Improve the image loading pipeline

GSOC 2018 PROPOSAL

Name : Aashish Gaba

Email : aashishgaba097@gmail.com, aashish.gaba@students.iiit.ac.in

Github handle : ishucr7

University : International Institute of Information Technology, Hyderabad

Program : Btech in Computer Science

I am currently pursuing my 2nd year of BTech in Computer Science. IIIT Hyderabad is among the best institutions for computer science in India.

Project Details

Name of the project :- **Improve the image loading pipeline**

Why are you interested in working with Oppia?

I am very fond of online learning platforms. Be it college exams, other competitive exams and improving knowledge, these educational platforms are always helpful. I found Oppia while I was searching for the orgs to begin my open source journey. Oppia is one of the reasons I learned angularjs and now I feel very comfortable in using it. Teaching using explorations and that too with the audio translations available in various languages makes teaching interactive. Humans learn better when they are allowed to make mistakes and correct them which is exactly what explorations help them with. The team members are very welcoming and helpful. The members are very responsive to any queries regarding the org or the codebase or the contributions. Since the platform helps children to learn better, helping in such a cause is satisfying in itself.

What interests you about this project? Why is it worth doing?

We currently have a system for loading audio that preloads and caches audio files. But there is no such system in the case of images. Currently, the images in lessons take a while to load. This results in students (especially those with poor connectivity) seeing no images for an extended period, which causes them to misinterpret questions and select incorrect answers, leading to frustration. Currently, there is no loading indication (if the image is in the process of loading), which results in a poor user experience as learners would be reading a card with important information missing.
The audio files are there on the Google Cloud Storage whereas the images data still exists on App Engine Datastore. Google Cloud Storage is better than App Engine Storage for storing immutable objects (images,audio). After completing this project, not only the existing image files will sit along the audio files but upcoming image files in future will be saved on Google Cloud Storage.

Prior experience (especially with regards to technical skills that are needed for the project).

I have been coding in python from my first semester in college.(present semester -- 4th) I :-

1) had built a image gallery app (using python and javascript) similar to JuiceBox as a part of my course, which allowed users to upload, share, like and comment on images.
2) made an ultimate tic tac toe bot in python.
   a) Using the Alpha Beta pruning and an evaluator function.
3) have some experience with Django (Blog app).
4) have worked with VLEAD as a part of our course project which involved developing the visualization of the Data Structure algorithms so as to make the learning experience for the students in the labs interesting.
5) made a game(without using any pygame module) which runs on terminal.

My experience with angular began when I started contributing to Oppia. I have been contributing to Oppia from October 2017 and have submitted nearly 20 PR's, filed some issues. I have implemented the correctness footer feature. I am familiar with the frontend and backend code of the Oppia.

Source code for some of my work can be found on my github profile

Links to 1-5 PRs you've made that showcase your best work, especially any Oppia ones.

- #4270
- #4363
- #4456
- #4724
- #4702

Project plan and implementation strategy.

The project aims on extending the existing caching and preloading functionality of the audio files to the image files. Along with that, the project plan involves shifting the storage of image files from App Engine Datastore to Google Cloud Storage, so that the audio and image files sit along.

- The reasons why we should preload and cache the image files :-
  - It increases responsiveness, decreases noticeable time lags.
  - Preloading the images rather than loading later helps ensure that users have a great experience in viewing the exploration.
  - With caching, all the files will be loaded beforehand in the actual order as they are in an exploration.

- Why Google Cloud storage over App Engine Datastore for storing images?
  - Google Cloud Storage is for storing immutable blob objects (images, and static files).
  - App Engine Datastore is for storing structured application data that are mutable (User entity, Blog post, etc).
  - Since the project is about storing the images, therefore GCS turns out to be the best option amongst App Engine Datastore and GCS.

- I intend on using mapreducers in one off migration jobs (to move the storage of images from app engine datastore to google cloud storage).

    - ➢ Why mapreducers?
        - ■ Because there are already existing one off jobs in Oppia's codebase which use mapreducers.

- ❖ Testing in Milestone 2 and Milestone 3
    - ➢ The tests in these milestones include the code which tests the storage and upload of images files from App Engine Datastore in dev mode and from GCS in production mode. Since I will be writing code in dev mode, the tests written would not be able to use the GCS system. But we have to test the GCS system before getting the code merged. For that, **we need a way to get the GCS locally. Since there's no way to do so, I will have to deploy my branch of Oppia to Google Cloud Platform and test it there**.(reference)

    - ➢ This way of testing needs to be done before the PRs in the milestone 2 and milestone 3 get merged.

Overview of the AssetsBackendApiService and caching and preloading service for audio.

- ➔ Currently there is only one `AssetsBackendApiService`
    - ◆ It serves as an interface for fetching and uploading the audio.

