

# GSoC 2021 Proposal

## Implement lightweight checkpointing

### (Apoorv Srivastava)

## About You

Hello Reviewers,

My name is Apoorv Srivastava. I am a sophomore at The LNM Institute of Information Technology, Jaipur. I want to put forward my proposal to implement lightweight checkpointing in the oppia's android application.

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

Oppia's mission is to help anyone learn anything they want in an effective and enjoyable way. With this vision in mind, I wish to contribute to oppia to make learning enjoyable for students all across the world. I also choose Oppia because of the talented and experienced team working at oppia. I believe I will be able to learn a lot by working with them.

## Prior experience

I have good prior working experience. I started contributing to Oppia in October of 2020, in the last 6 months I have fixed a number of issues and filled in new issues which I found during testing. I have also reviewed quite a few PRs so I am familiar with the coding style used at oppia.

I have been doing android development for the past year. Though I started with java, I have been doing it in Kotlin for the past 8 months. I have created a number of personal projects. One of these personal projects was TruCoder which was built entirely in kotlin, which is available on the google [play store](#). I have also interned with MediaVigil as an android developer. Where I had to create an app for their news website.

My contributions at Oppia :

1. Merged PRs:
  - a. <https://github.com/oppia/oppia-android/pull/2670>
  - b. <https://github.com/oppia/oppia-android/pull/2597>
  - c. <https://github.com/oppia/oppia-android/pull/2493>

- d. <https://github.com/oppia/oppia-android/pull/1934>
- e. <https://github.com/oppia/oppia-android/pull/2499>

2. A list of issue that I filled in can be found [here](#)
3. I also reviewed a number of PRs, one of them is for code coverage of the app with JaCoCo. [Here](#) is the link to that PR

## Contact info and timezone(s)

Name: Apoorv Srivastava

University: The LNM Institute of Information Technology, Jaipur.

Country: India

Email: [2000apoorv@gmail.com](mailto:2000apoorv@gmail.com)

Github: <https://www.github.com/MaskedCarrot>

Timezone: Indian Standard Time (IST) (+5:30 GMT)

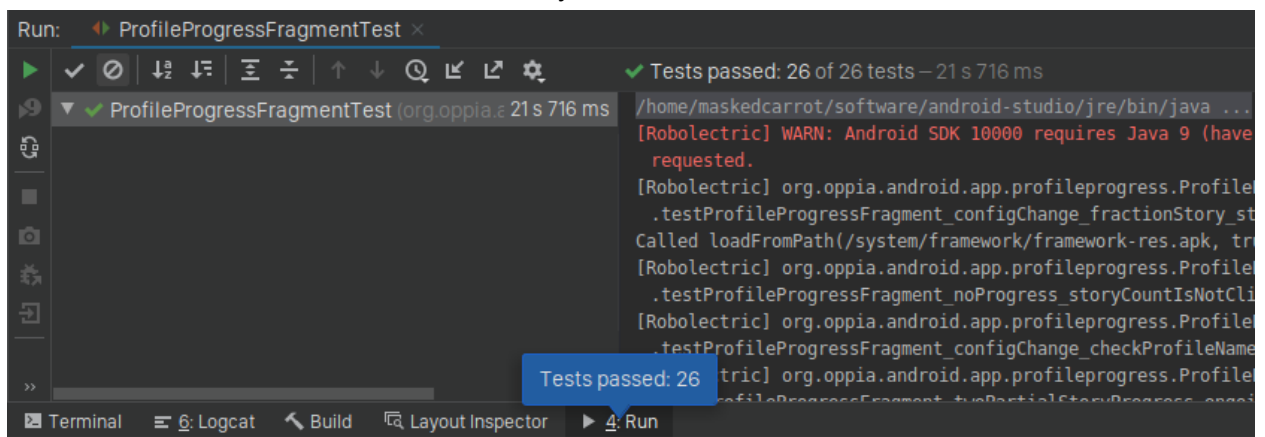
Preferred method of communication: Hangouts, e-mail and Gitter.

## Time commitment

I am committed to spending 7 hours a day on this project on weekdays (Monday to Saturday). The time I devote to the project on Sunday will depend upon the work to be completed in the project in that week. In total, I am committed to working about 45 hours a week.

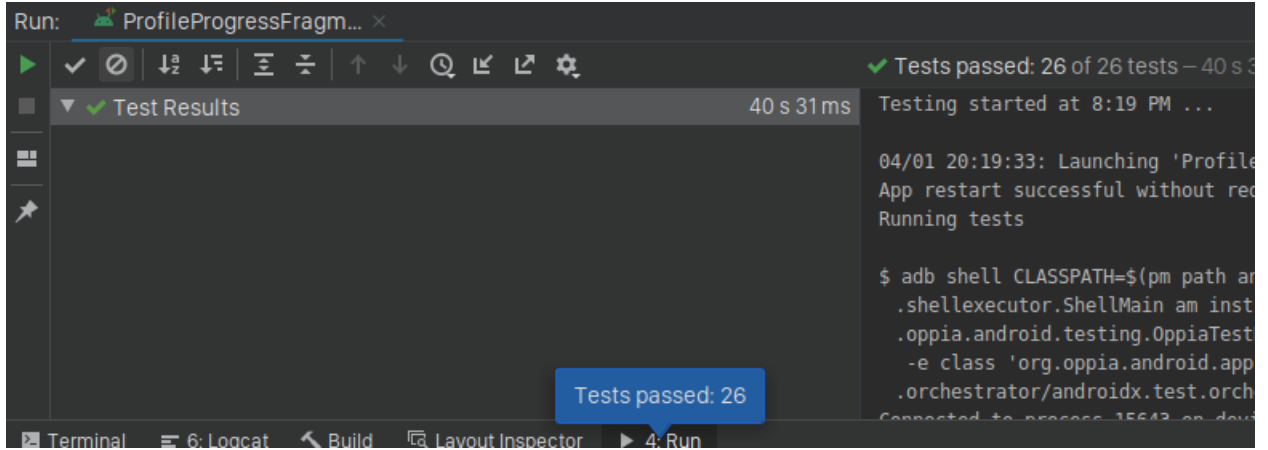
## Essential Prerequisites

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



The screenshot shows the Run console in Android Studio. The top bar indicates 'Run: ProfileProgressFragmentTest'. The console output shows 'Tests passed: 26 of 26 tests - 21 s 716 ms'. Below this, there is a warning from Robolectric: '[Robolectric] WARN: Android SDK 10000 requires Java 9 (have requested.)'. The console also shows several test method names, such as '.testProfileProgressFragment\_configChange\_fractionStory\_st', 'Called loadFromPath(/system/framework/framework-res.apk, tr', '.testProfileProgressFragment\_noProgress\_storyCountIsNotCli', and '.testProfileProgressFragment\_configChange\_checkProfileName'. A blue callout box highlights the text 'Tests passed: 26'.

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



## Other summer obligations

I have no other obligations during this summer.

## Communication channels

I am comfortable with any mode of communication that the mentor chooses, be it email, gitter or hangouts. Mentors can expect a response from me in about an hour.

## Application to other orgs

I am only applying to oppia

---

# Project Details

## Product Design

The users of this feature are the end-user of the oppia-android app. After the implementation of this project, the users will be able to resume the partially complete exploration.

### **What is lightweight checkpointing?**

Lightweight checkpoint saves the user progress, exactly where they left off in an exploration. When the user returns to the exploration, they have the option to start or resume the chapter i.e. the user can then continue the exploration normally as if they never left. This means that the current state and all previous states will have exactly the same form as they did when the user left the exploration.

The progress will be saved locally in the device storage, and therefore it doesn't matter whether the user is online or offline.

### **The general behaviour of lightweight checkpointing**

The following use case defines the behaviour of lightweight checkpointing in various situations.

1. The user advertently or inadvertently leaves a partially complete exploration. In this case, all the progress made by the user in that exploration will be saved. In case the exploration was marked as “not started” the exploration will now be marked as “in progress saved”. In case the progress is not saved (due to low memory, app crash etc) the exploration will be marked as “in progress not saved”.
2. The segmented progress bar in the Topic lessons tab will show a light orange colour arc for all the explorations that were marked “in progress saved” and “in progress not saved”.
3. When the user resumes a partially complete exploration, they will be given the option to start the exploration from the beginning or resume the exploration where they last left it.
4. When the user completes an exploration it will be marked as “Completed”. If the exploration has any progress associated with it in that profile, it will be deleted.
5. When the user restarts an exploration that was marked as “completed”, the exploration will not be marked as “in progress saved” or “in progress not

saved” and will remain as “completed”. Though the progress will still be saved if the user exits the partially complete exploration.

