Google Summer Of Code

2021 Proposal

Integrating the Oppia blog with Oppia.org

by Rijuta Singh

ABOUT ME

Why am I interested in working with Oppia, and on this project?

I've always been passionate about giving back to society and have felt deeply that it is one of my duties to do my personal best to be able to help those who have been less privileged. From a very young age, I had decided to do something for underprivileged children around me so that they at least get to study and get educated. I believe I am being educated to enable and empower those children.

I found Oppia while going through the list of GSoC organizations and it instantly drove my attention as I read about the website more and thought to myself that this is what I have been waiting for.
Oppia is a community of learners and teachers to help everyone learn whatever they want effectively and enjoyably. Oppia focuses on making this world a better place. It provides a platform to gain and share knowledge without any hurdles. And thus it was my place.

To add to this, the work environment in Oppia also motivates me to learn and contribute. Mentors and Leads are always available to help and review my work whenever I am stuck. Contributing to Oppia in the last few months has boosted my technical knowledge and has also taught me the importance of working in a team and helping fellow contributors. I would want to continue to contribute to Oppia even after the GSoC period ends for the reasons it serves and the environment it provides to its contributors and users.

I am interested in this project since it's a full-stack project which aims to create blogging functionality within the Oppia web base. This project will thus not only make the website richer with its content but will also allow an easy mode of sharing experience for all its contributors, learners, and volunteers with the world directly from the Oppia Website. This project involves writing code on both the frontend and backend and also provides me an opportunity to participate in the design process. Thus, this project promises to give me a full-stack dev experience and also makes me cherish how the addition of services and data models in the oppia codebase can be used to allow blogging and make such a great impact on the website. I would like to make this functionality as smooth as possible and provide an easy go way to its users.

Prior experience

It's not been long since I first heard about open source and web development. The COVID'19 lockdown that came into force in March 2020 made a 2nd-year Mechanical Engineering student try her hands on learning something new. The very philosophy of open source fascinates me and contributing to it is now becoming my passion, primarily because it feels nice to be surrounded by a bunch of like-minded people!

It was only in April 2020 that I started learning web development along with some competitive programming in Python. Since then I have given some contests on CodeChef, CodeForces, and Leetcode to get my hands on Python. I have also completed a series of courses on python using Coursera such as:

- *Python Data Structures: link*

- *Crash Course on Python by Google: link*
- *Using Python to interact with the Operating System by Google: link*
- *Create Your First Chatbot with Rasa and Python: link*

And a few more like 'python and openCV' etc…

Oppia was my first open-source organization, I started contributing to. And since then I have been working with Oppia as a member of the **LACE team** and have more than 20 successfully merged PRs**.**Most of these PRs are concerned with work in Typescript, Angular, AngularJs, and HTML/CSS.
It was difficult in the beginning but as time passed, I understood the code structure of Oppia, went through tutorials on Angular, searched whatever I could not understand in Oppia's codebase. Now I feel I have a good command over it. I have become much familiar with the Frontend karma and E2E tests. I am now able to help my fellow mates in the team on solving issues encountered by them while writing the karma tests.

Some of the PRs I have raised are:
1. The new Teach Page which is a part of the integration of Oppia.org with the Oppia Foundation Website:
    1.1. #11467
    1.2. Some responsiveness bugs were found later in the page and were fixed in :#12038
2. Adding creator-guidelines page: #12070
3. Migrating Splash page to Angular from Angular js: #11826
4. Solving question Editor Issue: #12308
   All other PRs can be found at Submitted PRs

I have also raised some issues (a few of which were raised during release testing):
1. *#11750*
2. *#11912*
3. *#11715*

Contact info and timezone(s)

Timezone: I will stay in India throughout the summer. The time zone will be Indian Standard Time (GMT+5:30)
Contact:
Mail: rijuta_s@me.iitr.ac.in
Mobile no. : +91- 8699815957 (Whatsapp)
Github Profile: Rijuta-s

Time commitment

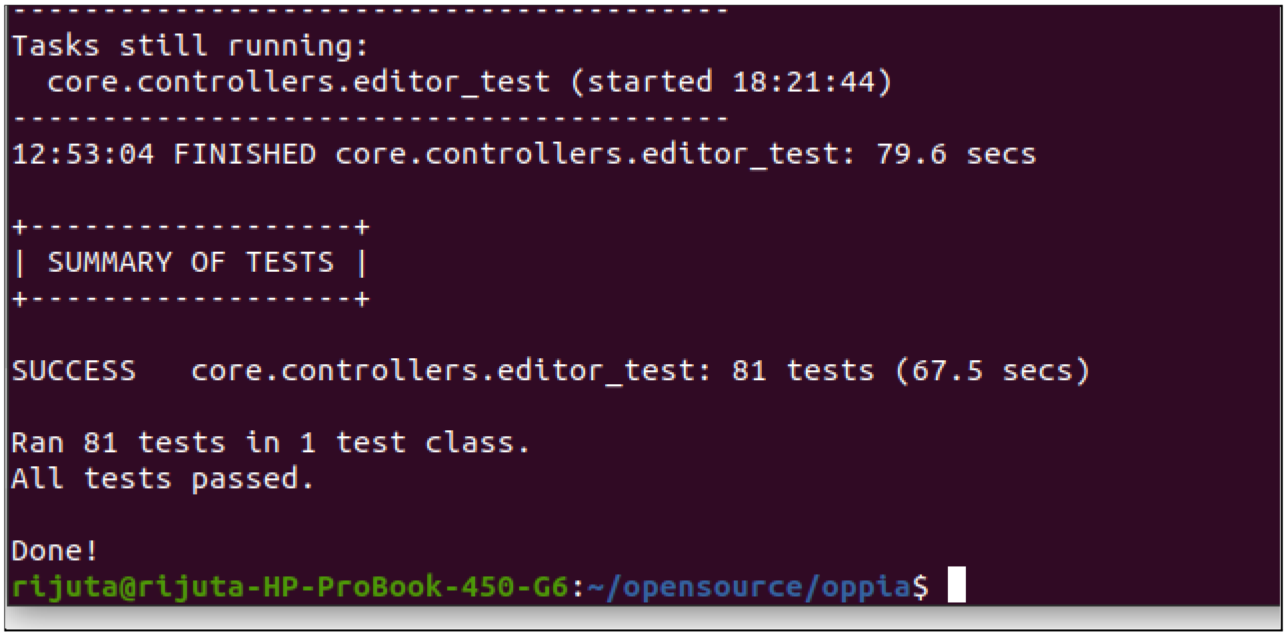
I will be working on the project from the first day itself i.e June 7.
I will be having my summer vacations from May 29 till August 2.
From June 7 to August 2: Working hours:- [9 AM - 7 PM]--[10 PM to 1 AM]
From August 2 to August 18: Working hours:- [6 PM to 9 PM] -- [10 PM--2 AM] weekdays.

On average, I will be able to devote 6-7 hours per day. I may have some other engagements on the 3rd and 4th Saturdays of the month(mentioned in summer obligations) reducing the working hours to maybe around 4-5 hours on Saturdays. Other than that as the college reopens on 2nd August, my working hours will reduce to 5-6 hours per day. Also, we do not have much work at the beginning of the semester, and to meet the deadlines, I will increase my working on weekends to 8-9 hours after the college reopens.

Essential Prerequisites

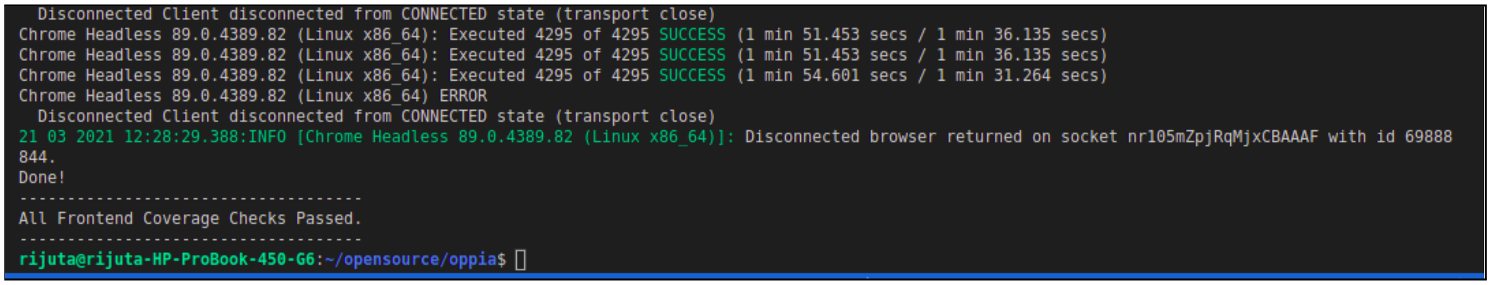
Answer the following questions:

- *I can run a single backend test target on my machine. (Show a screenshot of a successful test.)*