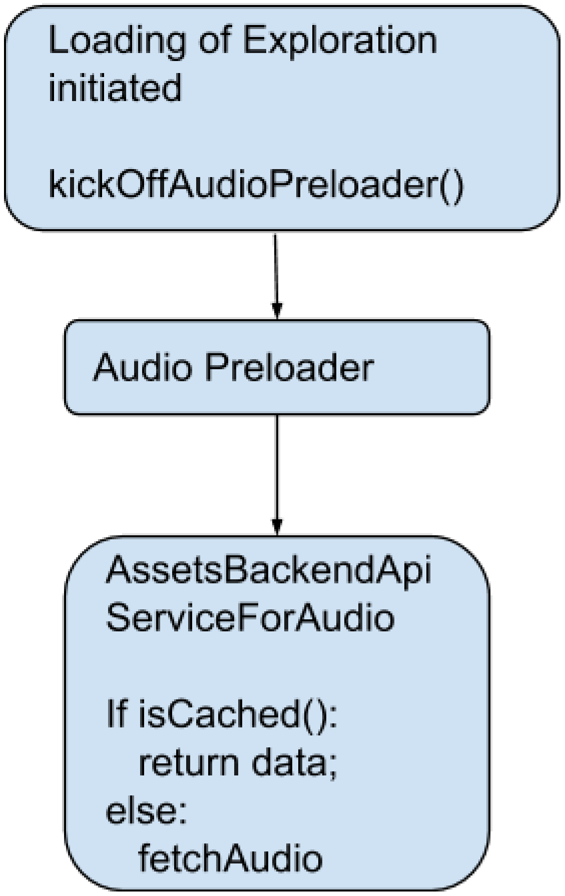
- ➔ I will create a service `AssetsBackendApiServiceForImage` and rename `AssetsBackendApiService` to `AssetsBackendApiServiceForAudio`. The `AssetsBackendserviceForImage` will follow a similar pattern as it is in case of audio.

    - ◆ It's better to **create two separate services** because of the following reasons:-
        - ● `restartAudioPreloader()` in the `AudioBarDirective.js` calls the `AssetsBackendApiService.abortAllDownloads()` which aborts all the downloads. Therefore **if we have a common service, then** `restartAudioPreloader()` **would actually abort the downloading for both audio and image instead of stopping the download for just audio.**

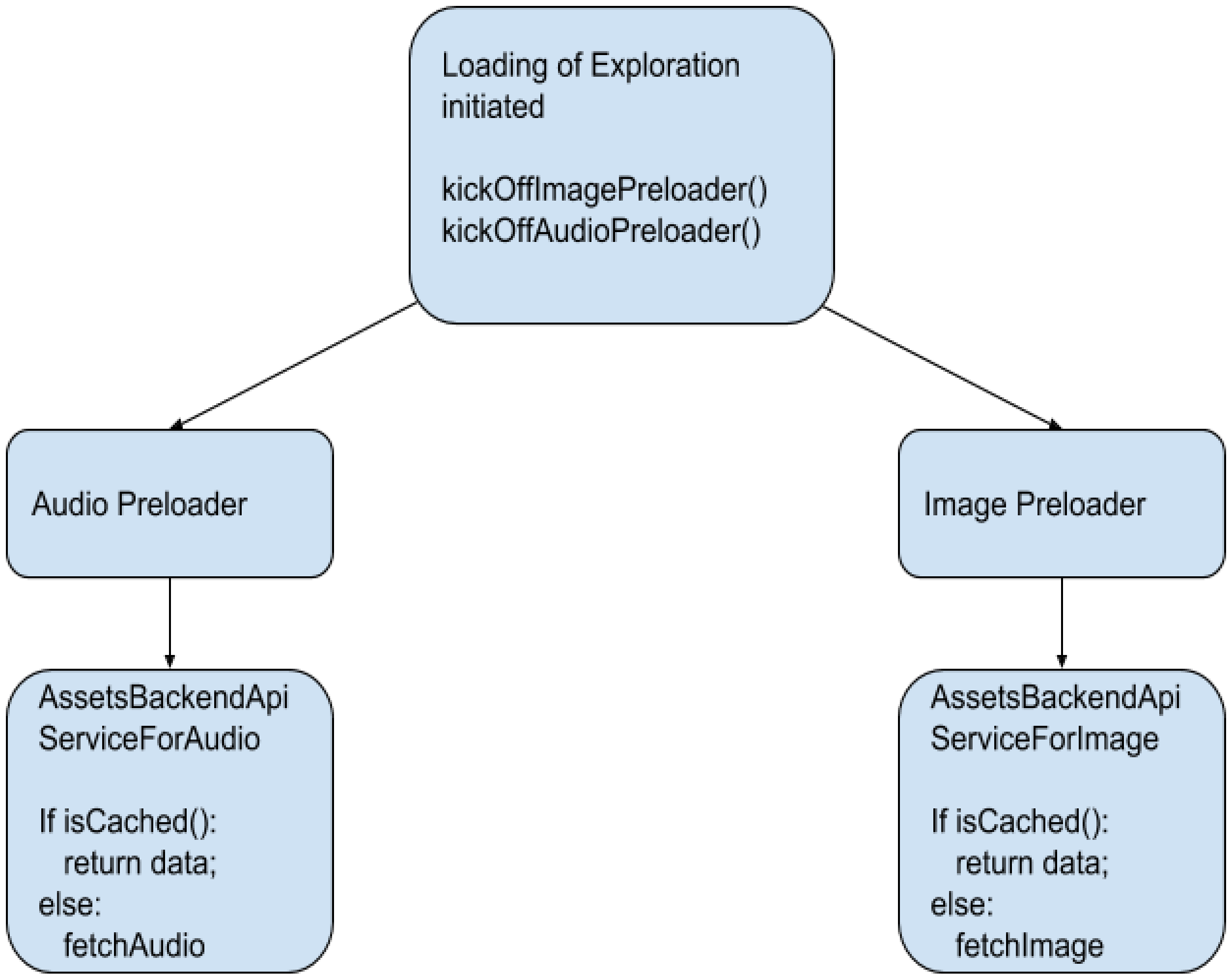
- Also, the functions `fetchImage()` and `saveImage()` are going to be different from the `fetchAudio()` and `saveAudio()`
- This would allow having a separate cache for image.

