and consumed easily. We can do it with the help of protocol buffers, as they are already used in the codebase for creating model classes.

Below are the parts of the exemplar proto file (platform_parameter.proto) to explain the model -

ParameterMap -

- It corresponds to the topmost abstraction of parameter values in the form of a Map<String, Parameter> where the key will correspond to the parameter's name and the value will correspond to the **Parameter** type objects (explained below), which will store the parameter's value.
- After reading data from the cache, we will transform the list of parameter values into a ParameterMap, stored as a dagger singleton.
- This Map will be provided as a dependency to the dagger module to read the values from it while providing individual parameters to the required place.

```
syntax = "proto3";

package model;

option java_package = "org.oppia.android.app.model";
option java_multiple_files = true;


/*
 Storage format of the parameters in runtime. Dagger
 modules will use this map to get the values
*/

message ParameterMap{
  map<string,Parameter> parameter = 1;
}
```

Parameter Map

```
/*
 Final storage format of parameters in the disk. This
 is made in the form of list so as to closely resemble
 the JSON response in cache.
*/

message ParameterDatabase {
  repeated ParameterWithName parameterWithName = 1;
}

message ParameterWithName {
  /* Parameter Names*/
  string name = 1;
  /* Parameter Value*/
  Parameter parameter = 2;
}
```

Parameter Database and ParameterWithName

Parameter Database -

- It is the final format in which the app will cache the parameters into the memory with Persistent-Cache-Store.
- It closely resembles the JSON response. Hence, using this saves us from changing the cache structure later in the future, even if we wanted to consume parameter values in any other format than a **ParameterMap.**
- After reading the cache from memory in the **ParameterDatabase** format, it will get transformed to **ParameterMap** before providing it to the Dagger Singleton.

ParameterWithName -

- This class just helps the Parameter Database to store list of Pairs
- The pair will consist of the Parameter name name and its value

```
/*
 It will correspond to a single Platform Parameter
 which can be stored and provided by our classes
*/
message Parameter {
  /*
    Value of this parameter which can be of
    any one of the following type
  */
  oneof value{

    /* Boolean type parameter value */
    bool boolean = 3;

    /* Integer type parameter value */
    int32 integer = 2;

    /* String type parameter value */
    string string = 4;
  }
}
```

Parameter

Parameter -

- It will represent a single Platform Parameter which is the most basic block of the complete implementation
- Dagger modules will provide the parameters as boolean, string or integer based on the individual **Parameter** object stored in the Map
- Every **Parameter** object will only have one type of value assigned to it. It can be of any kind between a boolean, integer or a string.

2.) Preparing Cache Store

We will create a **ParameterDatabaseController** that will enable us to access and update the cached parameter values with Persistent Cache Store. We already have specific controllers which serve this functionality; hence we can follow the same structure.

The basic outline of **ParameterDatabaseController** will be as follows

```
private const val PARAMETER_VALUES_DATA_PROVIDER_ID =
  "parameter_values_data_provider_id"

/* Controller for accessing the platform prameters */

@Singleton
class ParameterDatabaseController @Inject constructor(
  cacheStoreFactory: PersistentCacheStore.Factory,
  consoleLogger: ConsoleLogger,
  ...
) {

  private val parameterDatabaseStore =
    cacheStoreFactory.create(
      "parameter_database",
      ParameterDatabase.getDefaultInstance()
    )
  ...
}
```

ParameterDatabaseController

Parameter Database Controller -

- It will be present in the domain layer and will be able to access the "**parameter_database.cache**" file if present on disk (which will get created after a successful network request)
- It will have a method to read the cache-store and store the parameter values as a **ParameterMap**; the function will wrap this **ParameterMap** into a DataProvider before returning it. This data provider object will get converted into a LiveData to observe it for changes in the app module. This is needed to tell the app module that the parameter values have been loaded and it can function normally.
- Similarly, we will have a method to set the values into the cache-store. This method will take the parameter in the form of a List (or anything else in which we parse the response) and then convert it into a **ParameterDatabase** type object before storing it into the cache-store.

```
private val parameterDatabaseDataProvider by lazy {
parameterDatabaseStore.transform(
  PARAMETER_VALUES_DATA_PROVIDER_ID
){
    parameterDatabase ->
  val parameterMap = ParameterMap.newBuilder()
  val size = parameterDatabase.parameterCount;
  for (i in 0..size){
    parameterMap.putParameter(
        parameterDatabase.getName(i),
        parameterDatabase.getParameter(i)
      )
    }
    parameterMap.build()
  }
}

/*
 Returns a [DataProvider] containing ParameterMap
*/
fun getParameterDatabase()
  : DataProvider<ParameterMap>
  = parameterDatabaseDataProvider
```

Getter method for DataProvider of ParameterMap

```
/*
 Saves the list of Parameters, retrieved after
 parsing the response from the Oppia backend
*/
fun updateParameterDatabase(
  parameterList : List<ParameterWithName>
) {
  parameterDatabaseStore.storeDataAsync(
    updateInMemoryCache = false
  ) {
    it.toBuilder()
      .addAllParameter(parameterList)
      .build()
  }.invokeOnCompletion {
    it?.let {
      consoleLogger.e(
        "DOMAIN",
        "Failed when storing parameter values list",
        it
      )
    }
  }
}
```