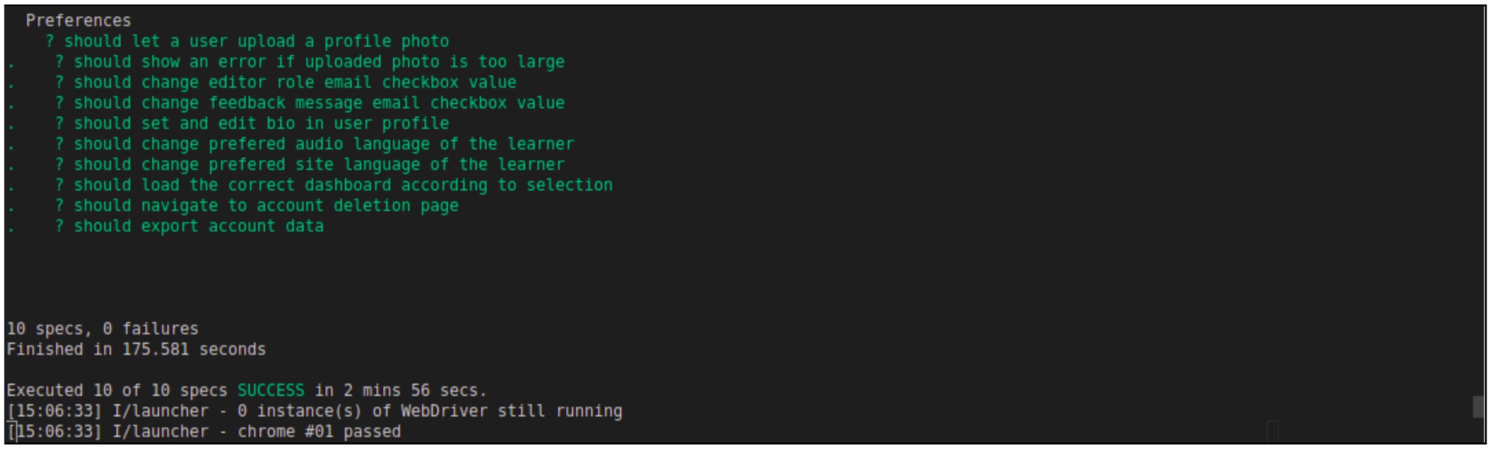
```
----------------------------------------
Tasks still running:
  core.controllers.editor_test (started 18:21:44)
----------------------------------------
12:53:04 FINISHED core.controllers.editor_test: 79.6 secs

+------------------+
| SUMMARY OF TESTS |
+------------------+

SUCCESS   core.controllers.editor_test: 81 tests (67.5 secs)

Ran 81 tests in 1 test class.
All tests passed.

Done!
rijuta@rijuta-HP-ProBook-450-G6:~/opensource/oppia$ 
```

- *I can run all the frontend tests at once on my machine.*



```
  Disconnected Client disconnected from CONNECTED state (transport close)
Chrome Headless 89.0.4389.82 (Linux x86_64): Executed 4295 of 4295 SUCCESS (1 min 51.453 secs / 1 min 36.135 secs)
Chrome Headless 89.0.4389.82 (Linux x86_64): Executed 4295 of 4295 SUCCESS (1 min 51.453 secs / 1 min 36.135 secs)
Chrome Headless 89.0.4389.82 (Linux x86_64): Executed 4295 of 4295 SUCCESS (1 min 54.601 secs / 1 min 31.264 secs)
Chrome Headless 89.0.4389.82 (Linux x86_64) ERROR
  Disconnected Client disconnected from CONNECTED state (transport close)
21 03 2021 12:28:29.388:INFO [Chrome Headless 89.0.4389.82 (Linux x86_64)]: Disconnected browser returned on socket nr105mZpjRqMjxCBAAAF with id 69888
844.
Done!
------------------------------------
All Frontend Coverage Checks Passed.
------------------------------------
rijuta@rijuta-HP-ProBook-450-G6:~/opensource/oppia$ 
```

- *I can run one suite of e2e tests on my machine.*



```
 Preferences
   ? should let a user upload a profile photo
.    ? should show an error if uploaded photo is too large
.    ? should change editor role email checkbox value
.    ? should change feedback message email checkbox value
.    ? should set and edit bio in user profile
.    ? should change prefered audio language of the learner
.    ? should change prefered site language of the learner
.    ? should load the correct dashboard according to selection
.    ? should navigate to account deletion page
.    ? should export account data



10 specs, 0 failures
Finished in 175.581 seconds

Executed 10 of 10 specs SUCCESS in 2 mins 56 secs.
[15:06:33] I/launcher - 0 instance(s) of WebDriver still running
[15:06:33] I/launcher - chrome #01 passed
```

Other summer obligations

I am involved in a research project under the guidance of my professor which might also require me to work during the GSoC period. Most of the work for it is done on the 3rd and 4th Saturday of the month. Thus on average, it requires me to give at most 6 hours per week only and thus I believe I should be able to manage it as well. Also, the working hours will get reduced substantially as we reach towards the end of the research statement in July. Apart from that, I will be having my summer vacation till August 2. As the normal college session resumes from August 2, I will be involved in lectures for roughly about 5 hours a day, 5 times a week. Thus for around the last 15 days in the GSoC period, I will have my college classes as per the current schedule released by the college administration. However, we do not have much work at the beginning of the semester hence I will be able to manage in that period as well.

Communication channels

Full name: Rijuta Singh
Preferred mode of communication: Hangouts, Mails, Discord, and Whatsapp
Email-Id: rijuta_s@me.iitr.ac.in
Mobile Number: +91-8699815957
Meeting with mentor: 2 times per week(flexible) [Google-Meet, Discord, or any other platform]
I will try to provide daily updates as much as possible to ensure smooth working and doubts clarification.
[I check my email regularly and can be contacted on any of the above-mentioned platforms]

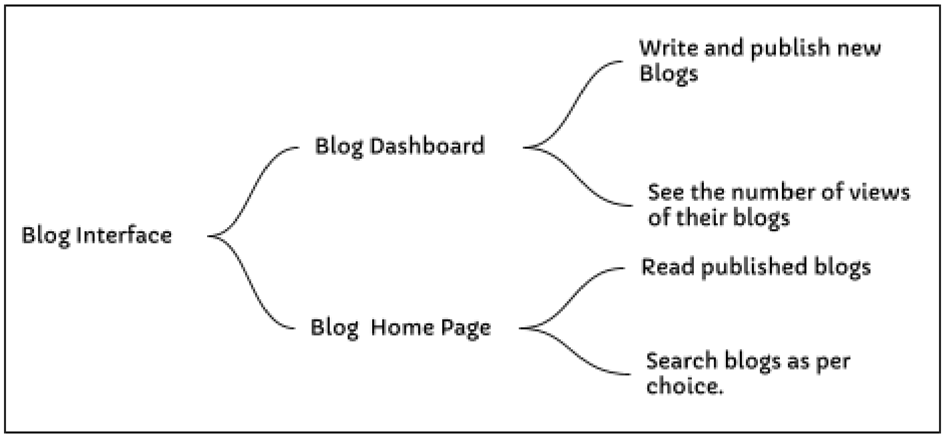
Application to multiple orgs

I will be submitting only one proposal - "Integrating the Oppia blog with Oppia.org"

Project Details

Product Design

Currently, Oppia.org's blogs are hosted on a separate site, Medium. I would like to create a new interface inside Oppia's website to directly enable the team members and users to share their stories with the world through it. There will be 2 parts of the interface -

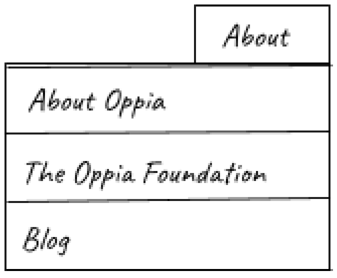

Blog Interface
Blog Dashboard
Blog Home Page
Write and publish new Blogs
See the number of views of their blogs
Read published blogs
Search blogs as per choice.

The link to complete page mocks: Link

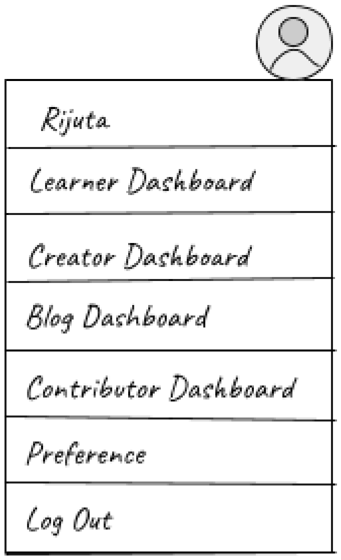
NOTE: UNDER THE GSoC TIMELINES ONLY BLOG DASHBOARD WILL BE COMPLETELY IMPLEMENTED. PRODUCT DESIGN CONTAINS BOTH HOMEPAGE AND DASHBOARD TO DEMONSTRATE THE FLOW.

There are two dropdowns from where the users can access the blog interface. If they aren't logged in or are not authorized to write a blog(**that is they are neither assigned the role of "blog_editor" nor are admins**), they can only go to the blog page by clicking on the "Blog" link inside the "About" dropdown menu (**future work).** If they are logged in and are also authorized to write blogs, then they can not only access the blog interface from the "About" dropdown but can also find the link to the "Blog Dashboard" from the "profile/Account" dropdown.

The users (**mostly members of the Oppia Team**) who are authorized to write the blogs ("blog editors" and "admins") can navigate to the Blog Dashboard by clicking on the "Blog Dashboard" option from the 'Account' drop-down menu.