➔ How Audio preloader works now

  - The loading of Exploration is initiated in the PlayerServices.js, it calls the `kickOffAudioPreloader()`.



➔ We will use a similar approach for preloading the images. This is how the workflow will be :-



➔ The `AudioPreloaderService` and `ImagePreloaderService` start preloading and caching the audio and image files respectively.

➔ They load the files using their `AssetsBackendApiService` by checking if the files are there in the cache.

- If they are in the cache then
  return the data
- else
  Audio (using `fetchAudio()`) is fetched from the Google Cloud Storage or App Engine Datastore whereas the images (using fetchImage) are fetched only from the latter.

Timeline for implementation

1st May - 13th May :-

- Interact with the mentors, discuss the project.
- Solve some issues (if already there) related to the project.
- Prepare a separate doc for refactoring of code. This would help in keeping refactoring of code in a clean and clear manner.

Milestone 1

- Create AssetsBackendApiServiceForImage and rename `AssetsBackendApiService` to `AssetsBackendApiServiceForAudio`.
- Create an ImagePreloaderService and implement the tests for it.
- Create ImageDisplayService and implement loading indicator functionality in the `ImageDirective.js`. Implement the tests for it.

Create AssetsBackendApiServiceForImage

- Rename `AssetsBackendApiService` to `AssetsBackendApiServiceForAudio`.
- Create the `AssetsBackendApiServiceForImage` similar to the one for audio.
  - It will be similar to the current `AssetsBackendApiService.js`
    - Just do not include the part where the download url template uses the GCS bucket -- since that is for fetching file from the GCS. We have not yet implemented the code for storing or fetching images from GCS. (It will be implemented later later in milestone 2)

- AssetsBackendApiService.js
  Line 30 - 34

```
var AUDIO_DOWNLOAD_URL_TEMPLATE = (
     GLOBALS.GCS_RESOURCE_BUCKET_NAME ?
     ('https://storage.googleapis.com/' +
GLOBALS.GCS_RESOURCE_BUCKET_NAME +

'/<exploration_id>/assets/audio/<filename>') :

'/audiohandler/<exploration_id>/audio/<filename
>');
```

➔ Create AssetsBackendApiServiceForImageSpec.js
  - The file tests the AssetsBackendApiServiceForImage created above. It checks that the image is being uploaded and fetched properly.

Create an ImagePreloaderService

➔ Create exploration_player/ImagePreloaderService.js
  - Name of the service :- “ImagePreloaderService”
  - Purpose :- “Service **to preload image into AssetsBackendApiServiceForImage's cache**”
  - Usage :- The service will be used in the exploration_player/PlayerService.js every time the exploration is being loaded (in editor preview or normally)

  CODE

Functions
kickOffimagePreloader, restartImagePreloader, isLoadingImageFile, getFilenamesOfImageCurrentlyDownloading , loadImage

- kickoffImagePreloader
  - **Input** :- sourceStateName
  - **Starts pre-loading of the images**.

- Gets filenames in required order as per the exploration using getImageFilenamesInBfsOrder function.
- Starts loading the images in the same order using the `AssetsBackendApiServiceForImage`.

- getImageFilenamesInBfsOrder
  - **Input** :- null
  - Gets the image filenames in order using the `ComputeGraphService.computeBfsTraversalOfStates()`

- loadImage
  - **Input** :- filename
  - Loads the image using the `AssetsBackendApiServiceForImage.loadImage()`

- restartImagePreloader
  - **Input** :- sourceStateName
  - Aborts the current downloading of image files.
  - Starts the kickOffImagePreloader again.

- isLoadingImageFile
  - **Input** :- filename
  - Checks if the given filename is being loaded.
  - Returns a boolean.