## The general behaviour of UI after implementation of lightweight checkpointing

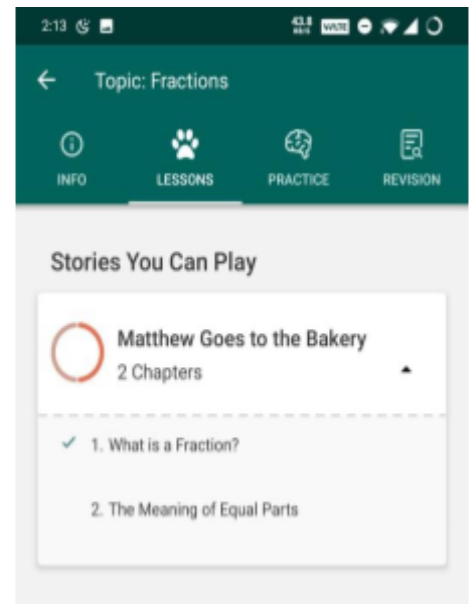
Every exploration that is in progress will be reflected by the UI in the topic lessons tab.

- For every chapter that is in progress, the **segmented circular progress bar** will show an arc of **light orange colour**.
- For every chapter that has been completed the progress bar will show an arc of **dark orange colour**.
- For every chapter that has not been stated yet the progress bar will show an arc in **grey colour**.

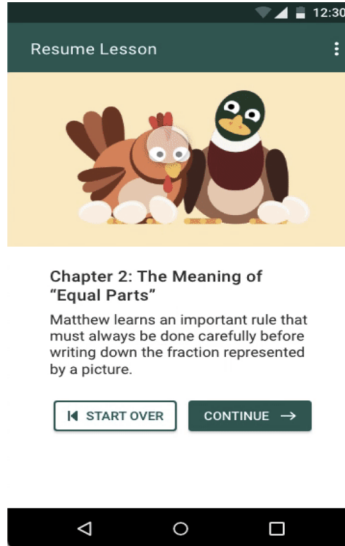
Once the user completes an exploration, the chapter containing that exploration will be marked as “**completed**” and the existing progress if any will be deleted.

When the user starts an exploration that has saved progress associated with it, they will be taken to a new screen like the one shown below. Here they will get an option to restart the exploration or continue the exploration from where they left off.

If the user chooses to continue the exploration, the existing progress will be used to resume the exploration. If the user chooses to start over the exploration, the exploration will start from the beginning and the existing progress will be overwritten.



UI showing “in progress” and “completed” chapters.



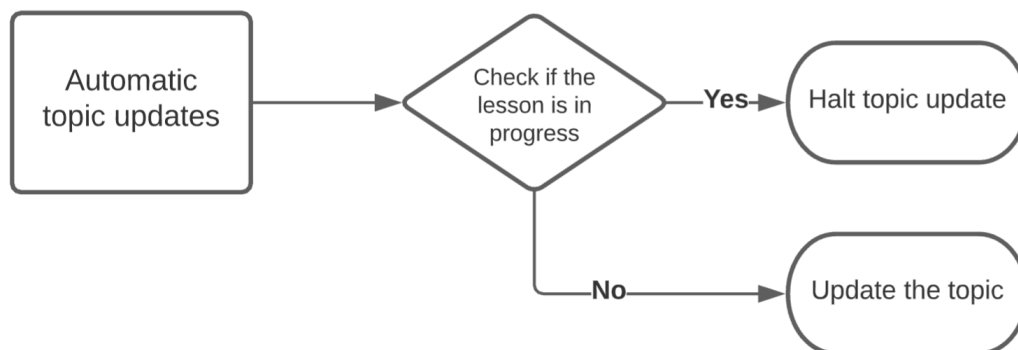
Mock showing the fragment to resume or restart the chapter

## Edge cases and constraints for lightweight checkpointing

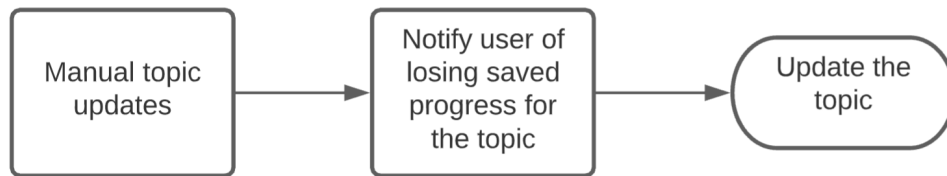
### Topic Updates

Topics are shared across all profiles. If a topic has an “in progress” exploration, the app will not auto-update the topics.

In case the user decides to update the topic manually, they will be informed that “updating the topic will cause all the users of the app on that device to lose progress for that topic”. (since downloaded topics are shared across multiple devices).



If the user still continues to update the topic, the progress for all explorations associated with that topic will be deleted across all profiles. This will be done by checking the version of the exploration downloaded.



### Topic deleted

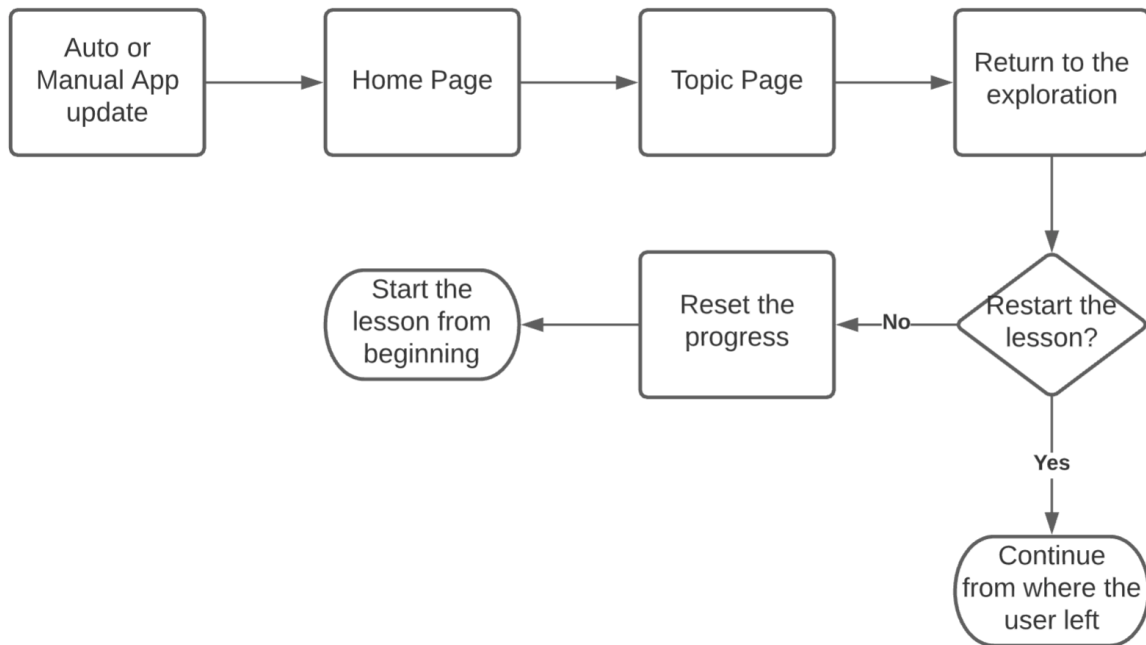
When a user deletes a topic, the topic is deleted across all profiles. So it is important that we don't delete the progress of the exploration associated with that topic.

When the topic is re-downloaded, its explorations **can** be resumed if they have a saved progress stored.

If a topic updates after a user deletes it, a newer version of the topic will be redownloaded. In this case we will use the "exploration version" to figure out if we can resume the exploration with the saved exploration progress or not.

### App is updated

There is no additional requirement in this case. The exploration progress saved in the previous version of the app will be used to resume the exploration in the updated version of the app.



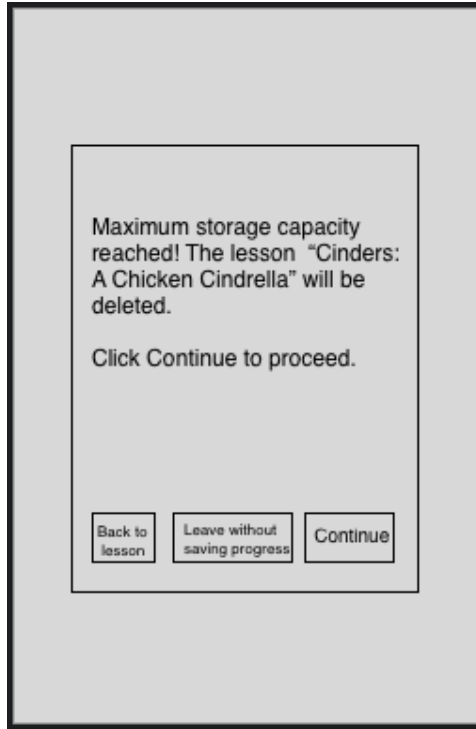
## Storage

Keeping a check on storage is really important. There will be an upper limit of 2 MB per profile to store the checkpoints.