The about drop-down menu



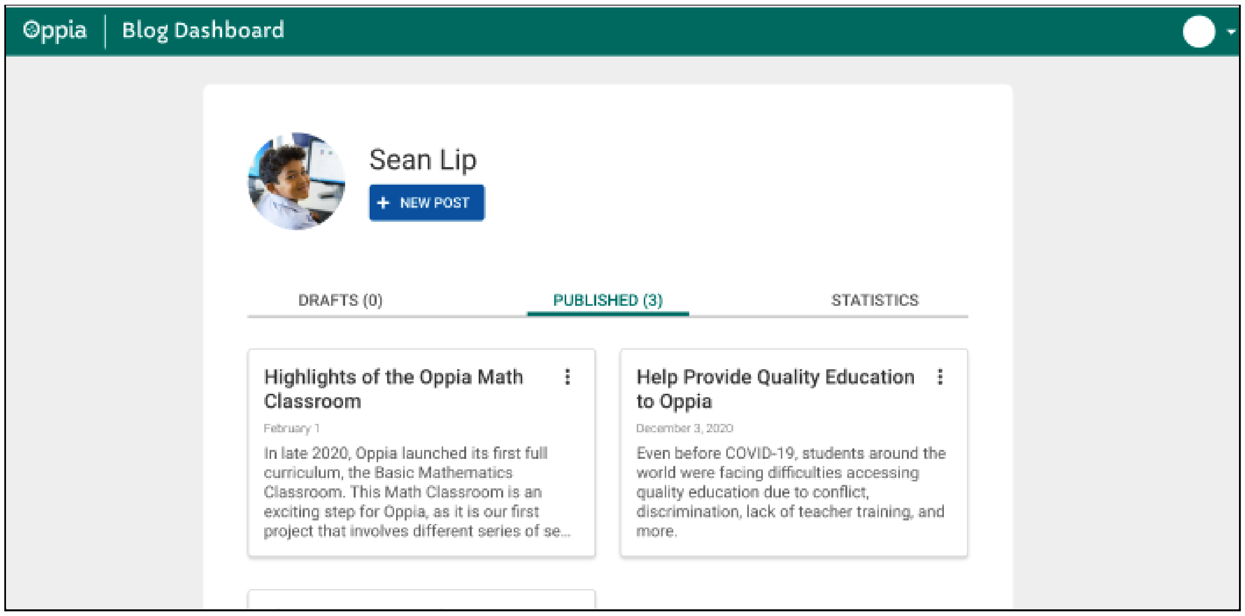
The profile drop down menu

Let us first walk through the "Blog Dashboard" :
(Only accessible to blog editors and admins)

THE BLOG DASHBOARD:

- Clicking on the Blog Dashboard link will take the user to the Main Page of the 'Blog Dashboard' Interface where their already published blog cards will be visible. They will find a "New Post" button which will take them to the blog editor interface, availing them to write a new blog. The user will also find links to "Drafts" and "Statistics" in the bar below the button. In case, they have not created any blog, a message "It looks like you haven't created any blog yet" will be displayed, where the button "Create your Blog" will take them again to the Blog Editor Interface.

- In the mock below, On clicking on Drafts, the same way cards (tiles of individual blogs) of unpublished (drafts) blogs will be displayed.



Highlights of the Oppia Math Classroom

February 1

In late 2020, Oppia launched its first full curriculum, the Basic Mathematics Classroom. This Math Classroom is an exciting step for Oppia, as it is our first project that involves different series of se...

Help Provide Quality Education to Oppia

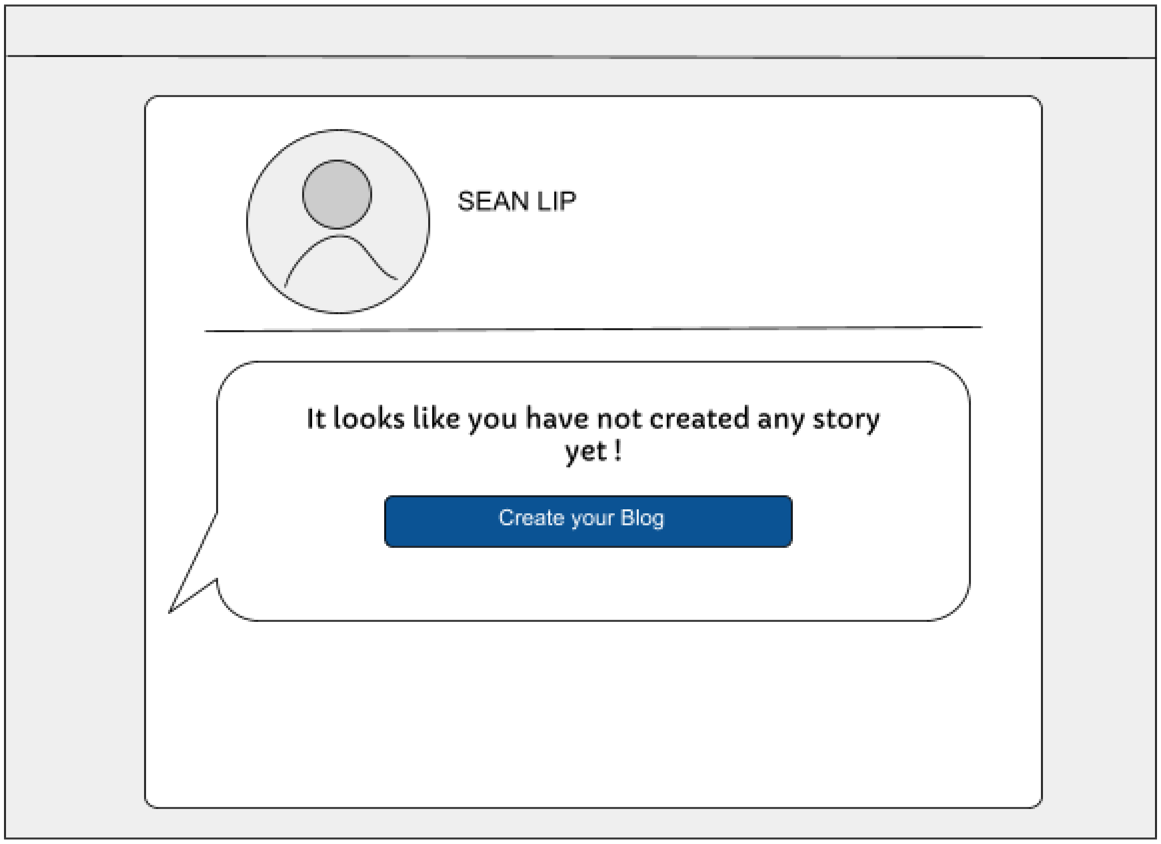
December 3, 2020

Even before COVID-19, students around the world were facing difficulties accessing quality education due to conflict, discrimination, lack of teacher training, and more.

The Blog Dashboard

- On clicking the 3 dots on the blog card- Dropdown with edit, unpublish and delete will appear. On clicking delete- a warning pop-up for confirmation of action will appear.

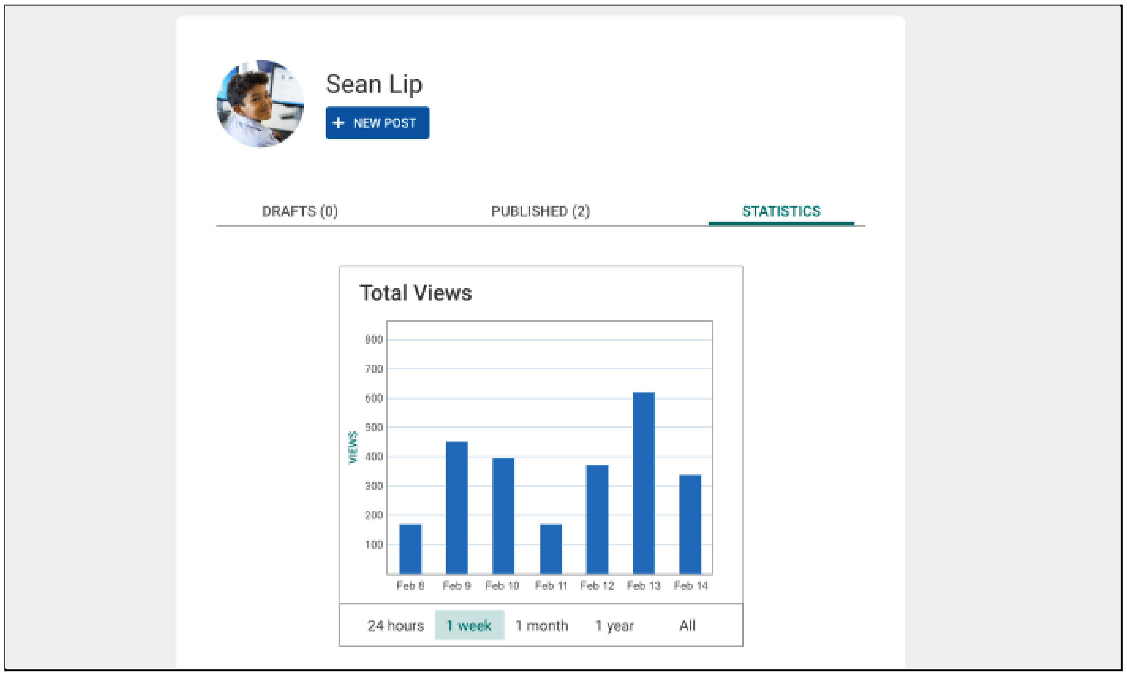
The dashboard will have the message box and no "new-post" button if the user has not created any blogs yet. Except for these two changes, the wireframe mock-up will resemble the above dashboard mock.