Setter method for Caching the parsed Network response

3.) Dagger Dependency Setup

We will use Dagger for Injecting parameter values into the files which need them. For this we need to create a Dagger Module which will contain all the @Provides annotated methods that will make individual parameters available for injection. Also we will need to create a Singleton for storing the **ParameterMap** object at runtime; so that it can work like a source containing all **Parameter** objects mapped to respective parameter names. Here is the distribution of complete Dagger setup -

Parameter Singleton

```
/*
 Source for storing parameters at runtime as
 a singleton
*/
@Singleton
class ParameterSingleton @Inject constructor(){

 /* ParameterMap (proto class object) */
  var parameterMapObject : ParameterMap? = null

 /*
  Setting the parameterMapObject for the very
  first time only
 */
  fun setParam(value : ParameterMap) {
    if(parameterMapObject==null) {
      parameterMapObject = value
    }
  }

 /*
  Getting Individual Parameters(if exists) from the
  parameterMap that is inside parameterMapObject
  */
  fun getParam(parameterName: String): Any? {
    return parameterMapObject
           ?.parameterMap[parameterName]
  }
}
```

Parameter Singleton -

- This is Dagger Singleton class which is going to store the ParameterMap for the complete lifetime of the App
- It will have a simple getter and setter method for **parameterMapObject (proto class object of ParameterMap)** so that the Parameter Values after being read from the Cache store can be stored here.

Platform Parameter Module

```
/*
 Dagger module to provide the individual parameters
 as an implementation of PlatformParameter Interface
*/
@Module
class PlatformParameterModule {

  @Provides
  @Singleton
  @Named(ExampleParameterName)
  fun provideExampleParameter (
    singleton: ParameterSingleton
  ): PlatformParameter {
    return object : PlatformParameter {
      override val value: Any
        get() = singleton
                .getParam(ExampleParameterName)
                ?:ExampleParameterDefault
    }
  }
  ...
}
```

Platform Parameter Module -

- Here we are providing individual Parameters that are wrapped inside an interface (PlatformParameter).
- We fetch the fresh parameters values from the ParameterMap with the help of the Dagger Singleton that we created earlier. In case we fail to find the fresh value inside the ParameterMap then we use a default value which will be hard-coded as shown.
- The Name and Default Values of all the Parameters will be stored as constants in a separate file named as ParameterConstants

Platform Parameter

```
/*
 Interface which wraps a Parameter Value before
 injecting them in the required files
*/
interface PlatformParameter {
  val value : Any?
}
```

Platform Parameter -

- Every Individual Parameter will be provided as an implementation of this Interface which can store any type of value. (in our case Boolean, Integer Or String)
- We will differentiate between two types of implementations with the help of a Qualifier annotation (@Named).
- This annotation requires a Unique String so as to have a differentiating property, hence we will use the Parameter Names which are unique for every Parameter

Parameter Constants

```
/* Names of all the Platform Parameters */
const val ExampleParameterName = "example_parameter"
...

/* Default Values of all the Platform Parameters */
const val ExampleParameterDefault = "example_parameter_default_value"
...
```

Parameter Constants -

- This File will store the names and default values of Individual Platform Parameters.
- While providing individual Parameters, we are checking if they exist in the source or not. And if they do not exist in the source we will provide the hardcoded default values

4.)Using them in App

To use this Dagger setup in the App we will have to make the Parameter Module globally visible in the Dagger Graph. For this purpose we will need to include the **Parameter Module** in the Application Component

```
/*
  Root Dagger component for the application. All
  application-scoped modules should be included in
  this component.
 */
@Singleton
@Component(
  modules = [
    ...,
    ParameterModule::class
  ]
)
interface ApplicationComponent : ApplicationInjector {
  @Component.Builder
  interface Builder {
   ...
  }
  ...
}
```

After including ParameterModule we only need to inject individual Platform Parameters at the required places. As every Parameter is provided as an implementation of interface (discussed above) we will cast its value to the type in which we want to use. We will differentiate between the different Parameters with the help of @Named Annotation. This will be done by using the Name of the parameter we want, so that we receive that specific implementation. Name of Parameters will be maintained inside a ParameterConstants file

```
/*
 Injecting Example Parameter that is wrapped inside
 an PlatformParameter Interface.
*/
class ExampleClass @Inject constructor(
  @Named(ExampleParameter)
  private val exampleParameter: PlatformParameter,
) {
  fun exampleFunc() {
    val paramValue = exampleParameter.value as String
    ...
  }
}
```

4.) Preparing the Network Layer

We will need a new API which will point to the endpoint for fetching the Platform Parameters from the Oppia Backend. Also we will write new Model classes based on the response structure.

Currently the Oppia Backend has this endpoint which gives the JSON response of the following structure. Only a (dummy_feature is currently available)

Endpoint -> https://oppia.org/platform_features_evaluation_handler

Response ->

```
)]}'
{"dummy_feature": false}
```

Therefore for our project we will fake a network response every time, and follow a simple JSON structure for the proof of concept. Later on this can be directed to the functional endpoint and a new JSON structure if needed. (Future works)

Response structure for using in this project will be as follows->

```
)]}'
{
"platform_parameters" : [
        {
                "parameter_name" :"param1",
                "parameter_value": true
        },
        {
                "parameter_name":"param2",
                "parameter_value": "allow"
        },
        ...
]
}
```

```
/*
 Service that provides access to platform
 parameter endpoint.
*/
interface PlatformParameterService {

  @GET("demo_endpoint_platform_parameter")
  fun getParametersByVersion(
    ...
    @Query("app_version") version: String

  ): Call<GaePlatformParameters>
}
```

PlatformParameterService

Platform Parameter Service -

- New API for fetching the Platform Parameters
- We will provide this interface from the Network Module as a dependency

```
/*
  Provides the Platform Parameter Service
  implementation.
*/
@Provides
@Singleton
fun providePlatformParameteService(
  @OppiaRetrofit retrofit: Retrofit
): PlatformParameterService {
  return retrofit.create(
    PlatformParameterService::class.java
  )
}
```

This is how it will be provided by Network Module as a dependency.