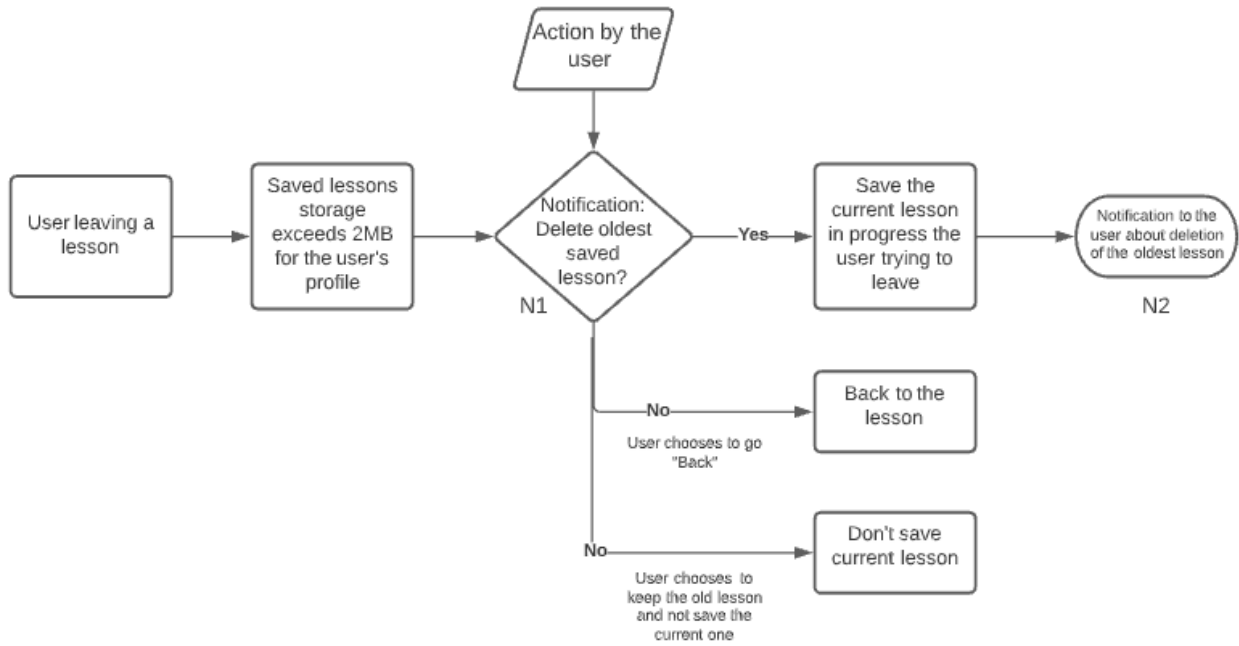
If the storage runs out, a new checkpoint will only be saved if space is freed up by deleting the oldest checkpoint.

It is important to note here that this limit does not guarantee that the checkpoint database will always be under 2MB, however, this does ensure that if the user exits the exploration advertently. The oldest checkpoint will be deleted if the storage limit is reached.





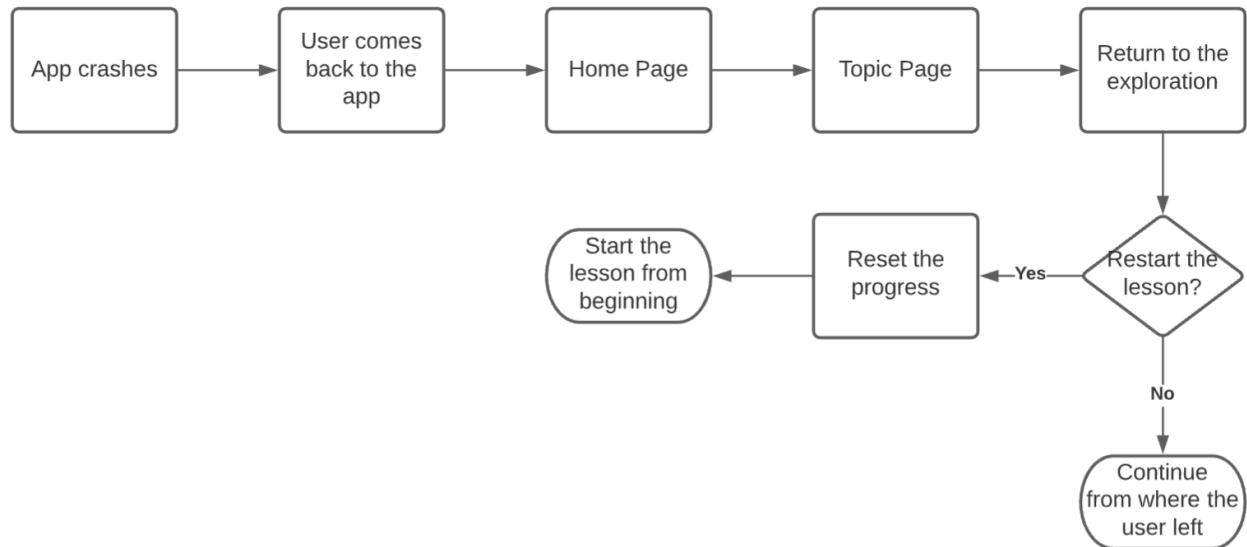
The mock showing dialog box will be shown to the user if the storage runs out.



## App Crashes or is force closed

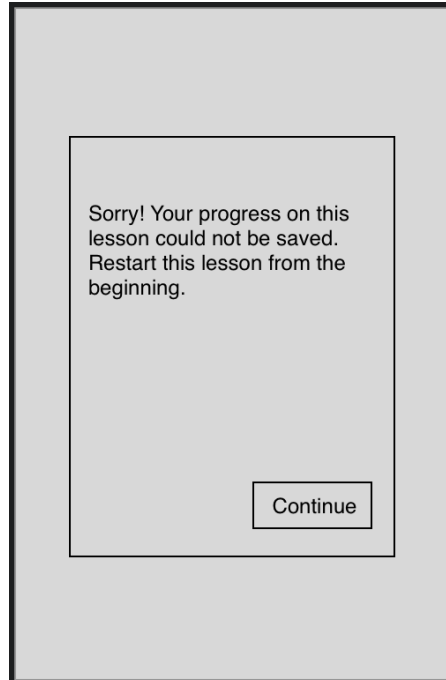
If the app crashes or is force closed (the app is closed by clearing it from recent apps) while the user was in an exploration. It is important that we save the user's progress so that users are able to resume the exploration from where they last left off.

It is important to note that though we have a constraint on storage, we still have to save the exploration progress if the app crashes.



## Progress not saved

If the user exits the exploration inadvertently and the exploration checkpoint storage is full, we will not be able to save the checkpoint progress. In that case, when the user restarts the exploration they will see a dialog box with the following message.



*Mock showing the dialog box which the user sees on restarting the app if progress was not saved on the crash.*

### Space analysis for checkpoint

Early implementation and analysis revealed that each checkpoint takes about **20 KB** of storage. This means we will be able to save around **100 checkpoints per profile** before the storage runs out. This is done by assuming that the user submits 3 incorrect answers before submitting the correct answer in the fourth attempt.

### Metrics

Another important part of building any project are the metrics, which help us measure how the new feature has impacted the user. After implementation of the project, the app will also save and log the following counts

- Number of times progress is saved successfully
- Number of times progress was not saved successfully
- Number of lessons saved advertently
- Number of lessons saved inadvertently
- Number of times user choose 'start over'
- Number of times user choose 'continue'
- Number of correct answers after returning to the lesson
- Number of incorrect answers after returning to the lesson.

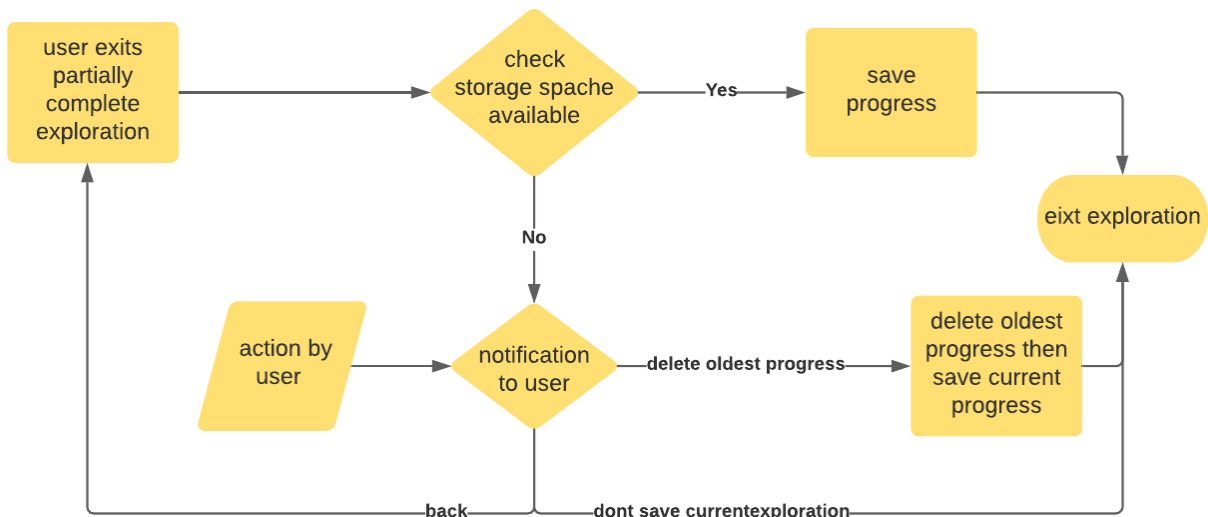
# Technical Design

## The Approach

### Saving the progress

The approach I propose here saves/updates the exploration checkpoint after every time the user interacts with the exploration. This is done to ensure that the progress is saved even in case of system deaths, low memory or app crashes.