The first look of the Blog Dashboard if the user did not create any blog

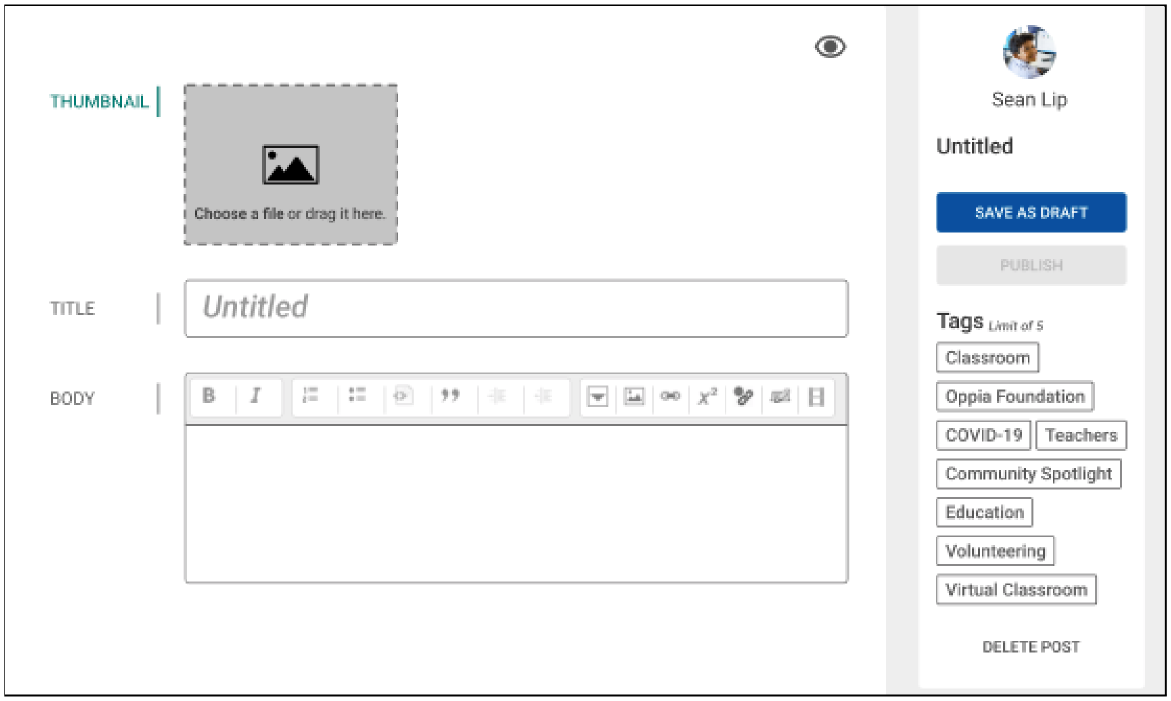
*** NOTE : Frontend and backend view of Statistics tab will be implemented afterward and is detailed in "future works" .**

- On clicking "Statistics", a bar graph representing the total number of views on their blogs collectively will be visible :



The Blog Dashboard showing statistics tab

- On clicking the "new post" button, the user will be taken to the Blog Editor interface :



The Blog Editor Interface

1. The ‘thumbnail’ option will allow the author to select the main image to be visible on the blog card on the blog page.

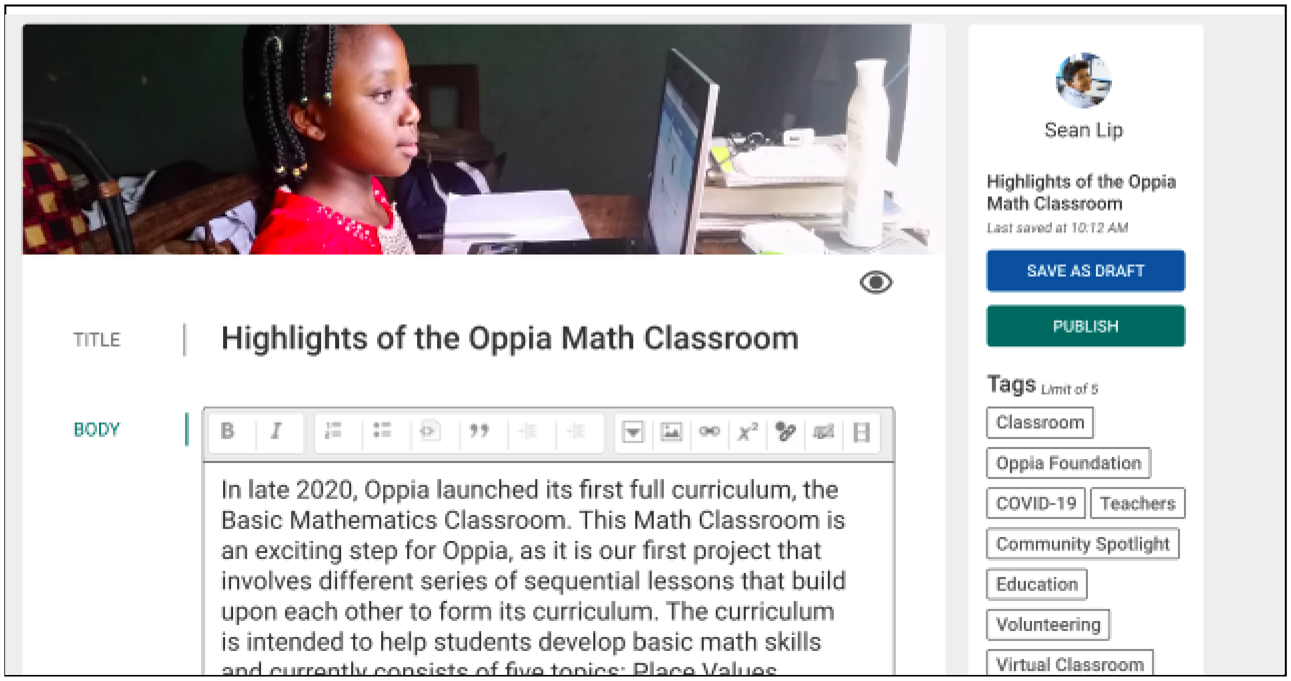
2. Then the blog can be deleted, saved as a draft, or published after making all the desired changes.

3. The blog editor will be visible in the body section of the blog. It will be the CKE-editor (Reasons for using CKE-editor), the base of Oppia’s RTE editor here. The editor will have its toolbar at the top with various plugins and features. All the features that will be provided are detailed in “The blog editor”.

4. Clicking on the “Eye Icon” will load the preview of the blog card which will be displayed on the blog page.

5. Here, the author can select up to 5 tags through which the blog can be **searched** later on the Blog Page.

    - Initially, the tags will be defined by **admins** in order to ensure that we have sufficient blogs on these topics. See adding tags for blog.
    - As soon as this is achieved, we will move onto user-defined tags which will be added by the user in the blog-dashboard itself. We will have an ‘add’ button below all the already defined tags to enable the user to add the tag.

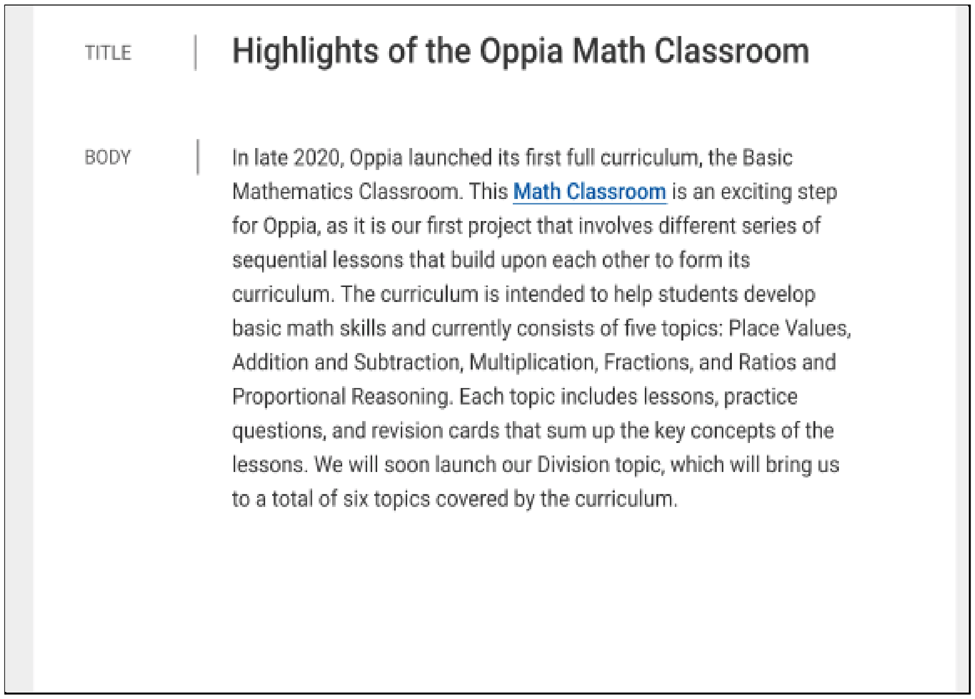


Highlights of the Oppia Math Classroom

In late 2020, Oppia launched its first full curriculum, the Basic Mathematics Classroom. This Math Classroom is an exciting step for Oppia, as it is our first project that involves different series of sequential lessons that build upon each other to form its curriculum. The curriculum is intended to help students develop basic math skills and currently consists of five topics: Place Values

The Blog Editor Dashboard with all content added

- After entering all the content of the blog and clicking outside the editor box, the editor will disappear and the content will appear like this (which will be a kind of preview of the blog page):