- getFilenamesOfImageCurrentlyDownloading
  - **Input** :- null
  - Returns the image filenames which are currently being downloaded

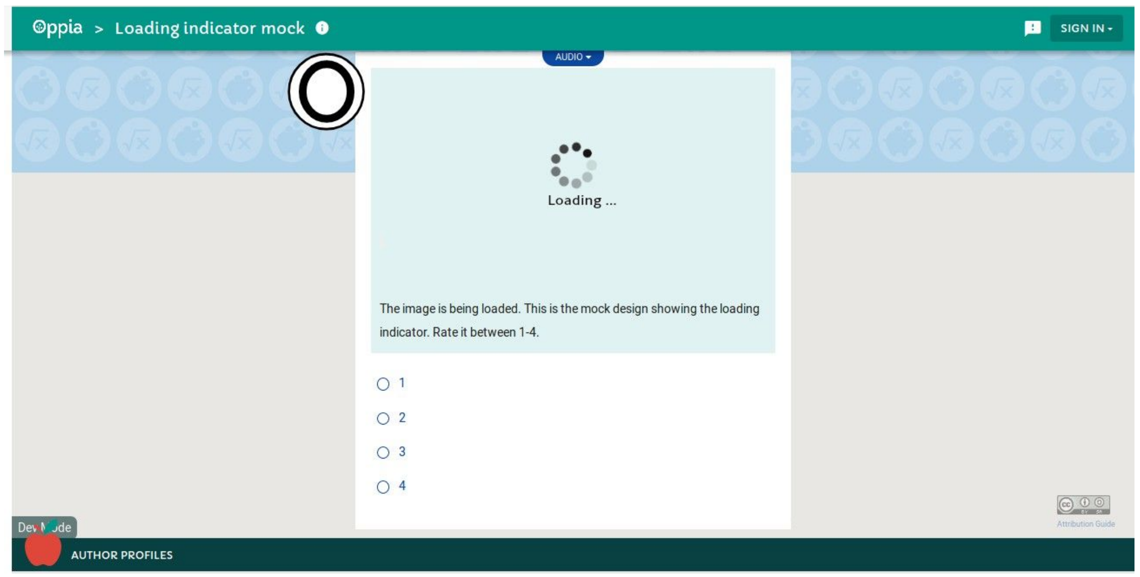
➔ Create `ImagePreloaderServiceSpec.js`.
  - Tests the `ImagePreloaderService` created above. It checks :-
    - that the image is preloaded using the `AssetsBackendApiServiceForImage`.
    - that the image gets loaded in case it was not preloaded.

Display a loading indicator

➔ Display a **loading gif**, if the image is currently unavailable (is not yet downloaded or is being downloaded), so that the user experience is not affected.
   - Below is the gif to be used(link)

     The gif is taken from loading.io under CCO license for free. **CCO license allows us to use it freely for any purpose without any attribution.**

   - Below is the mock design. Here is the link for full size image.

     The image is being loaded. This is the mock design showing the loading indicator. Rate it between 1-4.

   - Currently there is no condition for checking that the image is loaded or not. If it's there in the cache then display the image else the 'loading' gif must be shown.

➔ For the above functionality (Display a loading indicator) :-
   - Create `ImageDisplayService.js`
   - Name of the service:- `'ImageDisplayService'`

- Purpose :- “Service which **decides whether to display the image or loading indicator**”

CODE

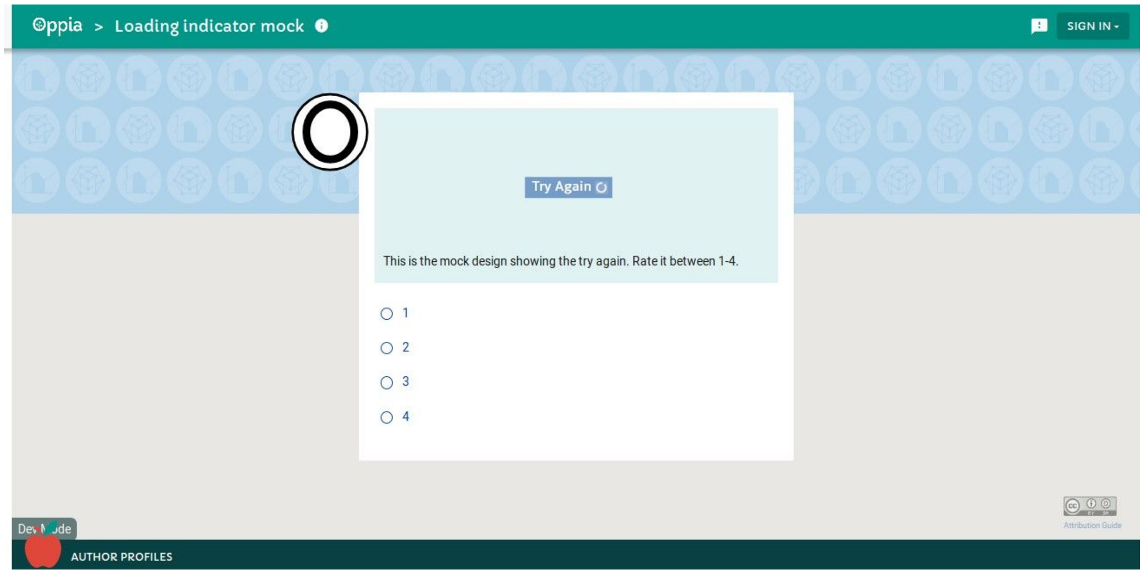
```
Variables

showLoadingIndicator
        Boolean
        Whether the Loading Indicator should be shown or not.

Functions

    ● loadAndDisplay
        ○ Input :-  filename
        ○ Sets the showLoadingIndicator to true
        ○ If the image is in the cache
                Set showLoadingIndicator to false
        ○ else
                AssetsBackendApiForImage.loadImage(filename)
        i.e load the image from the AssetsBackendApiServiceForImage
```

➔ Edit `ImageDirective.js`

```
In the ImageDirective.js

Add the following elements

Functions

    ● showLoadingIndicator
        ○ Input :- null
        ○ If  ImageDisplayService.showLoadingIndicator is true then
              call ImageDisplayService.loadAndDisplay()
        ○ else
              return false;
```

- Create `ImageDisplayServiceSpec.js`
    - It tests that
        - Loading indicator is shown when the image is loading.