In case the user exits the exploration advertently, the size of the checkpoint database will be checked. If the storage size has not exceeded the limit, the user will exit the exploration. If it has, the user will be presented a dialog box where they can choose if they wish to overwrite the oldest saved exploration progress with the current exploration progress.



*Flowchart describing the process to save exploration progress*

In case the user exits the exploration inadvertently, since the progress is saved after every interaction. The user will be able to resume the exploration when they return.

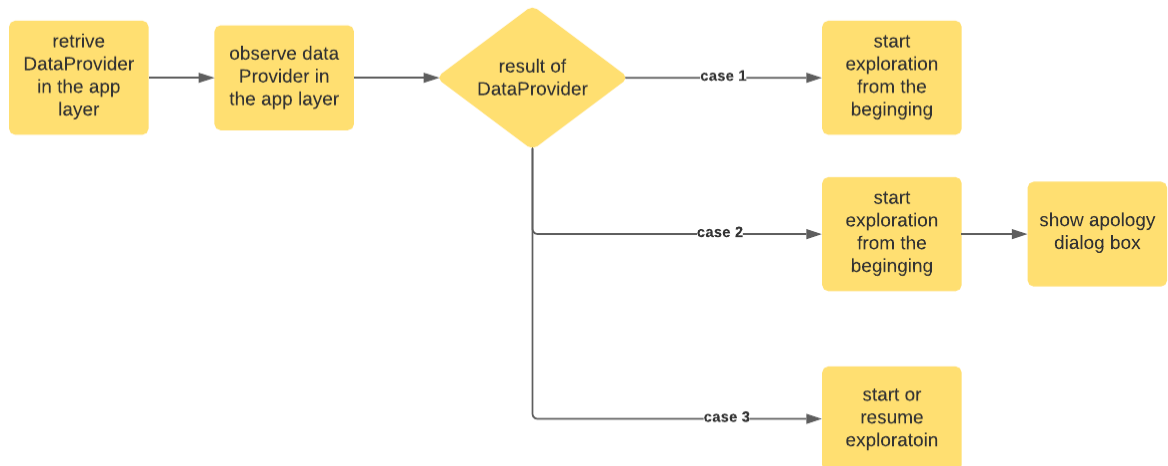
### The process to show an apology dialog when progress is not saved

If an exploration is marked as “in progress not saved”, it means that the user started the exploration but the progress they made was not saved. In this case we will show a dialog box to inform the user that the progress they made in the exploration was not saved.

## Retrieving saved progress

Retrieving the saved progress is quite straight forward, we just have to fetch the DataProvider, observe it for changes. Once ExplorationCheckpoint is available, one of the three processes will be followed.

1. ExplorationCheckpoint is of defaultInstance. This means that no exploration progress was found. In which case we will start the exploration from the beginning.
2. ExplorationCheckpoint is not of defaultInstance but the exploration was marked as “in progress not saved”. This means that the exploration could not be saved and the apology dialog has to be shown.
3. ExplorationCheckpoint is not defaultInstance and the exploration is marked as “in progress saved”. This means that valid progress was found and now we can start to resume the exploration.

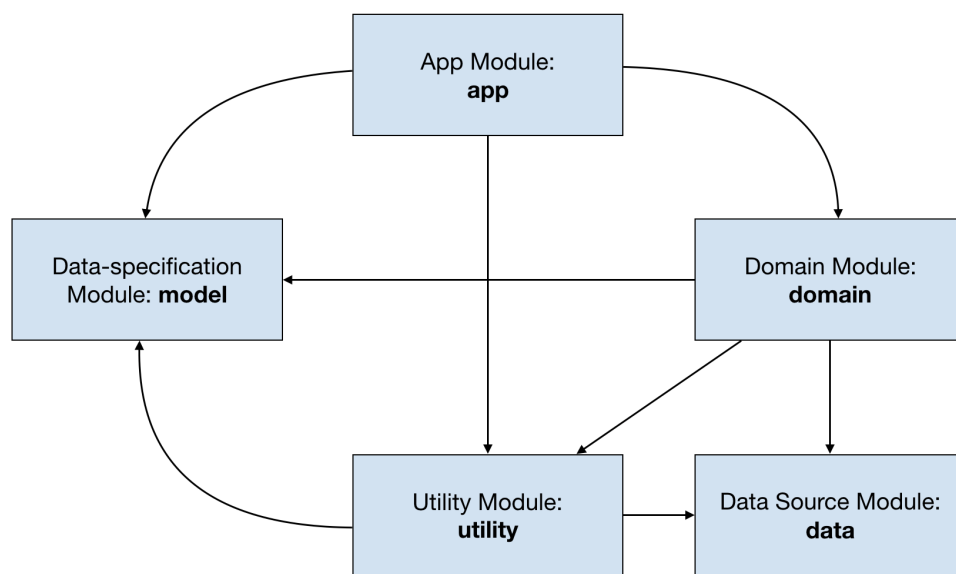


*Flowchart showing the process followed while retrieving saved progress*

## Architectural Overview

Oppia uses a combination of Model-View-ViewModel and Model-View-Presenter architecture in its android application. Further, the oppia-android has its directory structure divided into 5 modules which are as follows:

1. app module contains activities, fragments, ViewModels, presenters. It also contains robolectric tests.
2. data module is used to provide data to the app via backend or local storage.
3. model module contains the protobuf used in the app.
4. domain module contains the business logic used in the app including both frontend controller and business service logic.
5. utility module contains miscellaneous classes like OppiaClock which are used by other modules.



To implement this project we have to use all the modules. The module wise implementation of the project is represented as follows.

1. app module
  - a. Modification of existing presenters and layout files that will show if a lesson is currently in progress.
  - b. Implementation of a new activity, its presenter and its layout file that will allow the users to choose whether they wish to start the lesson from the beginning or they wish to continue where they left.
  - c. Modification of presenters and ViewModel to restart the exploration where it was left off.
2. data module
  - a. Used for creating and retrieving cache storage.

3. domain module
  - a. Addition ExplorationCheckpointController to record and retrieve exploration progress.
  - b. addition of ExplorationStorageModule to inject the constants of time period and storage limit.
  - c. Modification and usage of ExplorationProgressController, ExplorationDataControler and StateDeck to create, record and retrieve exploration checkpoints.
  
4. model module
  - a. Addition of protobuf to store the progress made by the user.
  - b. Modification of oppia\_logger.proto to save metrics.
  
5. utility module.
  - a. DataProviders to retrieve the store.

The following table lists the files that will be added, modified or deleted in each of the modules to implement this project.

#### Model module

exploration_storage.proto	model/src/main/proto/exploration_storage.proto	added
oppia_logger.proto	model/src/main/proto/oppia_logger.proto	modified
topic.proto	model/src/main/proto/topic.proto	modified

#### Domain module

ExplorationCheckpointController.kt	domain/src/main/java/org/oppia/android/domain/exploration/ExplorationCheckpointController.kt	added
ExplorationStorageModule.kt	domain/src/main/java/org/oppia/android/domain/exploration/ExplorationStorageModule.kt	added
StateDeck.kt	domain/src/main/java/org/oppia/android/domain/state/StateDeck.kt	modified

ExplorationProgressController.kt	domain/src/main/java/org/oppia/android/domain/exploration/ExplorationProgressController.kt	modified
ExplorationDataController.kt	domain/src/main/java/org/oppia/android/domain/exploration/ExplorationDataController.kt	modified
AnalyticsController.kt	domain/src/main/java/org/oppia/android/domain/oppialogger/analytics/AnalyticsController.kt	modified

#### App module

StoryChapterSummaryViewModel.kt	app/src/main/java/org/oppia/android/app/story/storyitemviewmodel/StoryChapterSummaryViewModel.kt	modified
TopicLessonFragmentPresenter.kt	app/src/main/java/org/oppia/android/app/topic/lessons/TopicLessonsFragmentPresenter.kt	modified
RecentlyPlayedFragmentPresenter.kt	app/src/main/java/org/oppia/android/app/home/recentlyplayed/RecentlyPlayedFragmentPresenter.kt	modified
ExplorationActivityPresenter.kt	app/src/main/java/org/oppia/android/app/player/exploration/ExplorationActivityPresenter.kt	modified
StateFragmentPresenter.kt	oppia-android/app/src/main/java/org/oppia/android/app/player/state/StateFragmentPresenter.kt	modified
StopExplorationDialogBoxFragment.kt	app/src/main/java/org/oppia/android/app/player/stopplaying/StopExplorationDialogBoxFragment.kt	deleted
InsufficientExplorationCheckpointStorageDialogFragment.kt	app/src/main/java/org/oppia/android/app/player/stopplaying/InsufficientExplorationCheckpointStorageDialogFragment.kt	added
ExplorationProgressNotSavedDialogFragment.kt	app/src/main/java/org/oppia/android/app/player/startplaying/ExplorationProgressNotSavedDialogFragment.kt	added
ResumeExplorationActivity.kt	app/src/main/java/org/oppia/android/app/player/startplaying/ResumeExplorationActivity.kt	added



ResumeExplorationActivityPresenter.kt	app/src/main/java/org/oppia/android/app/player/startplaying/ResumeExplorationActivityPresenter.kt	added
resume_exploration_activity.xml	app/src/main/res/layout/resume_exploration_activity.xml	added
SegmentedCircularProgressView	app/src/main/java/org/oppia/android/app/customview/SegmentedCircularProgressView.kt	modified
lesson_chapter_view.xml	app/src/main/res/layout/lessons_chapter_view.xml	modified

This project can be broadly divided into four parts

1. Implementation of mechanism to save, retrieve and delete checkpoint.
2. Implementation of mechanism to manage checkpoint during topic updates and deletion
3. Minor enhancements to the UI
4. Implementation of metrics

The next part of the proposal lists the files that will be needed to be added, updated or deleted in order to implement the project.