TITLE

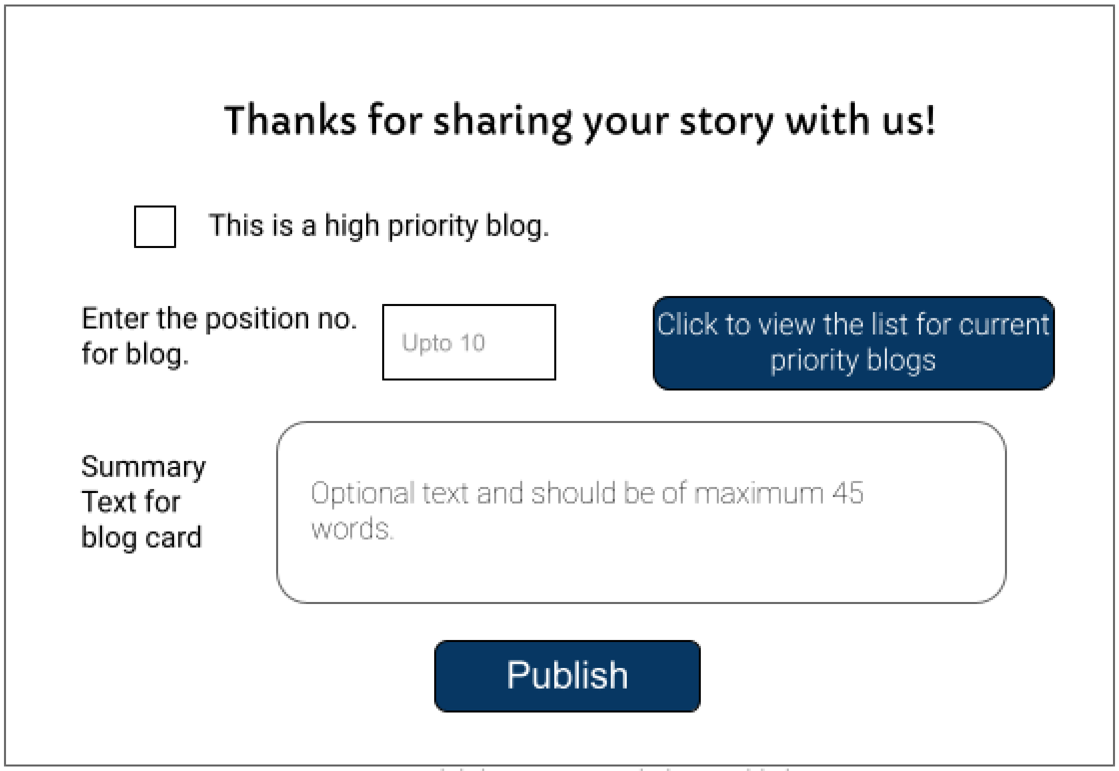
Highlights of the Oppia Math Classroom

BODY

In late 2020, Oppia launched its first full curriculum, the Basic Mathematics Classroom. This Math Classroom is an exciting step for Oppia, as it is our first project that involves different series of sequential lessons that build upon each other to form its curriculum. The curriculum is intended to help students develop basic math skills and currently consists of five topics: Place Values, Addition and Subtraction, Multiplication, Fractions, and Ratios and Proportional Reasoning. Each topic includes lessons, practice questions, and revision cards that sum up the key concepts of the lessons. We will soon launch our Division topic, which will bring us to a total of six topics covered by the curriculum.

- To publish a blog minimum of one tag is required and a thumbnail should be added. Along with that the title and minimum 50 words in the body should be added to enable the **"publish"** button.

- They can publish the blog after selecting at most 5 tags or can save it as a draft and can come later to finish it. If the user clicks on the "Publish" button, a confirmatory dialogue box will appear with a few fields:



Thanks for sharing your story with us!

☐ This is a high priority blog.

Enter the position no. for blog. | Upto 10

Click to view the list for current priority blogs

Summary Text for blog card | Optional text and should be of maximum 45 words.

Publish

UiB modal that opens on clicking publish

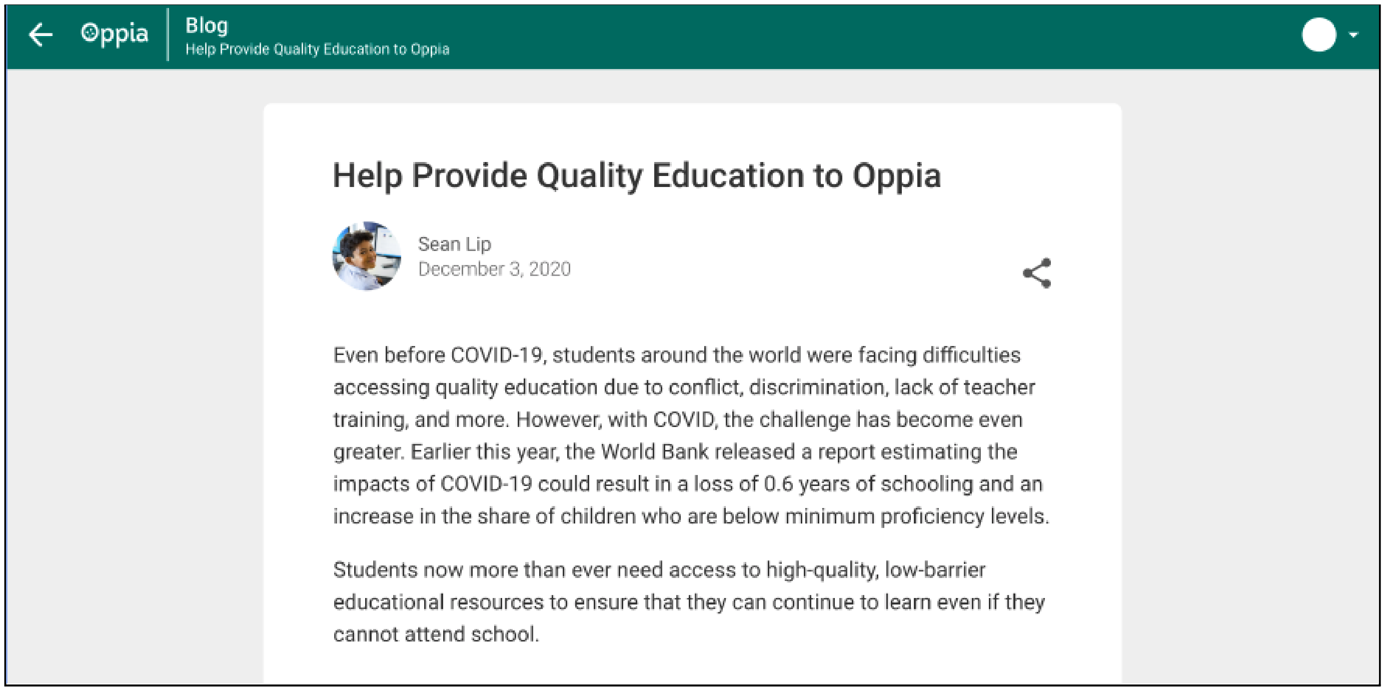
Displaying summary text will be necessary as not every blog's beginning and the title gives an insight into the actual content of the blog. It will be of at most 45 words. As such, all the blogs will appear in the order of the date published. However, we will definitely **not want** a few blogs such as those highlighting Oppia importance and its recent achievements to be not present on the top of the list and are shadowed by other blogs. There can be a maximum of **10 priority blogs.**

- To ensure such blogs always remain on the top of the list the check box "This blog is a priority blog" should be selected.
- **Priority blogs can only be authored by admins.**
  So only admins will be able to see the checkbox.
- This will make a dropdown menu visible which will contain the "titles" of the current high priority blog. And the input field where the position of the blog can be entered.

Clicking on the publish button in the dialogue box will take the author to the blog post page.

THE BLOG POST PAGE :

***NOTE:** [IMPLEMENTATION IS A PART OF FUTURE WORK(Added here to demonstrate the flow)]



Help Provide Quality Education to Oppia

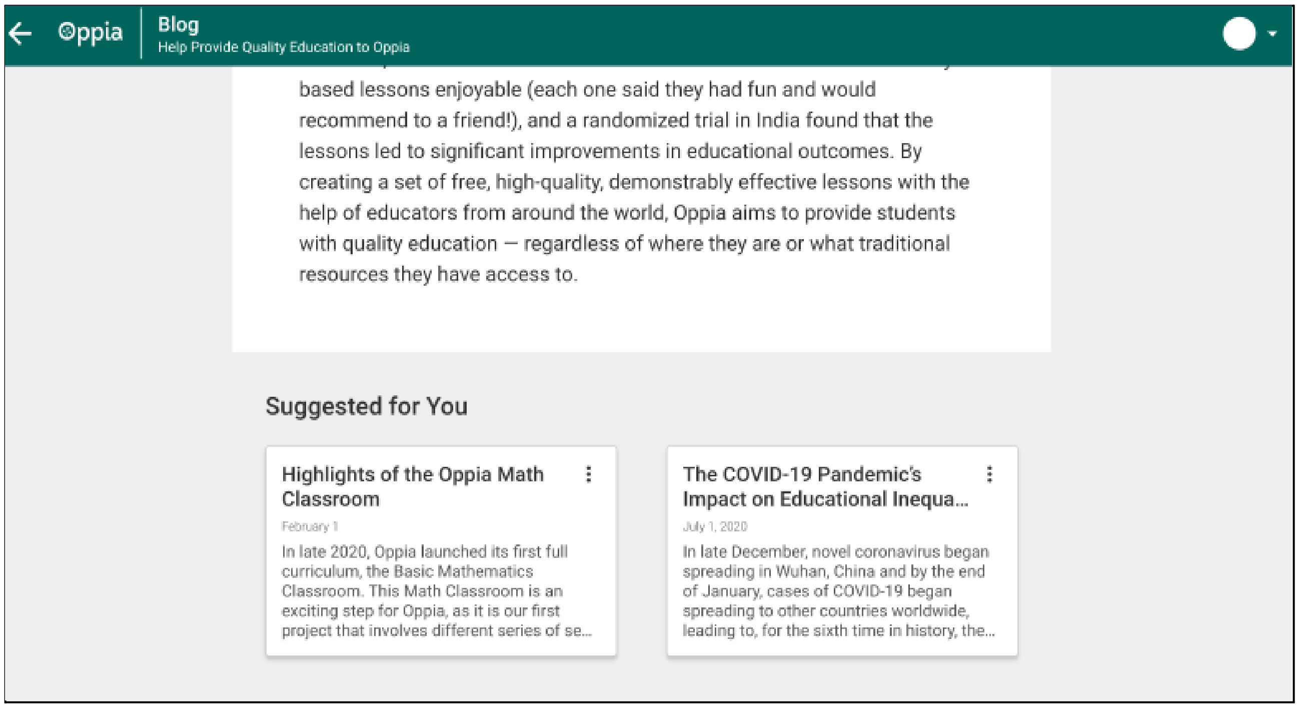
Sean Lip
December 3, 2020

Even before COVID-19, students around the world were facing difficulties accessing quality education due to conflict, discrimination, lack of teacher training, and more. However, with COVID, the challenge has become even greater. Earlier this year, the World Bank released a report estimating the impacts of COVID-19 could result in a loss of 0.6 years of schooling and an increase in the share of children who are below minimum proficiency levels.