Data classes which will help with Moshi to parse the raw response will be like this

```
/**
 Data class for List of PlatformParameter
 */
@JsonClass(generateAdapter = true)
data class GaePlatformParameters(

  @Json(name = "platform_parameters")
  val platformParameters: List<GaePlatformParameter>?

)
```

GaePlatformParameters

GaePlatformParameters -

- It will be similarly structured to the way we receive the JSON response. For our simplicity we assumed the response to be in the structure of a List of GaePlatformParameter (explained below)

```
/**
 Data class for a PlatformParameter
 */
@JsonClass(generateAdapter = true)
data class GaePlatformParameter(

  @Json(name = "parameter_name")
  val parameterName: String,

  @Json(name = "parameter_value")
  val parameterValue: Any

)
```

GaePlatformParameter

GaePlatformParameter -

- It will represent a single Parameter which we receive in the JSON response.

- It contains two fields, one will be assigned to parameter name and other to be a value of Any type (as parameter value can be of any type)
- For simplicity we assume a JSON as shown above. This can change based on the response structure at the time of implementation

5.) Work Manager Setup

We need a work manager to automate the fetching of parameter values from the backend and also to cache the response received. We already use one such Work manager to upload Logs to the Firebase console. We can follow a similar architecture for our case with the only difference that. For uploading logs we have to read the cache and then make a network request. But in our case the opposite path will be taken.

First we will make a PlatformParameterWorkerModule that will be included in the Application Component (as a Dagger Module).

```
/*
 Provides [PlatformParameterWorker] related
 dependencies.
*/
@Module
interface PlatformParameterWorkerModule {

  @Binds
  @IntoSet
  fun bindPlatformParameterWorkRequest(
    platformParameterWorkManagerInitializer :
    PlatformParameterWorkManagerInitializer
  ): ApplicationStartupListener
}
```

PlatformParameterWorkerModule

Platform Parameter Worker Module -

- It will provide a PlatformParameterWorkManagerInitializer (explained below). This Work Initializer class implements ApplicationStartupListener.

Work Manager Configuration Module -

- This will provide a new Work Manager configuration to our App that can include PlatformParameterWorkerFactory also (used to initialise Platform Parameter Worker).
- DelegatingWorkingFactory is just a helper class by androidx.work to associate more than one WorkerFactory implementations. This is needed to support the addition of a new PlatformParameterWorker

```
/* Provides Configuration for the work manager */
@Module
class WorkManagerConfigurationModule {

  @Singleton
  @Provides
  fun provideWorkManagerConfiguration(
    logUploadWorkerFactory: LogUploadWorkerFactory,
    platformParameterWorkerFactory : PlatformParameterWorkerFactory
  ): Configuration {

    val workerFactory = DelegatingWorkingFactory()
     workerFactory.addFactory(logUploadWorkerFactory)
     workerFactory.addFactory(platformParameterWorkerFactory)

    return Configuration.Builder().setWorkerFactory(workerFactory).build()
  }
}
```

WorkManagerConfigurationModule

Platform Parameter Worker -

- This is an implementation of Coroutine Worker for fetching and caching the platform parameters.

- In the doWork( ) function we will call the refreshParameters ( ) suspend function in order to start the network request.

- Logic of refreshParameters( ) -
  - Makes a network request with Network Module
  - Wait for response to be received in the form of GaePlatformParameters
  - Transform the response to the MutableList<PlatformWithName>
  - Send this List to ParameterDatabaseController for caching

```
/*
 Worker class that extracts the parameters from remote
 service and saves the response to the cache store
*/
class PlatformParameterWorker private constructor(
  context: Context,
  params: WorkerParameters,
  private val parameterDatabaseController: ParameterDatabaseController,
  private val platformParameterService: PlatformParameterService,
  private val consoleLogger: ConsoleLogger,
  @BackgroundDispatcher private val backgroundDispatcher: CoroutineDispatcher
) : CoroutineWorker(context, params) {

  override suspend fun doWork(): Result {
    return when (inputData.getString(WORKER_KEY) {
      PLATFORM_PARAMETER_WORKER -> {
        withContext(backgroundDispatcher) { refreshParameters() }
      }
      else -> Result.failure()
    }
  }
}
```

PlatformParameterWorker

```
private suspend fun refreshParameters(): Result {
  return try {
    val result = performNetworkRequest()
    val parameterWithNames
    : MutableList<ParameterWithname> = mutableListOf()
    result.let {
      it.platformParameters?.forEach{ parameter->
        val paramValue = parameter.parameterValue
        val paramName = parameter.parameterName
        val parameterWithName = ParameterWithName.newBuilder()
          .setName(paramName)
          .setValue(paramValue)
        parameterWithNames.add(parameterWithName)
      }
    }
    parameterDatabaseController.updateParameterDatabase(parameterWithNames)
    Result.success()
  } catch (e: Exception) {
    consoleLogger.e(TAG, "Failed to fetch the Platform Parameters", e)
    Result.failure()
  }
}
```