- Add a **Try again** button
    - Below is the mock design of the try again. Here is the link of full sized image.



This is the mock design showing the try again. Rate it between 1-4.

○ 1
○ 2
○ 3
○ 4

- If the http request (for getting the image) returns timeout or some other error then
    - display the **Try Again** button. This would benefit in a manner that only the image will have to be reloaded instead of the whole page.
    - The http request is sent from the `AssetsBackendApiService`, below is the code.

        - Line 42 - 50

```
var canceler = $q.defer();
     _filesCurrentlyBeingRequested.push(

FileDownloadRequestObjectFactory.createNew(filename,
canceler));
     $http({
       method: 'GET',
       responseType: 'blob',
       url: _getAudioDownloadUrl(explorationId,
filename),
       timeout: canceler.promise
     }).success(function(data) {
```

- I will use `setTimeout()` function. It creates a timeout timer end emits a **Timeout-event** whenever there is an error due to
    - No data read or write in the given timeout after connection. (when http request does not succeed). In the above http request timeout is set by using promise.

  That Timeout-event occurrence can be used to show the try again button.

BreakDown

Date	Work to be done
13 May - 21 May	PR1 -- ● Create `AssetsBackendApiServiceForImage` ○ Serves as an **interface to fetch and upload the images from the DataStore** ➔ Create `AssetsBackendApiServiceForImageSpec.js` ○ Includes **tests** for the `AssetsBackendApiServiceForImage`. ● Rename `AssetsBackendApiService` to `AssetsBackendApiServiceForAudio` ○ Serves as an **interface to fetch from and upload the audio to Datastore or GCS**. ➔ Rename `AssetsBackendApiServiceSpec.js` to `AssetsBackendApiServiceForAudioSpec.js` ○ Includes **tests** for the `AssetsBackendApiServiceForAudio`. ● Include the above services in the files where they are required This `AssetsBackendApiServiceForImage` and `AssetsBackendApiServiceForAudio` **are used by the preloading services for downloading and caching the image and audio respectively**.

22 May - 30 May	PR2 -- ● Create `ImagePreloaderService.js` 　○ Service to **preload the images in the AssetsBackendApiServiceForImage's cache**. ● Create `ImagePreloaderServiceSpec.js` 　○ Includes **tests** for the `ImagePrelaoderService` ● Include the above service in the files where it is required. The **images will be preloaded and cached after completion of above two PRs** (PR1 and PR2).
31 May - 5 June	PR3 -- ● Add a **loading indicator gif**. ● Add a **try again button** ● Create `ImageDisplayService.js` 　○ This handles the displaying of the images in the exploration. 　○ If an image is being loaded 　　■ A **loading indicator** will be shown. 　○ else if timeout occurs 　　■ show **try again button** 　○ else 　　■ The **image** is displayed. ● Include the above service in the files where it is required ● Implement the tests
6 June - 9 June	**Buffer for milestone 1**

MileStone 2

❖ Implement one-off job and write tests for it.

❖ Edit the ImageUploadHandler such that the images get uploaded to only GCS in production mode, and to only App Engine Datastore in dev mode. Write tests to check it.

- Edit AssetsBackendApiServiceForImage so that the images are fetched from the GCS in production mode and from App Engine Datastore in dev mode. Write tests to check the same.

One-off Job

- Before implementing the One-off-Job, implement
  - `isfile(self,filepath)` in the `GcsFileSystem` in the `fs_domain.py`
    - It checks if the file with the filepath given in the argument, exists in the GCS under the current exploration.
    - i.e there exists an instance with id `'*/exp_id/assets/filepath'` in GCS

- We will have to do the migration of the existing image data in the App Engine Datastore to Google Cloud Storage.

- Procedure :-
  The job will take each `FileMetadataSnapshotModel` instance from existing schema.
  - If the instance corresponds to the storing of image files then
    - Extract the filename, exp_id from the snapshot_id.
    - If the file which is a part of the given exploration exp_id does not exist in GCS [use the isfile(filepath) in the `GcsFileSystem`] then

      This means that the file does not exist in the GCS and hence we can add it.

      - use the snapshot_id, fetch the corresponding FileContentSnapshotModel to get the content of the file.
      - Create a file using the `GcsFileSystem`, make the commit for saving the file to GCS.

- The `FileContentSnapshotModel` / `FileMetadataSnapshotModel` instance has the id of the form :-
  (snapshot_id)

```
'exp_id/assets/filename-vI'
```

Term	Meaning
exp_id	id of the exploration
vI	the version number of the file

➔ The file (created using the `GcsFileSystem` in the `fs_domain.py`) will be stored in GCS as

```
'<bucket>/<exploration-id>/assets/<filepath>'
<filepath> = 'images/filename'
```

➔ NOTE

◆ the version number of the files doesn't matter, since we don't allow editing of the images or the audio files. Same is the case with the audio files, they are being added to the GCS without any version number.