Students now more than ever need access to high-quality, low-barrier educational resources to ensure that they can continue to learn even if they cannot attend school.

- Clicking on the share icon - "link of the blog" will be shown which can be copied then and shared.

- Also at the bottom, depending upon the tags and the author selected by the user, upto 2 recommendations of blogs will be shown.



Oppia | Blog
Help Provide Quality Education to Oppia

based lessons enjoyable (each one said they had fun and would recommend to a friend!), and a randomized trial in India found that the lessons led to significant improvements in educational outcomes. By creating a set of free, high-quality, demonstrably effective lessons with the help of educators from around the world, Oppia aims to provide students with quality education — regardless of where they are or what traditional resources they have access to.

Suggested for You

Highlights of the Oppia Math Classroom
February 1
In late 2020, Oppia launched its first full curriculum, the Basic Mathematics Classroom. This Math Classroom is an exciting step for Oppia, as it is our first project that involves different series of se...

The COVID-19 Pandemic's Impact on Educational Inequa...
July 1, 2020
In late December, novel coronavirus began spreading in Wuhan, China and by the end of January, cases of COVID-19 began spreading to other countries worldwide, leading to, for the sixth time in history, the...

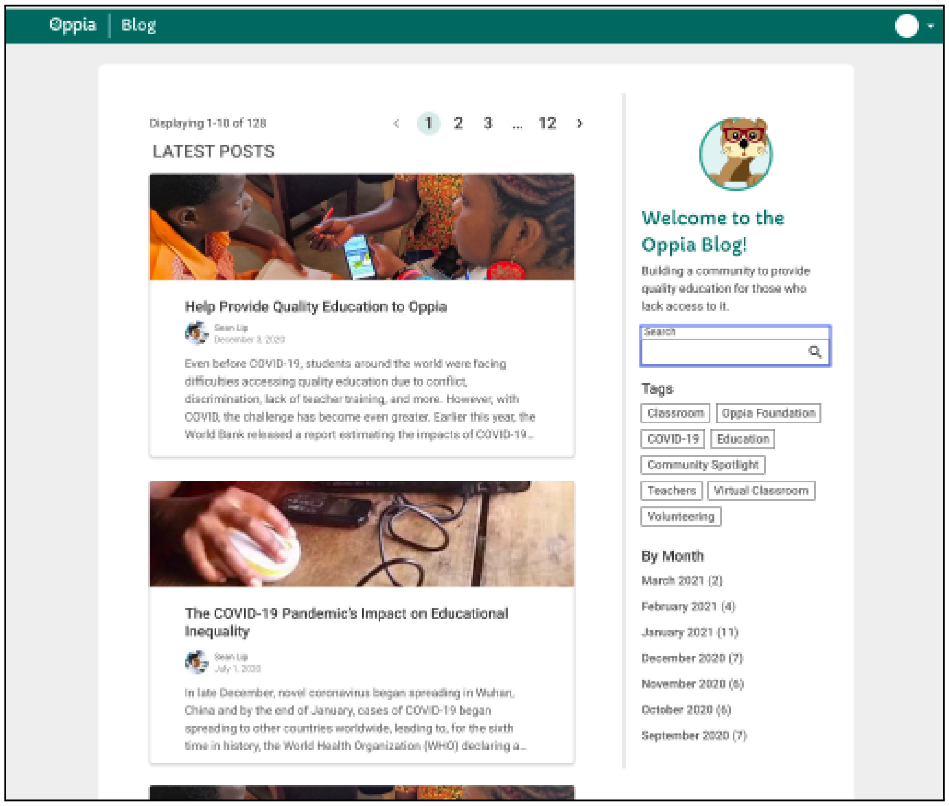
- Clicking on the back button will take the user to the Blog Dashboard

Lets now explore the blog home page:

THE BLOG HOME PAGE:

***NOTE:** [IMPLEMENTATION OF BLOG HOMEPAGE IS A PART OF FUTURE WORK (Added here to show the need of search handlers in the code base and to demonstrate the flow .)]

- Users can select the "blog" option from the 'About' drop-down menu. This will land the user on the Blog page:



Oppia | Blog

Displaying 1-10 of 128 ‹ 1 2 3 ... 12 ›

LATEST POSTS

Help Provide Quality Education to Oppia
Sean Lip
December 3, 2020
Even before COVID-19, students around the world were facing difficulties accessing quality education due to conflict, discrimination, lack of teacher training, and more. However, with COVID, the challenge has become even greater. Earlier this year, the World Bank released a report estimating the impacts of COVID-19...

The COVID-19 Pandemic's Impact on Educational Inequality
Sean Lip
July 1, 2020
In late December, novel coronavirus began spreading in Wuhan, China and by the end of January, cases of COVID-19 began spreading to other countries worldwide, leading to, for the sixth time in history, the World Health Organization (WHO) declaring a...

Welcome to the Oppia Blog!
Building a community to provide quality education for those who lack access to it.

Search

Tags
Classroom | Oppia Foundation | COVID-19 | Education | Community Spotlight | Teachers | Virtual Classroom | Volunteering

By Month
March 2021 (2)
February 2021 (4)
January 2021 (11)
December 2020 (7)
November 2020 (6)
October 2020 (6)
September 2020 (7)

1. The user can not only search the blogs from the author's name and the title but can also select tags or months to sort out blogs and get the list of required blogs.

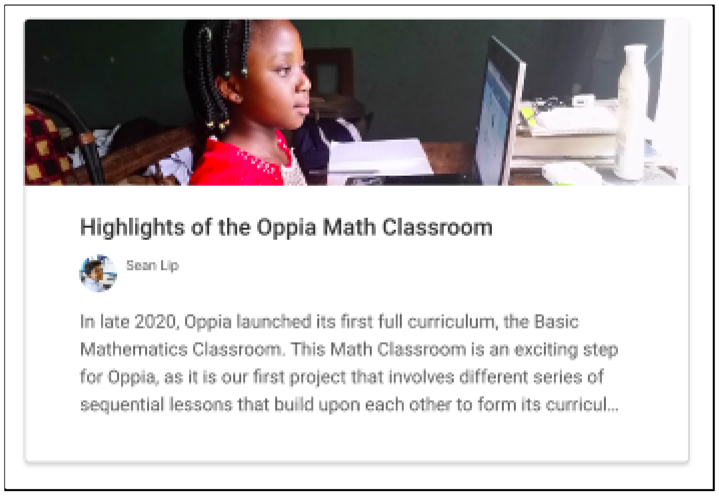
2. Blog Cards, which will be displayed on the page, will contain the author's name, date of publication, and a short summary of the blog.

3. A maximum of **10 blogs** will be displayed on a page.
   a. Clicking on the blog card will take the user to the blog post page (as shown above).
   b. Clicking on the author's name on the blog story page will again display a list of blog cards on the blog page written by the author.

4. "High Priority blogs" will occupy the top positions in case they are present. Instead of the heading being "Latest Post" it will be "Highlights" below which all priority blog cards will be displayed and after that below "Latest Post" all the cards in the order of their publishing date will appear.

THE BLOG CARD :

Each story will be represented on the Blog Homepage and Blog Dashboard by its "Blog Card".
1. It will contain the thumbnail of the blog.
2. The title of the blog.
3. The author's name and profile photo.
4. The short summary text describing the blog.
5. The date of publishing the blog.



Highlights of the Oppia Math Classroom

Sean Lip

In late 2020, Oppia launched its first full curriculum, the Basic Mathematics Classroom. This Math Classroom is an exciting step for Oppia, as it is our first project that involves different series of sequential lessons that build upon each other to form its curricul...

Clicking on the card will take the user to the "Blog Post Page"

THE BLOG EDITOR :

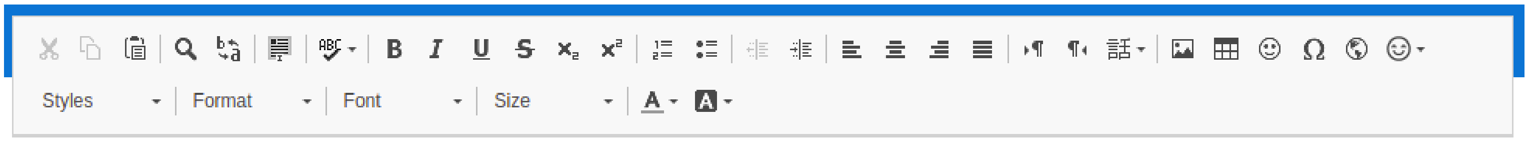
Blogs are more than general text articles. They depict experiences, emotions, and views that may not find words to be sufficient enough to communicate them well to the general users. Thus we will need features and plugins to help them.

Why CKEditor:

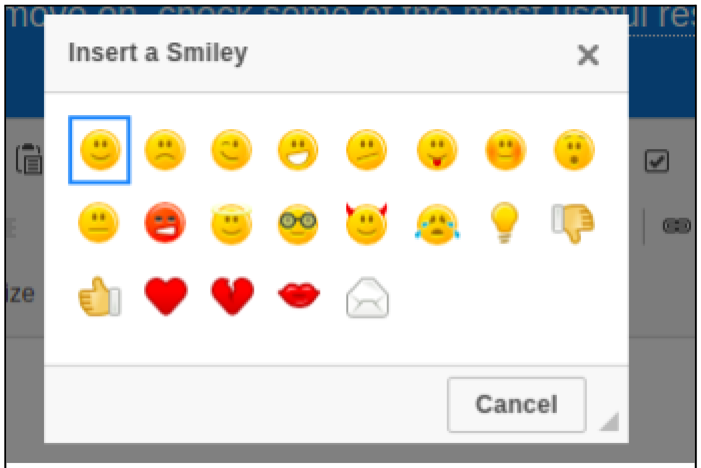
Currently, Oppia uses CKEditor version 4.12.1.