Suspend function for triggering the network request
(refreshParameters)

```
@Singleton
class PlatformParameterWorkManagerInitializer
@Inject constructor(
  private val context: Context,
) : ApplicationStartupListener {

  private val
  workRequestForPlatformParameters: PeriodicWorkRequest
  = PeriodicWorkRequest
    .Builder(PlatformParameterWorker::class.java, 1, TimeUnit.DAYS)
    .setInputData(inputDataForWorker)
    .setConstraints(workerConstraintsForWorker)
    .build()
  }

  override fun onCreate() {
  val workManager = WorkManager.getInstance(context)
  workManager.enqueueUniquePeriodicWork(
      OPPIA_PARAMETER_WORK,
      ExistingPeriodicWorkPolicy.KEEP,
      workRequestForPlatformParameters
    )
  }
}
```

PlatformParameterWorkManagerInitializer

Platform Parameter Work Manager Initializer -

- Used to Initialize the Work Manager for refreshingPlatform Parameters
- This will enqueue a new periodic work request to the WorkManager
- It implements an ApplicationStartStateLitener hence it will begin its processing as soon as the onCreate ( ) of Oppia Application is called.

6.) Checking the Implementation with some Exemplar Parameters

We will check the working of our implementation with the hardcoded values of the Platform Constants (using Default Parameter Map).

We will make a flag which can enable/disable the display of a Welcome Toast message every time the App starts. For this we will introduce a **welcome_user** boolean parameter locally in the Parameter Constants File.

We will also set up a fake network response containing welcome_user parameter. This is done because any changes to Oppia Web are not included in this project timeline.

welcome_user = false

welcome_user = true

Third-party Libraries

No, this Project does not require any new Third Party Library other than the ones which are already being used in Oppia-Android

Testing Approach

As we do not perform any UI changes hence we will only need Unit Testing for this project (i.e. Tests which run on Robolectric). Every Pull request will have all the basic tests which are needed for it to work properly.

We will test the Dagger architecture with the help of Default Value Parameters which will be hardcoded in the Parameter Constants

Testing the fetching from a network can be done by Faking a Response with the help of Retrofit. Also the Work Manager implementations will be tested on the same grounds as we did for Analytics Log Uploader Work Manager.

For Testing this implementation we will need to mock some functionality so that it is easy to test them. Mainly we will need to fake these three things -

1. Platform Parameters provided via Dagger

   We will need to set up a mock source of parameters and fake the *provides* methods for them too.

Fake Parameter Singleton

```
/*
 Source for storing fake parameters at runtime as
 a singleton
*/
@Singleton
class FakeParameterSingleton @Inject constructor() {

  /* ParameterMap (proto class object) */
  var parameterMapObject = ParameterMap
                           .newBuilder()
                           .putParameters(...)
                           .build()
  /*
   Getting Individual Parameters(if exists) from the
   parameterMap that is inside parameterMapObject
   */
  fun getParam(paramName: String): Any? {
    val param :Parameter? = parameterMapObject
                            .parameterMap[paramName]
    return param?.let {
      when(it.valueCase){
        Parameter.ValueCase.BOOLEAN -> it.boolean
        Parameter.ValueCase.INTEGER -> it.integer
        Parameter.ValueCase.STRING -> it.string
        else -> null
      }
    }
  }
}
```

Fake Parameter Singleton -

- This class will act as a singleton source of fake parameters for testing purposes.
- The only difference between this Fake and Actual Singleton class is of the hardcoded parameterMapObject. Doing so will help us to manage what all Parameters are available for testing

Fake Parameter Module

```
/*
 Dagger module to provide the fake parameters as
 an implementation of PlatformParameter Interface
*/
@Module
class FakeParameterModule {

  @Provides
  @Singleton
  @Named(WrongExampleParameterName)
  fun provideDefaultExampleParameterForWrongName (
    singleton: FakeParameterSingleton
  ): PlatformParameter {
    return object : PlatformParameter {
      override val value: Any
        get() = singleton
          .getParam(WrongExampleParameterName)
          ?:ExampleParameterDefaultValue
    }
  }
  ...
}
```

Fake Parameter Module -

- This Dagger Module will provide us the Individual Parameters that are available for testing. Hence it will be only included in PlatformParameterTest (explained below).
- Let's understand the example usage here which is as follows.
    - We know if we are not able to find a Parameter in the ParameterMap then in that case a default value must be provided. This can be due to many reasons and one of them is using an incorrect parameter name
    - If we used an Incorrect Name for Fetching a parameter then we won't be able to find it in the ParameterMap and we will get a Null value from the getParam( ) method. In this case we provide the default value of the Parameter
    - Hence this Dagger Module helps in testing the Logic for providing Parameters

In the end we can write simple Unit Tests which uses these Fakes to test the Dagger Setup like this

Platform Parameter Test

```
/** Tests for PlatformParameters */
@RunWith(AndroidJUnit4::class)
@LooperMode(...)
@Config(...)
class PlatformParameterTest {

  @Inject
  @Named(WrongExampleParameterName)
  lateinit var ExampleParameterWithWrongName : PlatformParameter

  @Inject
  @Named(ExampleParameterName)
  lateinit var ExampleParameterWithRightName : PlatformParameter

  @Test
  fun testPlatformParameter_returnsDefaultIfNotFound() {
    val value = ExampleParameterWithWrongName.value as String
    assertThat(value).isEqualTo(ExampleParameterDefault)
  }

  @Test
  fun testPlatformParameter_returnsValueIfFound() {
    val value = ExampleParameterWithRightName.value as String
    assertThat(value).isEqualTo(ExampleParameterValue)
  }
  ...
}
```

Platform Parameter Test -

- This File will contain all the Unit Tests that are related to Platform Parameter provided by the Dagger.
- In this example here we are checking whether we get a default value for the case when we cannot find the Parameter in the Map (as discussed in Fake Module section). Similarly we are checking for the true value when we can find the Parameter.