## Implementation of mechanism to save and retrieve checkpoints

### ExplorationCheckpointStorage protobuf

1. This protobuf will be added to store details about the exploration checkpoint.

### Topic protobuf

1. This protobuf will be modified because the exploration can now be in two new states i.e. "in progress saved" and "in progress not saved".

### ExplorationCheckpointController

1. This class will be responsible to save the exploration progress, retrieve the exploration checkpoints.

## **StateDeck**

1. This class will be responsible for resuming exploration and creating ExplorationCheckpoint.

## **ExplorationProgressController**

1. This class will also be responsible for resuming the exploration using StateDeck.

## **ExplorationDataController**

1. This class will supply Exploration Checkpoints from the app layer to ExplorationProgressController.
2. This class will also communicate to ExplorationCheckpointController to save the user progress.

The exploration can be started from any one of the following app layer classes. These classes use ExplorationDataController to start the exploration.

1. ***StoryChapterSummaryViewModel***
2. ***TopicLessonsFragmentPresenter***
3. ***RecentlyPlayedFragmentPresenter***

All these three class will

1. Observe combined dataProvider of ExplorationCheckpoint and ExplorationDatabase size. Supply these values or initialize lateinit values accordingly and then start the exploration.

## **StateFragment**

1. This will be responsible to show ExplorationProgressNotSavedDialogFragment(apology dialog) when the progress is not saved at the beginning of the exploration.
2. This class will also be responsible for saving the exploration after every user interaction.

## **StopExplorationDialogFragment**

This dialog fragment warns the user that they will lose all the progress in the exploration before they exit a partially complete exploration. Since we are saving the progress there is no need for this dialog fragment.

### **InsufficientExplorationCheckpointStorageDialogFragment**

This dialog box will be shown to the user if the user is exiting the partially complete exploration advertently. This will be modified to show three options

1. Back
2. Leave exploration without saving progress
3. Leave exploration after replacing the current progress with the oldest exploration progress.

### **ExplorationProgressNotSavedDialogFragment**

This dialog fragment will be shown to the user if the user restarts an exploration whose progress could not be saved during the user's previous attempt.

### **ExplorationActivityPresenter**

1. They will be responsible to show `InsufficientExplorationCheckpointStorageDialogFragment` when storage space runs out.
2. This class will also be responsible to save the data using the domain layer when the user exits the exploration.

### **ResumeExplorationActivity**

1. This activity will be responsible to start or resume the exploration using an `ExplorationCheckpoint`.

## **Implementation of mechanism to manage checkpoint during topic updates and deletion**

In case of topic deletion, we are not deleting the exploration progress. So the only case we need to consider here is a topic update. Now topic updates can happen in two scenarios.

1. Update to a topic already present on the device.
2. Update to a topic that was deleted from a device. This means an updated version of the topic is redownloaded.

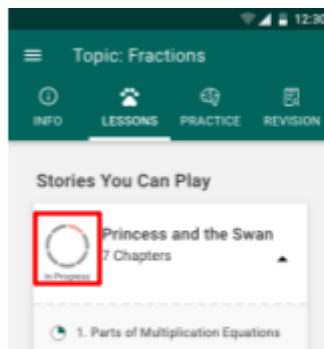
For the first case, we will have a function that updates a topic only if it has no active progress across any profiles. When the app auto-updates the topics this function will be called for every topic, and only those topics which do not have any active progress associated with them will be updated. If the user manually updates a topic, first the topic will be checked, if the topic has no active progress, it will be updated but if it has active progress, the user will be warned by a dialog box, if the user continues the progress will be deleted across all profiles for that topic.

For the second case when a topic is downloaded its version will be matched with all the profiles, and all the exploration progress whose version does not match will be deleted.

## Minor enhancements to the UI

### SegmentedCircularProgressView

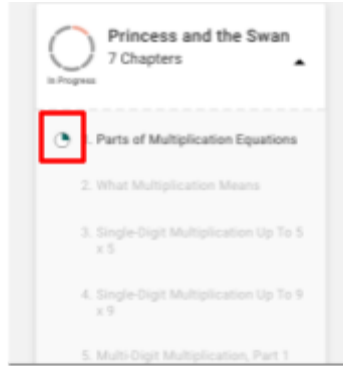
This class draws the segmented progress back on the Topic Lessons Tab. This class will be modified to show progress in the case of “in progress” chapters.



*Box highlighting the segmented progress bar showing “in progress” chapter*

### Lessons\_chapter\_veiw

This is the XML file that is shown in the recycler view of the topic lesson fragment. This class used data-binding to show a “tick icon” for all completed chapters. This XML will be modified in a similar manner to show an icon for all chapters that are “in progress”



*Box highlighting the icon shown for "in progress chapter"*

## Implementation of metrics

Metrics will be stored as count on the device and will be logged when they change using AnalyticController. Whenever any of the following counts are updated, they are logged using AnalyticController.

The first step in the implementation of metrics is a modification of oppia-logger.proto to save the counts.

### Progress Saving count

This count will save the number of times the progress is saved successfully on the device. This count will include all successful saving attempts, be it due to the user leaving advertently or saving of progress after every interaction.

### Error saving count

This count will save the number of times the progress is not saved successfully on the device. This count will include all unsuccessful saving attempts, be it due to the user leaving advertently or saving of progress after every interaction.

This count will be updated in ExplorationCheckpointController and logged by AnalyticController whenever there is a successful saving attempt by the app.

### Lessons saved advertently count

This count will save the number of times the user exits the exploration advertently. This count will only include when the user uses the 'X' icon or back button to exit a partially complete exploration.

### **Lesson saved inadvertently count**

This count will save the number of times the user exits a partially complete exploration inadvertently. This count will not count every saving attempt that is made after every user interaction and only is incremented when the app crashes or the app is cleared from recent.

### **Lesson completion count for saved lesson**

This count will the number of times a lesson is completed, if that lesson was resumed (that means the lesson was saved at some point)

### **“Start Over” count**

This count saves the number of times the start over is clicked. That means the number of times the lesson was started from the beginning.

### **“Continue” count**

This count saves the number of times the lesson is continued or resumed.

### **Correct answers count in a resumed lesson**

This count will save the number of correct answers in a lesson that was resumed.

### **Incorrect answers count in a resumed lesson**

This count will save the number of incorrect answers in a lesson that was resumed.

## Implementation Approach

### ***Alternate approach***

*An alternate approach can be instead of saving the entire checkpoint every time there is a user interaction, we just update the existing checkpoint with the latest checkpoint.*

***cons***

*There is a higher probability of a write transaction failing because there will be a higher number of write transactions.*

***The reason I rejected this approach***

*This approach is not reliable. Suppose there are 5 user interactions. And the 3rd interaction did get saved (due to disk failure or some other error), so the checkpoint will have the following user interactions 1 , 2 ,4 ,5. Now if the exploration is restarted using this checkpoint, the app can behave unexpectedly because there is missing information about one user interaction.*

## Implementation of mechanism to save and retrieve checkpoints

The first step is to create a new protobuf named **exploration\_checkpoint\_storage.proto**. This protobuf will be a model for the data which will be saved in order to save the checkpoint. Four new messages will be created in this protobuf.

### 1. ExplorationCheckpointStates

variable	usage
<code>completedState</code> (CompletedState)	stores CompletedState which stores a list of all AnswerAndResponse to a particular previous state. The last answer in this list is always guaranteed to be the correct answer.
<code>stateName</code> (string)	stores the name of the state to which the responses were made

### 2. ExplorationCheckpoint

variable	usage
<code>pendingTopStateName</code> (string)	stores the last state name visited by the user (top most state of the state deck)
<code>explorationCheckpointStates</code> (repeated ExplorationCheckpointState)	stores a list of all previous states visited by the user and the responses they made to those states.