1. Below are some of the features that CKEditor offers to make the user journey while writing a blog, fun, easy and engaging. This will enable a user to produce blogs that are attractive and colorful to its readers.[Plugins are already available in Oppia's codebase but currently, some are removed from the build as they are currently **not** being used in **Oppia's RTE for explorations**].

Toolbar with some useful plugins enabled.

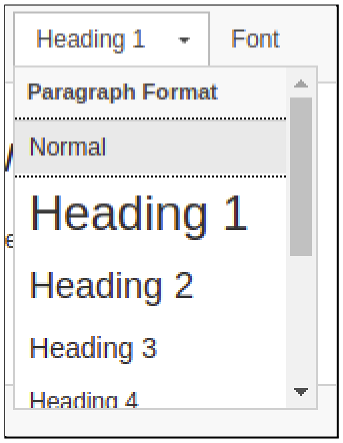


a. Inserting emojis:

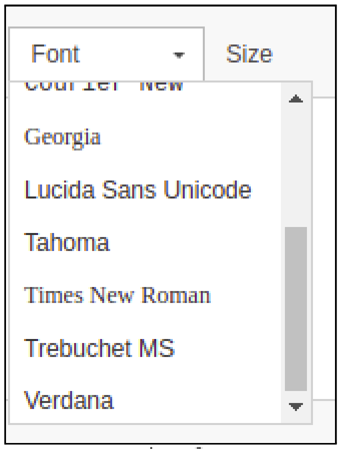


The Emoji Box that pops up on clicking emoji icon

b. Allows using different font families, font color, and features like Bold, Italics, Strikethrough, and many others:

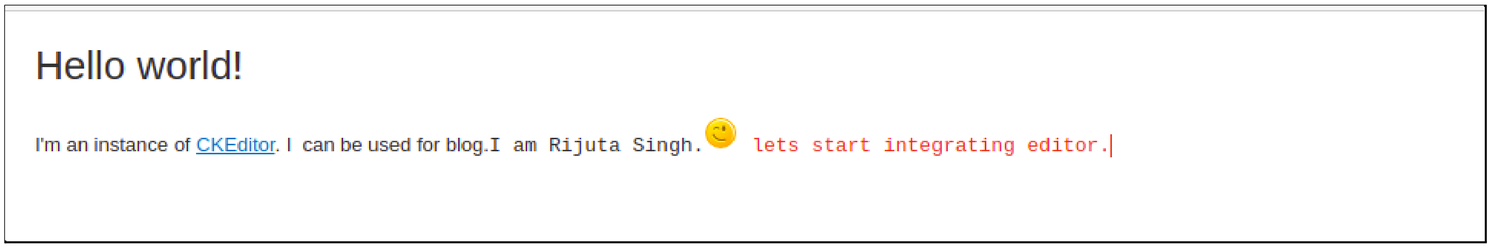


Dropdown for selecting heading type



Dropdown for selecting font-size

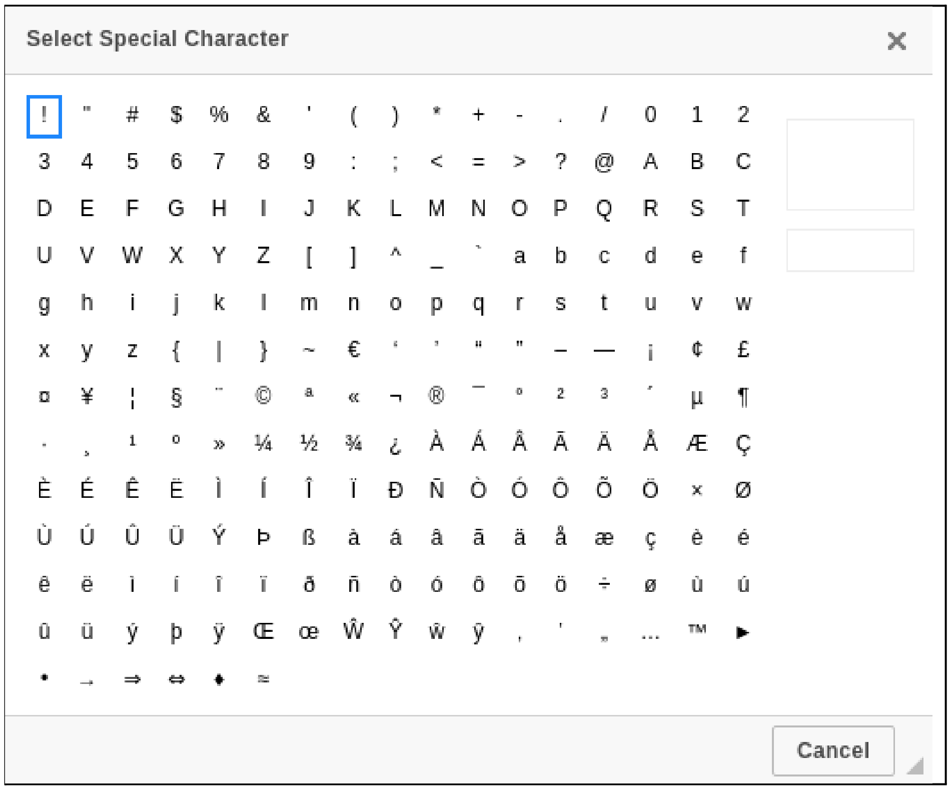
Rendered view



Hello world!

I'm an instance of CKEditor. I can be used for blog.I am Rijuta Singh. lets start integrating editor.

c. Enables adding special characters.



Select Special Character

! " # $ % & ' ( ) * + - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ € ‘ ’ “ ” – — ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ Œ œ Ŵ Ŷ ŵ ŷ ‚ ‛ „ … ™ ► • → ⇒ ⇔ ♦ ≈

And many more like adding images, Iframe, etc which are necessary to make a blog attractive enough and ensure that it is not merely a long piece of text!

2. One of the major reasons to use this editor is that Oppia’s Explorations are created using the RTEditor which is just a wrapper around CKEditor and hence the CKEditor is already well implemented inside Oppia’s code base thus, removing the need to add any other third-party libraries or files.

a. RTE is used to create explorations and thus it disables some of the plugins offered by CKEditor instance to ensure user-friendliness as they aren't required for making explorations and skills.
b. Therefore, inside the blog editor, some extra features may be required. So instead of using RTE as a base, **I will prefer using the CKEditor directly.Above this CKEditor, a ‘blog-editor’ wrapper will be applied which will be the same as current RTE but with all listed plugins enabled.**

No new additional plugins are required to be downloaded. They are just to be enabled to offer more features (i.e are to be included in the build and config) to make the blog editor more interactive and powerful.

3. **Final List of Features** to be included in the toolbar for the blog editor is as follows:

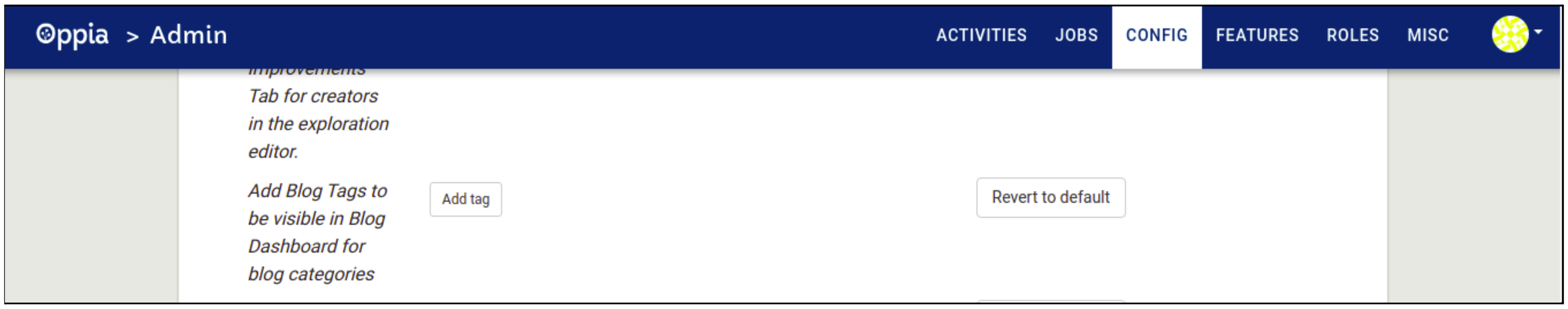
Bold	Sub-Script	Font-Style and Size	Emojis	Underline	Table
Italics	Super-Script	Headings	Special Characters	Image	Copy and Paste
Strike-Through	Font-Color	Font-alignments	Types of lists	I-Frame	Spell-check

ADDING TAGS FOR BLOGS:

Adding tags that will be visible in the blog dashboard to assign categories to blogs can only be done by Admins. (Until we support user-defined tags as explained above).