2. Network Response from Backend

   We will need to set up a mock network request and a fake response to test the complete flow of fetching the Parameters from the Backend.

   First of all we will prepare a JSON file that contains a fake response according to the GaeModels we have made.

PlatformParameter.json

```
{
  "platform_parameters" : [
    {
      "parameter_name" : "param1",
      "parameter_value":  true
    },
    {
      "parameter_name" : "param2",
      "parameter_value": "orange"
    },
    {
      "parameter_name" : "param3",
      "parameter_value":  0
    }
  ]
}
```

platform_parameter.json -

- This JSON file will store a fake response according to Models we prepared. And it will be kept in the asset directory (data module) along with other JSON files.
- Basically it will contain a List of individual Platform Parameters which have two properties. First is their name(String) and second is their value (Bool/String/Integer)

Note -

This JSON structure is according to these two model classes - **GaePlatformParameter** and **GaePlatformParameters**. Here a single pair of "*parameter_name"* and "*parameter_value"* is according to GaePlatformParameter and the list of these values is defined as an instance of GaePlatformParameters.

The **PlatformParameterService** is an Interface which contains the method for interacting with the backend to return the Parameter values in the form a JSON response that actually is in the form of GaePlatformParameters.

Now for testing purposes we will fake the behavior of Platform Parameter Service by using the MockRetrofit class. Along with MockRetrofit we will take the help of Moshi to parse this Json file to POJO (Plain Old Java Objects).

Mock Platform Parameter Service

```
/*
  Mock PlatformParameterService with dummy data from
  [platform_parameter.json]
*/
class MockPlatformParameterService(
  private val delegate: BehaviorDelegate<PlatformParameterService>
  ) : PlatformParameterService {

  override fun getParametersByVersion(
    version: String
  ): Call<GaePlatformParameters> {
    val parameters = createMockGaePlatformParameters()
    return delegate.returningResponse(parameters)
                   .getParametersByVersion(version)
  }
  ...
}
```

Mock Platform Parameter Service -

- This class implements the Platform Parameter Service and overrides the getParameterByVersion( ) method which takes the version of the App and returns the GaePlatformParameters.
- Here we use a method named as createMockGaePlatformParameters( ). This particular method is responsible to manage the reading and parsing of the dummy json that we created earlier. We add XSSI prefix to the dummy response so as to mock the form of response we will actually get from the backend.

```
/*
  This function creates a mock GaePlatformParameters with data from dummy json.
*/
private fun createMockGaePlatformParameters(): GaePlatformParameters {
  val networkInterceptor = NetworkInterceptor()

  var parameterResponseWithXssiPrefix =
  NetworkSettings.XSSI_PREFIX + ApiMockLoader.getFakeJson("platform_parameter.json")

  parameterResponseWithXssiPrefix =
  networkInterceptor.removeXssiPrefix(parameterResponseWithXssiPrefix)

  val moshi = Moshi.Builder().build()

  val adapter: JsonAdapter<GaePlatformParameters> =
            moshi.adapter(GaePlatformParameters::class.java)

  val mockGaeParameters = adapter.fromJson(parameterResponseWithXssiPrefix)
  return mockGaeParameters!!
}
```

Now we can test this complete network response flow via Unit Tests.

Platform Parameter Service Test

```
/*
 Test for [PlatformParameterService] retrofit instance using the
 [MockPlatformParameterService]
*/
@RunWith(AndroidJUnit4::class)
@LooperMode(...)
class PlatformParameterServiceTest {
  private lateinit var mockRetrofit: MockRetrofit
  private lateinit var retrofit: Retrofit
  ...

  @Test
  fun testPlatformParameterService_usingFakeJson_deserializationSuccessful() {
    val delegate = mockRetrofit.create(PlatformParameterService::class.java)
    val mockPlatformParameterService = MockPlatformParameterService(delegate)

    val parameter = mockPlatformParameterService.getParametersByVersion("1.0")
    val parameterResponse = parameter.execute()

    assertThat(parameterResponse.isSuccessful).isTrue()
    assertThat(
      parameterResponse.body()!!.platformParameters?.get(0)?.parameterName
    ).isEqualTo("param1")
  }
```

Platform Parameter Service Test -

- This file will contain the Unit Tests which are needed to test the Network Request flow
- In this test we actually check whether the response was successfully read and parsed.

3. Work Manager Implementation

For the case of Platform Parameters, we will use a Work Manager for performing two major tasks. First task is to make a network request for fetching new Parameters and second is to cache these Parameters into the cache-store. For testing purposes we only need to fake a network response and not a cache-store. This is because the cache process can be easily tested with Mockito without any fakes.