➔ Extracting filename (the filename without its version) out of the snapshot_id, for checking the valid image type.

- Use get_unversioned_instance_id() of `FileMetadataSnapshotModel` to get the instance_id from snapshot_id.

```
instance_id = exp_id/assets/filename
```

➔ **Code**

Create `image_data_migration_jobs_one_off.py`

Create class `ImageDataMigrationJob` (jobs.BaseMapReduceJobManager)

- entity_classes_to_map_over
    - returns all from `FileMetaDataSnapshotModel`
- map(item)
    - If the instance corresponds to valid image type extensions then
        - Extract the exp_id, filename from the instance_id
        - If there does not exist an instance in GCS with the id `'<bucket>/exp_id/assets/images/filename'` then

          This checks that even if job is run again it won't copy the data which is already there in GCS.

            - Query the FileSnapshotContentModel with the same snapshot_id from App Engine Datastore. Get the image content from it.

              Now we have instance.user_id and the content

              ```
              fs = fs_domain.AbstractFileSystem(
              fs_domain.GcsFileSystem(exploration_id))
              fs.commit(instance.user_id, '%s/%s' %(
              'images' , filename, content)
              ```

              This basically adds the file to the GCS, it is similar to how the audio files are added to GCS in `editor.py.`
              It will be stored in GCS with id as `'<bucket>/<exploration-id>/assets/<filepath>'`

              Where `<filepath> = 'images/filename'`
- reduce(user_id, values)
    - pass

Register job in `jobs_registry.py`

NOTE

- There is another way of doing the data migration i.e by extracting the images from the rich text content of an exploration
    - This method would be specifically to an exploration.
    - We will use this method later for testing one off job (migrating the image data from App Engine Datastore to Google Cloud Storage).

- Why the method used, over the method which extracts the images from the rich text content of an exploration?
    - The method used considers the instances of all the image files that have been uploaded and adds them to the GCS.
    - This covers all the images that had been uploaded (added to an exploration), so we don't have to worry about the images in the different versions of an exploration.

➔ Testing :-
- Create `Image_data_migration_jobs_one_off_test.py`
    - In the one off job above, we added the files to the GCS by getting all the FileSnapshotModel (of image type) instances.
    - We can test it the other way around. The exploration which has some images to display will be used for testing. Image from an exploration has to be there in the GCS.
    - We have an exploration id and the current version. For all versions of the exploration get the image filenames, check if the same exist in the GCS.

    This ensures that whether the image files of all the exploration with all their versions are being transferred to the GCS or not.

Edit ImageUploadHandler

➔ The ImageUploadHandler in controllers/`editor.py` handles the uploading of the image to the App Engine Datastore.

➔ Since it uses `ExplorationFileSystem` ("a datastore-backend read write file system for a single exploration"), the image gets stored to Datastore in both cases -- dev and production mode.

- ➔ controllers/editor.py
  - ❖ In the class ImageUploadHandler
    - ➢ Line 852 - 853

```
fs = fs_domain.AbstractFileSystem(

fs_domain.ExplorationFileSystem(exploration_id))
```

We need to make the image upload handling in such a manner that images get uploaded to only GCS in production mode, and to only App Engine Datastore in dev mode.

This will be similar to what we have for audio in controllers/editor.py

- ❖ In the class AudioUploadHandler
  - ➢ Line 941 - 944

```
 file_system_class = (
            fs_domain.ExplorationFileSystem if
feconf.DEV_MODE
            else fs_domain.GcsFileSystem)
        fs =
fs_domain.AbstractFileSystem(file_system_class(exploration
_id))
```

  - ➢ Line 945 - 948

```
fs.commit(
            self.user_id, '%s/%s' % (self._FILENAME_PREFIX,
filename),
            raw_audio_file, mimetype=mimetype)
```

This calls the commit function of the file system used (either ExplorationFileSystem or GcsFileSystem).

self._FILENAME_PREFIX is ‘audio’ here. In case of the ImageUploadHandler it will be set to ‘images’.

➔ In the `resources_test.py` there is a class `ImageHandlerTest` which already tests the uploading and downloading of the images.

  Why use the already existing `GcsFileSystem` for storing images in GCS?

  - Because the audios are already being stored to the GCS using the GcsFileSystem.

  - domain/fs_domain.py
    - A file created using the `GcsFileSystem` will get stored to the GCS, since its commit function adds the file to the GCS.
    - Line 480 - 497
      - The commit function of the `GcsFileSystem`.

```
def commit(self, unused_user_id, filepath, raw_bytes,
mimetype):
       """Args:
           unused_user_id: str. Unused argument.
           filepath: str. The path to the relevant file
within the exploration.
           raw_bytes: str. The content to be stored in the
file.
           mimetype: str. The content-type of the cloud
file.
       """
       bucket_name =
app_identity_services.get_gcs_resource_bucket_name()

       # Upload to GCS bucket with filepath
       # "<bucket>/<exploration-id>/assets/<filepath>".
       gcs_file_url = (
           '/%s/%s/assets/%s' % (
               bucket_name, self._exploration_id,
filepath))
       gcs_file = cloudstorage.open(
           gcs_file_url, 'w', content_type=mimetype)
       gcs_file.write(raw_bytes)
       gcs_file.close()
```

Edit AssetsBackendApiServiceForImage

- ➔ The service will be similar to the existing `AssetsBackendApiService` which currently serves as an interface for fetching and uploading the audio files from either the Datastore or GCS.