<code>pendingUserAnswers</code> (repeated AnswerAndResponse)	stores a list of AnswersAndResponse for the pending top state.
<code>stateIndex</code> (int32)	stores the index of the last state user was at.
<code>hintIndex</code> (int32)	stores the index of the hint which was revealed to the user.
<code>Version</code> (string)	stores the version of exploration whose progress was saved.

### 3. OrderedExploration

<code>explorationName</code> (string)	Stores the name of the exploration
<code>explorationId</code> (string)	Stores the Id of the exploration

### 4. ExplorationCheckpointDatabase

<b>variable</b>	<b>usage</b>
<code>explorationCheckpointMap</code> (map<string, ExplorationCheckpoint>)	Stores a map from explorationId to ExplorationCheckpoint
<code>orderedExploration</code> (repeated OrderedExploration)	A list of explorations in the order they were saved. This list contains the name and Id of the exploration. This list will be used to delete the oldest checkpoint in O(1).

Then we will create a class named **ExplorationCheckpointController** in the domain module. This class will be responsible for saving, retrieving the ExplorationCheckpoints for a particular explorationId.

This class will create a **different cache for every profileId**, this ensures that different profileIds can have separate storage available to save the progress. (**each profile will have a limit of 2MBs**)



```

private val cacheStoreMap =
    mutableMapOf<ProfileId, PersistentCacheStore<ExplorationCheckpointDatabase>>()

/**Private function to retrieve cache store and populate map*/
private fun retrieveCacheStore(
    profileId: ProfileId
): PersistentCacheStore<ExplorationCheckpointDatabase> {
    val cacheStore = if (profileId in cacheStoreMap) {
        cacheStoreMap[profileId]!!
    } else {
        val cacheStore = cacheStoreFactory.createPerProfile(
            CACHE_NAME,
            ExplorationCheckpointDatabase.getDefaultInstance(),
            profileId
        )
        cacheStoreMap[profileId] = cacheStore
        cacheStore
    }

    cacheStore.primeCacheAsync().invokeOnCompletion {
        it?.let {
            logger.e(
                "ExplorationCheckpointController",
                "Failed to prime cache ahead of LiveData conversion for ExplorationCheckpointController.",
                it
            )
        }
    }
    return cacheStore
}

```

This class will have a function to retrieve the saved ExplorationCheckpoint. This function will return an ExplorationCheckpoint if progress is found or a default instance of ExplorationCheckpoint if no progress is found.

```

/** returns a [DataProvider] which holds the current database size - size of the current exploration
checkpoint*/
fun explorationDatabaseSize(
    profileId: ProfileId,
    explorationId: String
): DataProvider<Int> {
    return retrieveCacheStore(profileId)
        .transformAsync(
            RETRIEVE_EXPLORATION_CHECKPOINT_DATABASE_SIZE_DATA_PROVIDER_ID
        ) {
            AsyncResult.success(
                it.serializedSize - (it.explorationCheckpointMap[explorationId]?.serializedSize ?: 0)
            )
        }
}

```

This class will also have a function that will return the oldest save checkpoint name and explorationId as OrderedExploration. This will be useful when we show the dialog box if the storage is full.

We will also create a dagger module named **ExplorationStorageModule**. This module will be required to supply constants like the storage limit of 2MB.

### **Analysis of time complexity from write and read data operations.**

Every profile will contain an ExplorationCheckpointDatabase. This database will contain a map from explorationId to ExplorationCheckpoint and a list that contains the name and Id of the exploration.

We are saving the exploration name in the list because the InsufficientStorageDialogFragment requires us to show the name of the oldest exploration in the form of the following message. "The progress of \$oldestCheckpointName will be deleted."

### **Saving progress**

Whenever we save progress for a new ExplorationCheckpoint, we also append the name and explorationId of this exploration in the list. **This operation takes O(1).**

### **Deleting oldest progress to save the current progress**

When we want to delete the oldest progress we remove the explorationId at the 0th index of the list. And remove that explorationId from both the list and the map. **This operation takes O(1).**

### **Retrieving progress**

When we want to retrieve progress, we use the explorationId to get the progress from the map. **This operation takes O(1).**

An alternate approach to find the oldest checkpoint can be saving a timestamp along with the ExplorationCheckpoint.

I rejected this approach because of the following reasons

1. the time complexity to find the oldest checkpoint here becomes  $O(n)$  because we have to traverse the whole map.
2. It can be said that the list in the approach I am using saves the explorationId twice(once in the map and once in the list), though in this alternate approach we will not be saving explorationId twice, but we will have to save timestamp for the

exploration instead.

## Modification of ChapterPlayState

Every chapter can be put into one of the four categories

1. Chapter is not started
2. Chapter is in progress but progress is not saved
3. Chapter is in progress and progress is saved
4. Chapter is completed

So we have to replace the STARTED\_NOT\_COMPLETED state with two new states, in\_progress\_saved and in\_progress\_not\_saved.

The modified ChapterPlayState will have:

1. COMPLETION\_STATE\_UNSPECIFIED
2. NOT\_STARTED
3. IN\_PROGRESS\_SAVED
4. IN\_PROGRESS\_NOT\_SAVED
5. COMPLETED
6. UNRECOGNIZED

When the user starts an exploration **from the beginning**, the exploration will be marked as IN\_PROGRESS\_NOT\_SAVED. When the user starts the exploration, a new checkpoint will be created and saved. If the save is successful the ChapterPlayState will be changed to IN\_PROGRESS\_SAVED.

When the user **resumes** an exploration, the exploration will already be marked as IN\_PROGRESS\_SAVED so it will not be changed.

Every time the user interacts with the exploration, the updated checkpoint will be saved and the ChapterPlayState will be updated as follows:

1. If the checkpoint is saved successfully the chapter will be marked as IN\_PROGRESS\_SAVED.
2. If the checkpoint is not saved successfully the chapter will be marked as IN\_PROGRESS\_NOT\_SAVED.

Once the user completes the chapter the chapter will be marked as COMPLETED.

### I. Implementation of mechanism to save checkpoints

1. To save ExplorationCheckpoints we will create a function in StateDeck that takes the current values of the StateDeck and creates an ExplorationCheckpoint with them.

```

/** checkpoints the [EphemeralState] of the exploration and returns a [ExplorationCheckpoint] */
internal fun markExplorationCheckpoint(): ExplorationCheckpoint {

    val explorationCheckpoint = ExplorationCheckpoint.newBuilder()

    previousStates.forEach { state ->
        val explorationCheckpointStates =
            ExplorationCheckpointStates.newBuilder()
                .setCompletedState(state.completedState)
            explorationCheckpoint.addExplorationCheckpointStates(explorationCheckpointStates)
        }
    explorationCheckpoint.pendingTopStateName = pendingTopState.name
    explorationCheckpoint.hintIndex = hintList.size
    explorationCheckpoint.stateIndex = stateIndex
    explorationCheckpoint.addAllPendingUserAnswers(currentDialogInteractions)

    return explorationCheckpoint.build()
}

```

The class `ExplorationCheckpointController` apart from the functions for data operations will also have an enum and a class as follows

```

/** Indicates that the checkpoint was saved but the database has exceeded the storage limit. */
class CheckpointDatabaseSizeLimitExceeded(msg: String) : Exception(msg)

/** These Statuses correspond to the exceptions above such that if the deferred contains. */
private enum class ExplorationCheckpointActionStatus {
    CHECKPOINT_SAVED_DATABASE_SIZE_LIMIT_NOT_EXCEEDED,
    CHECKPOINT_SAVED_DATABASE_SIZE_LIMIT_EXCEEDED
}

```

When a new exploration checkpoint is saved, the size of the database will be checked and the corresponding deferred result will be returned.

```

private suspend fun getDeferredResult(
    deferred: Deferred<ExplorationCheckpointActionStatus>
): AsyncResult<Any?> {
    return when (deferred.await()) {

        ExplorationCheckpointActionStatus.CHECKPOINT_SAVED_DATABASE_SIZE_LIMIT_EXCEEDED ->
            AsyncResult.success(CheckpointDatabaseSizeLimitExceeded("Database size exceeded"))

        ExplorationCheckpointActionStatus.CHECKPOINT_SAVED_DATABASE_SIZE_LIMIT_NOT_EXCEEDED ->
            AsyncResult.success(null)
    }
}

```

We will add two new variables in the StateFragmentPresenter, **isDatabaseFull** (Boolean) and **currentChapterState** (enum of "in\_progress\_saved" and "in\_progress\_not\_saved"). isDatabaseFull is a flag that tells us if the database has exceeded the storage limit and currentChapterState is added so that whenever it changes we can also change the value of **ChapterPlayState**.

Every time the user interacts with the exploration( submits an answer or clicks the continue button), an exploration checkpoint will be created (containing all necessary variables of StateDeck). This checkpoint will be saved and a dataProvider will be returned to the StateFragmentPresenter. This dataProvider will be observed in the StateFragmentPresenter.

The result of the dataProvider can either be success or failure.

If the result is successful, the **currentChapterState** variable will be set as "in\_progress\_saved" and if **ChapterPlayState** will be marked as IN\_PROGRESS\_SAVED.