We will fake a network response in the similar way that we defined in the previous section, ie. with the help of MockRetrofit. Previously we discussed about the Platform-Parameter-Worker (part of Technical Design pg-30) where the doWork( ) function was calling another suspend function named as refreshParameters( ). In this method we use the PlatformParameterService to make a network request. Hence for testing purposes we will use MockPlatformParameterService instead. This will enable us to get the Fake Network Response (stored inside dummy json).

A Sample Test which enques a One Time Work Request for fetching the Platform Parameters and then checks that the parameter is stored in the Cache-store or not.

PlatformParameterWorkManagerTest

```
/*
 Test for [PlatformParameterWorker] instance using the [ParameterDatabaseController]
 & [MockPlatformParameterService]
*/
@RunWith(AndroidJUnit4::class)
@LooperMode(...)
class PlatformParameterWorkManagerTest {

  private lateinit var mockRetrofit: MockRetrofit
  private lateinit var parameterController: ParameterDatabaseController

  @Mock
  lateinit var mockOppiaParameterDatabaseObserver: Observer<AsyncResult<ParameterMap>>

  @Captor
  lateinit var oppiaParameterDatabaseResultCaptor: ArgumentCaptor<AsyncResult<ParameterMap>>
  ...

  @Test
  fun testWorkManager_makeWorkRequest_checkTheRetrievedParameter() {
    val workManager = WorkManager.getInstance(ApplicationProvider.getApplicationContext())
    val request: OneTimeWorkRequest = OneTimeWorkRequestBuilder<PlatfromParameterWorker>()
                                      .setInputData(...)
                                      .build()
    workManager.enqueue(request)

    val parameterDatabase = parameterController.getParameterDatabase().toLiveData()

    parameterDatabase.observeForever(
      this.mockOppiaParameterDatabaseObserver
    )

    testCoroutineDispatchers.advanceUntilIdle()

    verify(
      this.mockOppiaParameterDatabaseObserver,
      atLeastOnce()
    ).onChanged(oppiaParameterDatabaseResultCaptor.capture())

    val parameter = oppiaParameterDatabaseResultCaptor
                    .value.getOrThrow().getParameter("param1")

    assertThat(parameter.name).isEqualTo("param1")
    assertThat(parameter.value).isEqualTo(true)
  }
}
```

Note -
The exact code snippet for the using a network API is still under development in Oppia-Android. Therefore we assume that till the start of the Coding period, we will have an exact idea of making a network request too.

Milestones

Milestone 1

Key Objective: We will have a Dagger architecture that is able to provide compile time Platform Parameters which are hard coded. New ParameterDatabaseController which handles the Parameter Database Cache. GaeModels and a dummy json for the purpose of Network Request. Also we will have tests for all the code changes till yet. (ie. Controllers, Dagger Injection, Fake Parameters)

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	New Proto Classes for Cache Storage of Platform Parameters (ie. ParameterMap, ParameterDatabase, ParameterWithName and Parameter)	None	7 June 21	11 June 21
1.2	Parameter Singleton for storing the Values fetched from cache-store at runtime. New Parameter Constants File that contains all the parameter names and default values.	1.1	14 June 21	19 June 21
1.3	Parameter Module for providing Individual parameter values. A Platform Parameter Interface which is the final form of parameter values in which they are consumed by App.	1.2	23 June 21	29 June 21
1.4	Parameter Database Controller in domain layer for managing the Cached Platform Parameters and Initialize Singleton with the cache-store values.	1.2	27 June 21	3 July 21
1.5	Gae Models for representing the network response and a dummy json according to these models.	None	6 July 21	11 July 21

Milestone 2

Key Objective: We will have a complete setup for performing Network Request (ie. GaeModels, Endpoint, Fake Response). A new Work Manager implementation for Platform Parameters (ie. CoroutineWorker, WorkerFactory, WorkManagerConfiguration, Injectable Factories). To show the

usage of this architecture an Example of PlatformParameter Implementation for Welcome User Toast-Message. Also the Tests for the code changes till yet (ie.Work Manager Requests and Fake Network Response)

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	PlatformParameter API for hooking up the backend	1.5	17 July 21	22 July 21
2.2	Platform Parameter Worker and its Worker Factory Classes along with Work Manager Initializer	2.1	24 July 21	30 July 21
2.3	Including Platform Parameter Worker Factory in the Work Manager Configuration Module along with adding Platform Parameter Worker Module in the Application Component	2.2	2 August 21	7 August 21
2.4	Exemplar Usage of Platform Parameter in App with Splash Screen Welcome Toast.	2.3	10 August 21	15 August 21

Optional Sections

Additional Project-Specific Considerations

Privacy

This feature only requires the Build Parameters of the App instance. These are needed to be sent in the request for fetching the required platform parameters. Hence there is no unauthorized User data collection.

Security

This project will involve adding a new endpoint which receives build parameters and returns the Platform Parameters in a JSON response. The only issue that can happen here is that of spamming the endpoint with multiple requests. Other than that user cannot gain any unauthorized access.

Accessibility (if user-facing)

This project will aid the developers of Oppia-Android and doesn't require any UI changes. Therefore we do not need to plan about making it Accessible.

Documentation Changes

We can add these things to Oppia-Android Wiki

- What are Platform Parameters
- How are they used in Oppia-Android
- How to add new parameters locally while developing

Ethics

Using Platform Parameters will speed up the development process in Oppia-Android along with giving us complete control over the App which is released. This project will set up a complete architecture for using Platform Parameters, hence it is going to be benefactory.