- ➔ templates/dev/head/services/`AssetsBackendApiService.js`
    - ◆ Line 30 - 34

```
var AUDIO_DOWNLOAD_URL_TEMPLATE = (
     GLOBALS.GCS_RESOURCE_BUCKET_NAME ?
     ('https://storage.googleapis.com/' +
GLOBALS.GCS_RESOURCE_BUCKET_NAME +
      '/<exploration_id>/assets/audio/<filename>') :
     '/audiohandler/<exploration_id>/audio/<filename>');
```

- ◆ In case of `AssetsBackendServiceApiForImage`, image download url template will be assigned the value such that it later fetches from:-
    - GCS --- In production mode
    - App Engine Datastore --- In dev mode

BreakDown

Date	Work to be Done
10 June - 24 June	PR1 -- ● Implement one off job `(image_data_migration_jobs_one_off.py)` ○ For **migrating the image data from App Engine Datastore to GCS**. ● Create `image_data_migration_jobs_one_off_tests.py` ○ The **tests** for the above one-off job. ● Additional testing by deploying my version of Oppia
25 June - 30 June	PR2 -- ● Edit the ImageUploadHandler in `editor.py` ○ The **images will be uploaded to GCS in production mode and to App Engine Datastore in dev mode**. ■ This is similar to AudioUploadHandler ● Write **tests** to check that images get uploaded to GCS. ● Additional testing by deploying my version of Oppia
1 July - 4 July	PR3 -- ● Edit the `AssetsBackendApiServiceForImage`.js ○ The **images will be fetched from GCS in production mode and from App Engine Datastore in dev mode**. ● Write **tests** to check that the images are fetched from the GCS. ● Additional testing by deploying my version of Oppia
5 July - 10 July	Buffer for milestone 2

ONE OFF JOB SHOULD BE RUN IN JULY'S RELEASE (around 15th).

Points supporting the above breakdown of tasks.

- Adding the above PRs (Milestone 2) in the specified order won't affect the develop branch because
    - the ["PR1"] migration job will be run only during release.
    - the ["PR2"] uploading and ["PR3"] fetching of images, in the develop mode, will still be from the App Engine DataStore.
    - The GCS system would come into effect after the migration --- release time (when the code gets into the production).

- All the above PRs would be merged in the given order at least a week before the release date(15th usually). So there won't be a problem in cutting a release from develop branch.

- Also, I will be testing them by deploying my branch version of Oppia (similar to production mode). So all of the PRs --- work data migration, changing uploading system of images, changing fetching system of images will each be merged only after this "deploying my branch" testing.

- Keeping in mind the importance of the above PRs I have kept buffer time of 5 days.

- After getting them merged, and running the one-off job:-
    - Existing images will be copied from App Engine Datastore to GCS.
    - Image will get uploaded to and fetched from GCS.

Pitfalls that can happen	Reason why they won't
Incomplete transfer of files from the old system to the new system	● Since one off job creates an instance in GCS corresponding to every image found in the App Engine Datastore. ○ Images are stored in App Engine Datastore as the FileContentSnapshotModel instance and its corresponding `FileMetaDataSnapshotModel` instance

	Therefore, the incomplete transfer of files from the old system to the new system is not possible. ● Moreover the one off job tests ensure that all the images from an exploration are there in the GCS.
New files get written to both systems at once	● No, the files get written to only App Engine Datastore in dev mode and to only GCS in production mode. ● The tests in controllers/`resources_test.py` check that the images are uploaded to the either GCS or dev. It's not possible that the files get written to both the systems.
New files don't get written to any system at all	● As explained above, the file gets written to either App Engine Datastore or GCS based on the conditions specified(whether its dev mode or production mode). It's not possible that the image files don't get written to any system at all.

Milestone 3

❖ Write a one off job to remove the images from the App Engine Datastore.

❖ Refactor the code, i.e safely remove the code that relates to old system.

❖ Compress images automatically when large images are uploaded to the server.

One off job

- ➔ The deprecation of the old system requires deletion of the images from the App Engine DataStore. So this one off job deletes the images from the App Engine Datastore.