## II. Implementation of mechanism to retrieve checkpoints

Exploration can be started from any of the following three classes.

- ❖ *StoryChapterSummaryViewModel*
- ❖ *TopicLessonsFragmentPresenter*
- ❖ *RecentlyPlayedFragmentPresenter*

The implementation of retrieving the progress will be identical from all of the three classes. As an example, I am assuming we are starting the exploration from TopicLessonFragmentPresenter.

1. When the exploration name is clicked, we will retrieve the data provider to retrieve the ExplorationCheckpoint from ExplorationCheckpointController, convert it to live data and observe it

Once the data is retrieved, we will check if it is equal to

**ExplorationCheckpoint.defaultInstance()**, if this is true that means no progress was found for the current explorationId, in this case, we will start the exploration from the beginning by supplying the defaultInstance of ExplorationCheckpoint to startPlayingExploration() of ExplorationDataController(modified later in this proposal).

```
/** function to retrieve [ExplorationCheckpoint] using [ExplorationCheckpointController]
 * This function is observed when the [selectChapterSummary] function of this class. */
private fun subscribeToExplorationCheckpointLiveData() {
    explorationCheckpointLiveData.observeOnce(
        fragment,
        Observer<ExplorationCheckpoint> { explorationCheckpoint ->
            when(explorationCheckpoint) {
                ExplorationCheckpoint.getDefaultInstance() ->
                    // no progress was found or
                    // the exploration version did not match.
                    playExploration(
                        internalProfileId,
                        topicId,
                        storyId,
                        explorationId,
                        /* backflowScreen= */ 0,
                        explorationCheckpoint // passing the defaultInstance of ExplorationCheckpoint
                        // which will start the exploration from the beginning.
                    )
                else -> explorationCheckpointFound()
            }
        }
    )
}

/** starts ResumeExplorationActivity if ExplorationProgress is found. */
private fun explorationCheckpointFound() {
    routeToResumeExplorationListener.routeToResumeExploration(
        internalProfileId,
        topicId,
        storyId,
        explorationId,
    )
}
```

If it is false we will start the ResumeExplorationActivity and send the ExplorationCheckpoint and the internalProfileId to this activity.

2. Now here if the user clicks “continue” we will use start the exploration by passing the ExplorationCheckpoint but if the user clicks “start over” we will start the exploration by passing the defaultInstance of ExplorationCheckpoint to startPlayingExploration() of ExplorationDataController. Here we will also supply internalProfileId to ExplorationDataController.
3. Now we will modify startPlayingExploration() function of **ExplorationDataController** to receive an ExplorationCheckpoint and internalProfileId as function parameter and passing it onto beginExplorationAsync of the ExplorationProgressController.

```
fun startPlayingExploration(
    internalProfileId: Int = -1,
    explorationId: String,
    explorationCheckpoint: ExplorationCheckpoint
): LiveData<AsyncResult<Any?>> {
    return try {
        explorationProgressController.createExplorationCheckpoint()
        //internalProfileId and ExplorationCheckpoint passed to ExplorationProgressController
        explorationProgressController.beginExplorationAsync(
            internalProfileId,
            explorationId,
            explorationCheckpoint
        )
        MutableLiveData(AsyncResult.success<Any?>(null))
    } catch (e: Exception) {
        exceptionsController.logNonFatalException(e)
        MutableLiveData(AsyncResult.failed(e))
    }
}
```

4. Next, we will modify **ExplorationProgressController**, here we will set the internalProfileId and the ExplorationCheckpoint as global variables for this class by initializing lateinit variables with them. Internal profileId will be used to save the new checkpoint, and ExplorationCheckpoint is used to resume the exploration.

```

private var internalProfileId = -1
private lateinit val explorationCheckpoint

// only needed we if we want to save checkpoints after every interaction
// this variable will also be initialized in beginExplorationAsync() function from
// ExplorationDataController.
private var explorationDatabaseSize = 0

internal fun beginExplorationAsync(
    internalProfileId: Int,
    explorationId: String,
    explorationStates: ExplorationStates
) {
    explorationProgressLock.withLock {
        check(explorationProgress.playStage == ExplorationProgress.PlayStage.NOT_PLAYING) {
            "Expected to finish previous exploration before starting a new one."
        }

        //lateinit variables initialized here
        this.explorationCheckpoint = explorationCheckpoint
        this.internalProfileId = internalProfileId

        explorationProgress.currentExplorationId = explorationId
        explorationProgress.advancePlayStageTo(ExplorationProgress.PlayStage.LOADING_EXPLORATION)
        asyncDataSubscriptionManager.notifyChangeAsync(CURRENT_STATE_DATA_PROVIDER_ID)
    }
}

```

Next we will modify the `finishLoadingExploration()` to use `StateDeck.resetDeck()` if `defaultInstance` of `ExplorationCheckpoint` is supplied else `StateDeck.resumeDeck()` will be called.



```

private fun finishLoadExploration(exploration: Exploration, progress: ExplorationProgress) {
    // The exploration must be initialized first since other lazy fields depend on it being inited.
    progress.currentExploration = exploration
    progress.stateGraph.reset(exploration.statesMap)

    when (explorationCheckpoint) {
        ExplorationCheckpoint.getDefaultInstance() -> {
            // reset exploration if no saved explorationCheckpoint is found or if user chooses to restart.
            progress.stateDeck.resetDeck(progress.stateGraph.getState(exploration.initStateName))
        }
        else -> {
            // resume the exploration if a valid exploration progress is found
            progress.stateDeck.resumeDeck(
                pendingTopState = explorationProgress.stateGraph.getState(
                    explorationCheckpoint.pendingTopStateName
                ),
                previousStates = getPreviousStatesFromCheckpoint(
                    explorationCheckpoint.explorationCheckpointStatesList
                ),
                currentDialogInteractions = getCurrentDialogInteractionsFromCheckpoint(
                    explorationCheckpoint.pendingUserAnswersList
                ),
                hintList = getHintListFromCheckpoint(explorationCheckpoint.hintIndex),
                stateIndex = explorationCheckpoint.stateIndex
            )
        }
    }
}

```

Here `getPreviousStatesFromCheckpoint()`, `getCurrentDialogInteractionsFromCheckpoint()` and `getHintListFromCheckpoint()` are private function that are used to get corresponding list from `ExplorationCheckpoint`. Sample implementation of `getPreviousStatesFromCheckpoint()` is shown below, and others are similar to this.

```

private fun getPreviousStatesFromCheckpoint(
    explorationCheckpointStatesList: MutableList<ExplorationCheckpointStates>
): MutableList<EphemeralState> {

    val previousStates: MutableList<EphemeralState> = ArrayList()

    for (previousState in explorationCheckpointStatesList) {
        previousStates += EphemeralState.newBuilder()

            .setState(explorationProgress.stateGraph.getState(previousState.stateName))
            // false when the index is 0 i.e first state.
            .setHasPreviousState(explorationCheckpointStatesList.indexOf(previousState) != 0)
            .setHasNextState(true)
            .setCompletedState(previousState.completedState)
            .build()
    }
    return previousStates
}

```

- Our final step is to add a function in StateDeck to resume the exploration. This function is very simple; it gets all its values from ExplorationProgressController and sets these values to their respective variables.

```

/** Resumes this deck to a new, specified [ExplorationCheckpoint]. */
internal fun resumeDeck(
    pendingTopState: State,
    previousStates: MutableList<EphemeralState>,
    currentDialogInteractions: MutableList<AnswerAndResponse>,
    hintList: MutableList<Hint>,
    stateIndex: Int
) {
    this.previousStates.clear()
    this.currentDialogInteractions.clear()
    this.hintList.clear()
    this.pendingTopState = pendingTopState
    this.previousStates += previousStates
    this.currentDialogInteractions += currentDialogInteractions
    this.hintList += hintList
    this.stateIndex = stateIndex
}

```

# Implementation of mechanism to manage checkpoint during topic updates and deletion

## I. The mechanism for topic updates

To handle topic updates we will create a function **canTopicBeUpdated()**. This function will check if the topic has any exploration which has progress associated with them. If this function finds progress it will return False otherwise it will return True. This can be done either by saving the topicId in the version variable of ExplorationCheckpoint and comparing the topicId or passing a list of explorationId associated with that topic and checking for all explorationIds. This function checks for explorationCheckpoint saved across all profiles

While handling topic updates if have to take care of two situations

1. Topic update when auto-update is turned on: In this case, for every topic first the function canTopicBeUpdated() runs and based on the result topic will be updated.
2. Topic update when updated manually: In this case when the user clicks to update the topic first canTopicBeUpdated() runs if the result is True topic is updated, but if the result is false the user is shown a dialog box to warn them that progress across all profiles will be deleted. If the user still proceeds the progress across all profiles for that topic is deleted.