Future Work

List of things which will be done after the completion of project -

- We will group all the logic for kicking off the flag-fetch process in functions so that we can incorporate the refreshing flag functionality in the Developer options menu.
    - Vinita Murthi 's Suggestion - It may be worth adding a button that can trigger a sync in the developer options menu
    - Rajat Talesra - Developer options mock link. https://xd.adobe.com/view/e8aa4198-3940-47f9-514a-f41cc54457f6-9e9b/screen/5ced965e-1a0a-48cf-85dd-f28ba68f0b99/

- Sending names of parameters in the network request to the backend, whose values are demanded by App
    - Ben Henning 's suggestion - If it helps, we never remove old flags from the backend. Sending the names of flags should be noted in the future work section as an optimization to prevent unbounded growth in the network response size for flags for older clients.

Additional Questions

The proposal seems to lack a couple of important details:

a) How the compile-time injection will work

b) How the app will resolve having different compile-time parameters from those provided by the backend (and particularly how name clashing might cause issues in such cases in the future)

Please explain your plans for these.

Answer -

From the discussion in the email thread -
Feature flags or config parameters which are available first as local are compile time parameters, before they are added as runtime parameters in the backend.

Inference from this statement -
The value of a Compile-Time Parameter will be hardcoded into the App as a default value for that Parameter. We will rely on default value until the Parameter gets added into the Backend thereby being returned in the network response. Depending on default values as above, allows us to use the same dagger architecture for both compile-time and runtime parameters.

a) Compile-Time Injection

In the end we will be using Dagger for injecting Individual Platform Parameters which can be differentiated on the basis of Qualifier Annotations.

How do we Load Values for injection
To put this in technical words, we will have a Dagger Module that will contain some @Provides annotated methods. These methods actually return a Single Platform Parameter after retrieving it from a source. The Source for these values will be a Parameter Map that will be stored inside a singleton class. This Singleton will get its parameter map from a Database Controller. This Database Controller will receive a trigger to read the parameter values from the cache-store and store the result in the singleton class. The trigger for this read will be made by the Splash Activity Presenter that will start working every time Splash Screen appears (ie. every time app launches)

When do we inject Compile-Time Parameter
If in case we are not able to find a Parameter in the Source, then we will provide a Default value for that Parameter which is hardcoded beforehand. This can happen in two cases that either the loading of cached values failed or the Parameter value which we are looking for is not a runtime parameter (ie not in the cache store and network response). In second scenario we are actually dealing with a compile time parameter and hence the true value of this compile-time parameter is stored as its default value

b) Clashing of Parameter Names

For any version of the App, all the possible cases that can occur while using the Platform Parameters are as follows

- Parameter exist only in the Android client (ie. compile-time) -
  If the parameter only exists in the Android client then we won't receive it in the response from the backend. Hence in this case we will depend on the Default value for that Parameter and no case of name clashing occurs.

- Parameter exist both in the Android client and Backend (ie. runtime) -
  If the parameter exists both in the Android client and the Backend then we will receive it in the response from the Backend (whenever the network request is

successful). Hence in this case we will depend on the received value for the case of successful fetch, and we will use default value for the case of any failure. Here also no case of name clashing occurs.

- Previously, Parameter exist only in the Android Client but now added to the Backend (ie earlier compile-time but now as runtime) -
  In this case our App was earlier depending over the default values of that particular Parameter, but now the App will try to use the updated value of the Parameter that is received in the Network Response. If we desired to update the value of this compile-time parameter then adding it to the Backend will not be a problem here. But if we didn't want to do so, then we will need to block the updated Parameter value to reach the App via Network Response. We can do so by sending the App-version in the Network Request, so that the Backend can be made aware about the App which is requesting for Platform Parameters. Then using this information Backend will send only those Parameters in the response which are allowed to be used with that particular App Version (as decided by the team).

Appendix

This section contains the ideas which went through review and discussions but were not clear enough. It also contains the original review comments.

Below from here are the discussions over Two different Approaches for completing this project. It was reviewed by all three mentors Vinita Murthi, Sarthak Agarwal and Ben Henning. This text will be deleted when all the comments get resolved.

What will the implementation look like?

Approach - 1

We will include a new JSON file named "flags_config.json" in the Oppia-android codebase. This file will have the following features -

- It will be an Android-client copy of the original Platform-Parameter file stored in the backend.
- It will contain the complete set of default flag values and their filters (like locale, build variant, etc.)
- It will be used for all the cases when we won't be able to find any other source of flag values(for, e.g. data wipe-off, first install)

How will we use it

- This file will serve as a central location to toggle any flag values we may need while developing for Oppia-android.
- It will also give us a wholesome view of the features which are currently enabled.
- We will create an Adapter/Parser for this file and provide it with the help of Dagger Modules, which will also take care of filtration before providing the values.
- If we receive a new JSON response from the backend, then we will persist it on the User's device and use this new response to parse the values from.
- This file will be updated when the flag values get changed in the backend after a team discussion.