- ➔ Procedure :- The job will take each `FileMetadataSnapshotModel` instance from existing schema.
  - If the instance corresponds to the storing of image files then
    - Query the FileContentSnapshotModel with the same snapshot_id.
    - Delete both the instances (`FileMetadataSnapshotModel` as well as `FileContentSnapshotModel`)
    - Deleting here means setting the deleted property of the instance to true (which is false by default)

- ➔ Code
  - ◆ Create `delete_image_from_datastore_jobs_one_off.py`
    - Create class `DeleteImageFromDatastoreJob`(jobs.BaseMapReduceJobManager)
      - entity_classes_to_map_over
        - returns all from `FileMetaDataSnapShotModel`
      - map(item)
        - If the instance corresponds to valid image type extensions then
          - Query the FileContentSnapshotModel with the same snapshot_id from App Engine Datastore and set the `instance.deleted = true.` This deletes the instance corresponding to FileContentSnapShotModel
          - `item.deleted = true.` This deletes the instance corresponding to FileMetaDataSnapShotModel
          - `item.put(), instance.put()`
      - reduce(key, stringified_values)
        - pass
    - Register job in `jobs_registry.py`

- ➔ Tests :- Create `delete_image_from_database_jobs_one_off_test.py`
  - Create a service that counts the number of images (which are a part of an exploration) in the App Engine Datastore

Term	Meaning
No_deleted	number of such images with the deleted set to true
No_not_deleted	number of such images with deleted set to false

  - Now run the job, since all the images(which are a part of an exploration) will have the deleted property set to true.
  - The [No_deleted + No_not_deleted] before running the one off job must be equal to No_deleted after running the job.
  - Other tests to be added along with the implementation of the one off job.

Refactor the code

Content to be added as the project proceeds.

Line no	What does it do?	Action to be taken	Why?

controllers/`resources_test.py`

43 - 61	It tests the uploading and downloading of the images.	Remove the downloading part of the tests	The downloading part will be tested in the `AssetsBackendApiServiceForImage`

controllers/`resources.py`

63 - 64	It tells that the ImageHandler class is for returning an image.	Replace the comment with the following text :- “It handles image retrievals only in dev”.	In case of production mode images will be fetched from GCS.

Compress images

- ➔ There is library called pillow. **Pillow** is a fork of the the Python Imaging Library, which builds on PIL by adding more features and support for Python 3. It can be used to compress images without losing much quality. I have gone through multiple references and found some of them as relevant and efficient.(Reduce image sizes without loss of quality, Image Compression).
  - It supports different file formats, such as `PNG, JPEG, GIF, PPM, TIFF and BMP`.
  - The primary types we are concerned with are `PNG, JPEG, JPG, GIF`.
    - This is so because the allowed image formats and extensions in the uploading of images are these 4 only.
    - In ImageUploadHandler class in editor.py
      Line 823 - 824

```
allowed_formats = ', '.join(

feconf.ACCEPTED_IMAGE_FORMATS_AND_EXTENSIONS.keys())
```

    - `feconf.py`
      Line 213 - 217

```
ACCEPTED_IMAGE_FORMATS_AND_EXTENSIONS = {
    'jpeg': ['jpg', 'jpeg'],
    'png': ['png'],
    'gif': ['gif'],
}
```

- ➔ Currently in Oppia, user is allowed to upload the images with size less than 1 MB.

- ➔ In controllers/`editor.py`
  - Implement the code to include the feature that checks the size of the image file uploaded, if the image size is greater than or equal to the SIZE_LIMIT (1 MB) then
    - it compresses the image automatically and then checks the size of the compressed image.

            - if size of compressed image < 1 MB then
              uploads the image to the GCS or App Engine Datastore.
            - else
              Tells the user to upload image with smaller size than the one uploaded user currently uploaded.
- ➔ In controllers/`resources_test.py`
    - Implement the code that uploads the image of size > 1 MB and checks that the file gets compressed or not.

- ➔ The maximum size of the image file to be uploaded will be such that after performing compression, the size of the compressed file is less than 1 MB.

BreakDown

Date	Work to be Done
12 July - 22 July	PR1 -- ● Implement one-off job (`delete_image_from_datastore_jobs_one_off.py`) ○ **Deletes the images in the app engine datastore**. ● Write **tests** for the above one off job. (`delete_image_from_datastore_jobs_one_off_tests.py`)
22 July - 31 July	PR2 -- ● **Refactoring** the code
1 August - 10 August	PR3 -- ● Edit `editor.py` ○ to include the feature to **automatically compress the image above the SIZE_LIMIT** ● Implement **tests** in the controllers/`resources_test.py` ○ Write tests to check that the images get compressed automatically.
11 August - 15 August	Buffer for milestone 3

Summer Plans

Time Zone

IST (India Standard Time)

How much time will you be able to commit to this project?

During May to July, I will be able to spend 7-8 hours a day, 6 days a week i.e roughly 40-48 hours a week. I'll make sure to put at least 40 hours a week during this time. After that (i.e during August) I will be able to spend 4-5 hours a day 7 days a week i.e 28-35 hours a week.

What jobs, summer classes, and other obligations might you need to work around? Please be upfront about any existing commitments you may have.

Our college has vacations from May to July so I have no commitments during that time. My classes begin from 3rd August (* can change, will update accordingly).

Communication

What is your contact information, and preferred method of communication?

E-mail : aashishgaba097@gmail.com, aashish.gaba@students.iiit.ac.in
Mobile no. : +91-8437740902
Github handle (gitter) : ishucr7

Oppia is very active on **gitter**, so preferred method for most of the communication and meetings will be gitter. I will maintain daily devlogs as recommended by mentors to document the daily work.

How often do you plan on communicating with your mentor?

We'll remain in touch over gitter(or Hangouts) twice a week to discuss the workflow to be followed or whenever I need advice.