## II. The mechanism for topic deletion and re-download

When a topic has deleted the progress associated with it will not be affected in any way. When the user re-downloads the topic there can be two cases

1. The topic re-downloaded is of the same version: In this case, the topic can be resumed without any implementation. The existing progress will be used to resume the lesson.
2. An updated version of the topic is re-downloaded: In this case, the topic cannot be resumed. Here we have two implementations.
  - a. Because the explorations are versioned we can compare the exploration version for the downloaded topics with that of the saved exploration checkpoints. In case the version doesn't match we then delete the checkpoint.
  - b. We delete the checkpoint for all exploration whose topic was updated.

Both the approaches will work as both delete the checkpoints which can not be used. But the first approach is clearly better as it provides a little more flexibility in case the topic version changed due to a change in some of the explorations but some explorations were not updated.

## Minor enhancements to the UI

1. To show the arc for every chapter in progress the functions `onDraw()` and `initialize()` of ***SegmentedCircularProgressView*** will be changed as follows.

```
private fun initialise() {
    ... // code already there

    chapterStartedNotFinishedArcPaint = Paint(Paint.ANTI_ALIAS_FLAG)
    chapterStartedNotFinishedArcPaint.style = Paint.Style.STROKE
    chapterStartedNotFinishedArcPaint.strokeCap = Paint.Cap.ROUND
    chapterStartedNotFinishedArcPaint.strokeWidth = strokeWidth
    chapterStartedNotFinishedArcPaint.color =
        ContextCompat.getColor(context, R.color.oppiaProgressChapterStartedNotFinished)
}

override fun onDraw(canvas: Canvas) {
    ...// code already there

    // Draws arc for every started not finished chapter.
    for (i in 0 until chaptersStartedNotFinished) {
        val startAngle =
            angleStartPoint + i * (sweepAngle + STROKE_DASH_GAP_IN_DEGREE) +
            STROKE_DASH_GAP_IN_DEGREE / 2
        canvas.drawArc(baseRect!!, startAngle, sweepAngle, false, chapterStartedNotFinishedArcPaint
        )
    }

    ...// code already there
}
```

2. T. Next to show an icon next to the chapter “in progress” similar to the tick icon we will have to update the layout file **`lessons_chapter_view.xml`** and use data-binding to show the icon. The implementation will be exactly the same as that for the tick icon.

## Implementation of metrics

The first step to implement metrics is to create a message structure in **`oppia_logger.proto`** to save the metrics

First, we will implement metrics that will be uploaded to Firebase Analytics.

1. LightweightCheckpointingLog

<b>Timestamp</b> (int64)	Stores a timestamp for the metric
<b>Metric</b> oneof(int32, float)	Stores either the count (int) or the rate(float) of the metric
<b>checkpointMetricType</b> (enum)	Enum containing the following <ol style="list-style-type: none"> <li>1. CHECKPOINT_METRIC_UNSPECIFIED</li> <li>2. RATE_PROGRESS_SAVING_SUCCESSFUL</li> <li>3. RATE_PROGRESS_SAVING_FAILURE</li> <li>4. RATE_SAVED_LESSON_COMPLETION;</li> <li>5. COUNT_LESSONS_SAVED_DELIBERATELY</li> <li>6. COUNT_CHAPTER_SAVED_INADVERTENTLY</li> <li>7. RATE_START_OVER = 7;</li> <li>8. RATE_CHAPTER_CONTINUE</li> <li>9. RATE_CORRECT_ANSWER_IN_SAVED_LESSON</li> <li>10. RATE_INCORRECT_ANSWER_IN_SAVED_LESSON</li> </ol>

Now we will define a structure for the message to save the raw counts for each type of the metric

2. CountWithExplorationId : These are the message that requires an explorationId associated with them like RATE\_CORRECT\_ANSWER\_IN\_SAVED\_LESSON

<b>explorationId</b> (string)	The Id of the exploration the metric is associated with.
<b>count</b> (int32)	The raw count of the metric

### 3. MetricCount

<code>metricType</code> <code>oneOf(int32, CountWithExplorationId)</code>	Stores int32 when only the count needs to be saved for the metrics and saves <code>CountWithExplorationId</code> when the count also has to be saved along with the metric.
--	---

### 4. MetricCountDatabase

<code>MetricCountDatabase</code> <code>(map&lt;string, Metriccount&gt;)</code>	Maps a <code>metricName</code> to <code>MetricCount</code>
---	--

Next, we will do some new functions in the **AnalyticController**. The implementation of these functions will be similar to that used to log `EventLogs`.

The map `MetricCountDatabase` will map count names (which will be constant strings) to the count of the metric. Whenever the metric has to be logged, the count will be retrieved incremented by one and logged to Firebase. The metric whose rate has to be calculated will have an extra step of calculation of rate before they can be logged to Firebase.

**If we save/update the checkpoint after every user interaction then** the implementation of every metric except `COUNT_CHAPTER_SAVED_INADVERTENTLY` is pretty straight forward. One possible approach to handle the count of inadvertent saves is to log them every time an exploration marked as “in progress not saved” is started.

## Third-party Libraries\*

There are no third-party libraries that are required for the implementation of this project.

## Testing Approach

Testing is a very important part of any project. It ensures that the new code added to the codebase does not break the existing functionality of the project.

Here in this project, we are required to write roboelectric and espresso tests for the following.

1. `ExplorationCheckpointController`: This class will be thoroughly tested. We will have to check that all of the functions like `save`, `delete`, `update` function for the

ExplorationCheckpoint work correctly. There will be a new class named **ExplorationCheckpointControllerTest** that will be used to test this class.

2. We will have to check that the entire domain layer implementation of this project works correctly. This includes the classes StateDeck, ExplorationProgressController and ExplorationDataController.
3. Next, we will also test the ResumeExplorationActivity. It will be made sure through testing that the components of this activity are correctly displayed. Also, this class will be tested to make sure it works correctly in the implementation of lightweight checkpointing.
4. Minor UI changes (SegmentedProgressBar and lessons\_chapter\_view are correctly displayed and the icons or progress is visible correctly for progress chapters.

## Milestones

### Milestone 1

**Key Objective:** Implementation of mechanism to save and retrieve checkpoints in the domain layer, and adding necessary UI classes in the app layer.

Milestone 1 will not change any actual functioning of the develop branch instead it will prepare the domain and app layer classes for implementing checkpointing in milestone 2

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Addition of <b>exploration_storage</b> protobuf to save checkpoints, modification of <b>oppia_logger</b> proto and modification of <b>topic</b> proto(for chapterPlayState)	N/A	Jun 8, 2021	Jun 11, 2021
1.2	Implementation <b>ExplorationCheckpointController</b> to add support to save, retrieve, delete and find the oldest checkpoint. And adding tests for ExplorationCheckpointController.	1.1	Jun 13, 2021	Jun 18, 2021
1.3	Implementation domain layer mechanism to save and resume exploration. (StateDeck , ExplorationProgressController and	1.2	Jun 24, 2021	Jun 29, 2021

	ExplorationDataController)			
1.4	Modification of StoryProgressController for in progress chapters and implementation of UI changes(segmented progress bar) to show in progress chapters	1.3	July 2, 2021	July 5, 2021
1.5	Implementation of <b>ResumeExplorationActivity</b> , its presenter and layout file.	N/A	July 4, 2021	July 8, 2021
1.6	Addition of new dialog fragments 1. InsufficientStorageDialogFragment 2. ProgressNotSavedDialogFragment	N/A	July 7, 2021	July 11, 2021

## Milestone 2

**Key Objective:** Implement Lightweight checkpointing for explorations and success metrics.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Modification of StateFragment and ExplorationActivity to save exploration on user interaction.	1.3	July 19, 2021	July 24, 2021
2.2	Modification of App layer classes to start exploration ( <i>StoryChapterSummaryViewModel, TopicLessonsFragmentPresenter, RecentlyPlayedFragmentPresenter</i> )	1.3	July 27 2021	1 August, 2021
2.3	Implementation of mechanism to handle topic updates	1.3	3 August 2021	August 7, 2021
2.4	modification of AnalyticsController to save metrics	1.1	August 9, 2021	August 13, 2021
2.5	Implementation of success metrics.	2.1,2.2,2.3 2.4	August 14, 2021	August 18, 2021



# Optional Sections

## Additional Project-Specific Considerations

### Privacy

Lightweight checkpoint saves all the user data locally on the device, so there is no privacy concern for this project.

### Security

All the checkpoints will be stored internally within the app so this feature does not create any new security issue.

### Accessibility (if user-facing)

The new UI being introduced here is the ResumeExplorerActivity which does not include a very complex UI. Further, it will be made sure that this activity passes the tests of the A11y scanner.

It will be made sure that the light orange colour and the icon introduced for the “in progress chapter” pass the a11y tests on the a11y scanner.

The last UI component that will be added in this project are the two dialog box( InsufficientStorageDialogFragment, ProgressNotSavedDialogFragment), both of these dialog boxes will also be made to pass the a11y tests of a11y scanner.

### Documentation Changes\*

There are no changes in the documentation that will be required after the implementation of this project.

### Ethics\*

Since this project just involves saving the user progress, there are no ethical considerations that have to be taken care of.

### Future Work

Future work will include the implementation of Full version checkpointing. This is an extension of lightweight checkpointing. It involves saving the progress at specific points in the exploration.