Probable Issues

- How will this file be updated?
    - We will update this file in two scenarios. First If there is a new implementation whose flag should be added in the list and second when we complete a feature.
    - We will simply run a script to fetch new JSON from the backend and replace the older one, after it is done we can commit this into the repository.
    - As any change of flag values in the backend can only be done by an admin therefore this operation will be performed by an authorised person.
    - ~~As this file is in the .gitignore, we will have to update it manually by removing it from .gitignore--> committing the updated JSON--> including it again in .gitignore~~.
- How many times do we need to parse JSON?
    - Following this approach, we will need to parse the JSON every time the App starts for the first time.
    - (whether it is the statically stored one or it is from the backend response)
- Will Parsing create any Issues?
    - Parsing can take time depending on the file's size; hence we will need to either block further Ui thread until completed or use any other efficient reading method.
- Any other issues?
    - While developing, we will use the complete Android-client copy of the Platform-Parameter file, which needs further filtering based on different rules like locale, build variant etc. But this is not the case with the response we get from the backend. We don't need any filtering on the backend response as it is done by the Controllers there. Hence this will create a bit of inconsistency of not filtering the backend response.

Approach - 2

We will create a new package of Dagger Modules, which will provide the flags of individual features for whom we want a gating mechanism. They will have specific characteristics -

- At first install, we will request to fetch Platform-Parameter JSON file, which will be specific to this device with all the filtering already done on the backend.
- Dagger modules will depend on Datastore / Shared-Preference for getting the new flag values

- In case of any failures in fetching the JSON or any data wipe-off, we will use the default values that will only reside inside the dagger modules.

How will we use it?

- We will depend on the Datastore/Shared-Preference to store the values from the JSON response after reading it with the help of Adapter/Parsers.
- If we don't have any value stored in Datastore/Shared-Preference, we will provide the default values from within the modules. This situation can occur for the case of first install or any failure like data wipe-off or failed fetching.
- Also, we will always reset the values in these Preferences whenever we get a response from the backend.
- We will only parse the response once, in order to store those values in the Preferences and then we can use these flags easily
- As Preferences is a quick method for data retrieval, we will not face any blocking requests. Hence it is reasonable to store the flags in key-value pairs.

Probable Issues

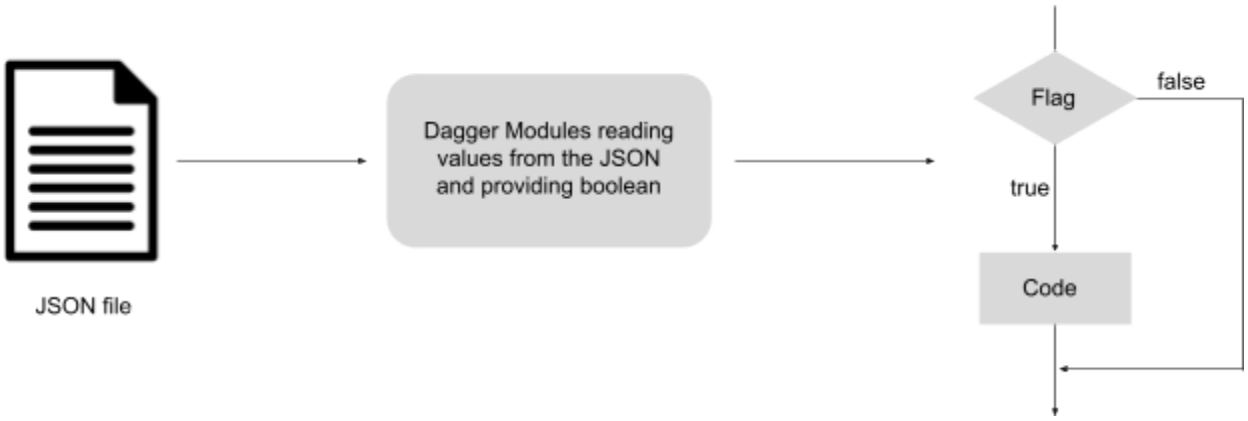
- Will Datastore/Shared-Preference be feasible?
    - As the App can own the data stored in Preferences to be in MODE_PRIVATE, thus making it safe.
    - Also, the limit for storing key-value pairs is reasonable enough for our purpose.
- How will we manage its Lifecycle?
    - Whenever we are fetching a new JSON and storing its values in Preferences, we will create a new Preference Table and delete the previous one only after the successful storage of new values.
    - In this way, we will avoid the use of new fetched values and continue to use them only after they have been successfully updated.
- Any Other Issues?
    - As we will be setting only one default value for each parameter, we won't provide different values based on additional filters like locale. Still, this case is only for the default values during the development period. As we will fetch JSON for a new install and in periodic intervals, we get the correct values.

Kindly ignore the following old idea, it will be kept here until the comments are resolved by murthi.vinita@gmail.com . This approach was very vague and it assumes a JSON structure which is different from the one provided from the backend. But the newer Approaches resembles some of its logic from this idea

I will divide the implementation following the Milestones that are in the Idealist. Hence after the completion of -

Milestone-1

I will be using a simple JSON file containing an array of key-value pairs, which will correspond to a particular flag and its value in boolean, string or integer type. Dagger modules will use this JSON file to provide simple booleans (we will return a boolean for the string/integer values after comparing it with parameters we want); hence this will act as a flag covering out a feature in the if-else statements. I will also add a test in our pre-push checks to prevent a commit that contains a different flag value than the one in static JSON. We will maintain this JSON in our codebase so that there can be a reference to what all features are currently available for use.



Milestone-2

I will complete the setup of fetching a simple JSON file from the backend (with the help of WorkManager and NetworkModule) for the first installation of the app and regularly after a specific interval which the team decides. Dagger modules that we created earlier will read their default values from the static JSON included in the built app and use it only until we get a different response from the backend. That response will get stored in the internal storage of the end-users device where it will be secured too. I will test this setup by enabling/disabling few controls which are given to admin to give the proof of its working