To add tags:

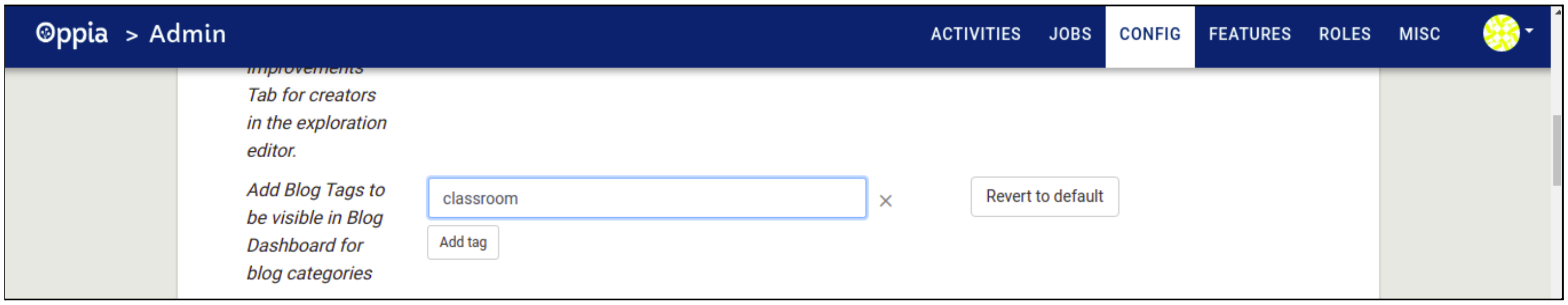
- Users should be logged in as admin. This will make the admin page accessible.
- Navigate to "admin page" via the profile drop-down.
- Go-to "config-tab".
- Click on the "add tag" button in front of the "*Add Blog Tags to be visible in Blog Dashboard for blog categories" config property.*
- *This will open the schema-based editor to input character strings.*



improvements Tab for creators in the exploration editor.

Add Blog Tags to be visible in Blog Dashboard for blog categories

The Admin page -Config tab with blog tag property



improvements Tab for creators in the exploration editor.

Add Blog Tags to be visible in Blog Dashboard for blog categories

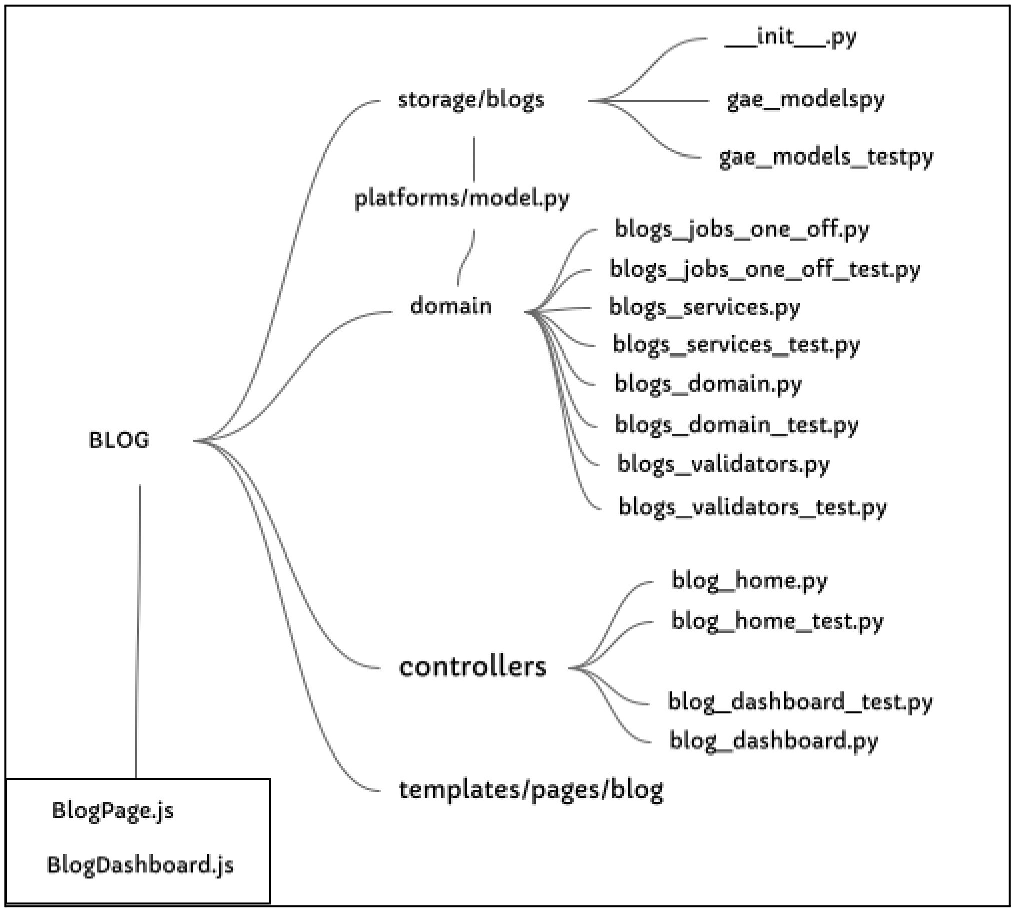
classroom

The schema based editor in active mode while adding tag

- Clicking on "Save Button" will make the added tags visible in the dashboard.

Technical Design

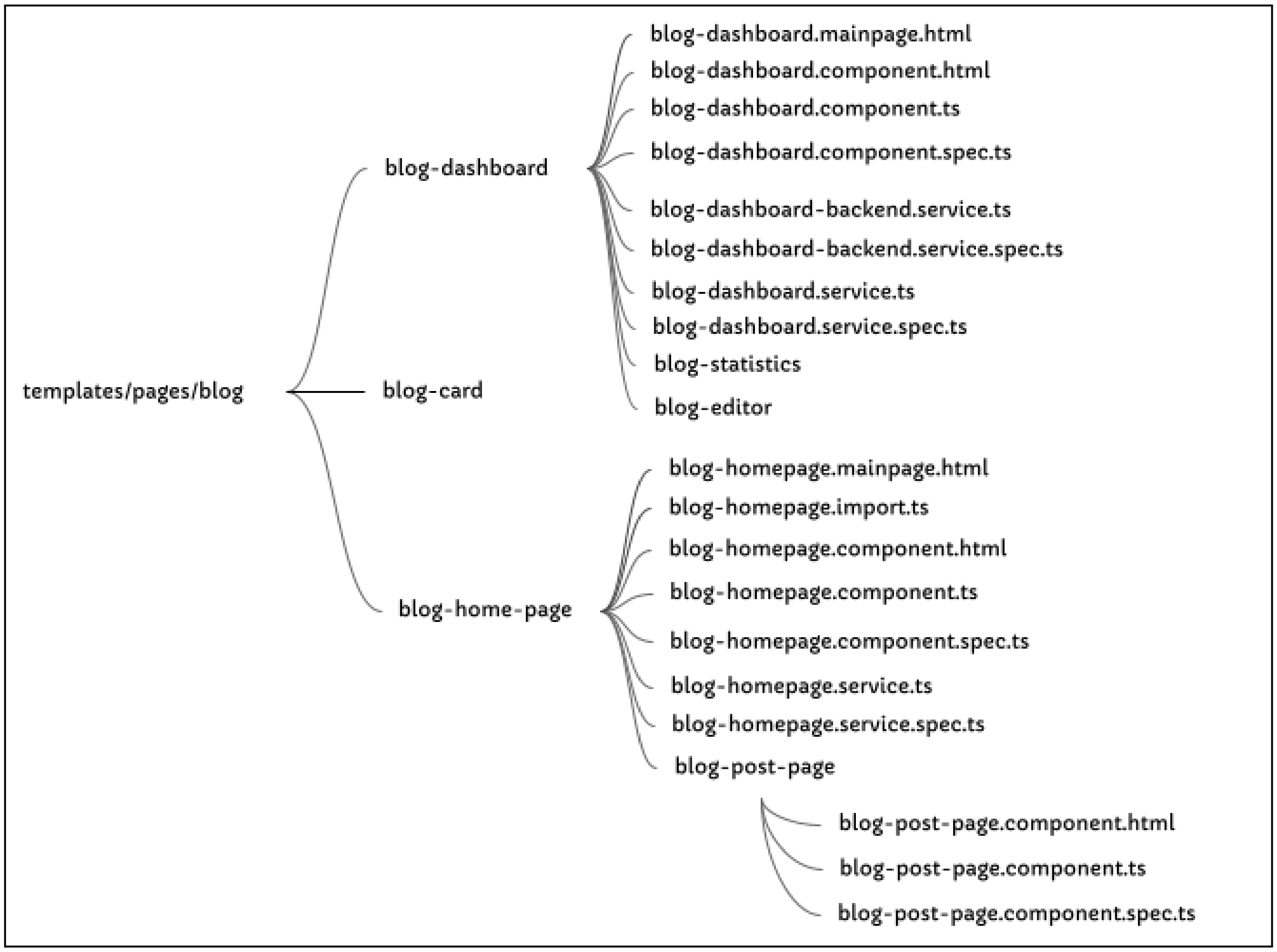
Architectural Overview



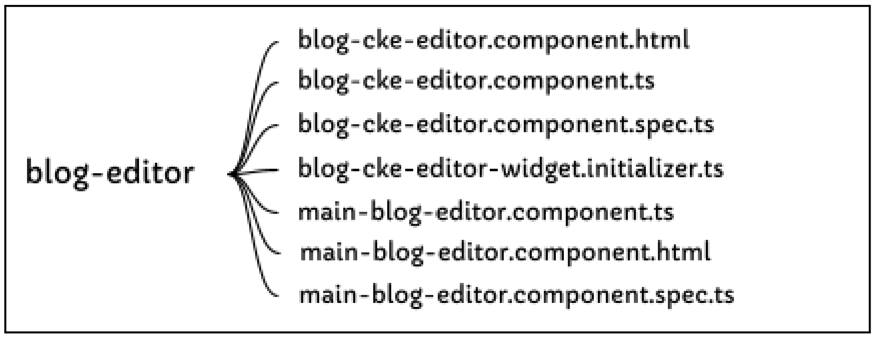
To create a blogging feature, the general structure for new files will be as follows:

- New storage model in (oppia/core/storage/**blog**)
- Related domain objects and services accompanied by their tests and storage model validator in (oppia/core/domain/)
- Controllers (for handling requests) from both the dashboard and editor accompanied by their tests in (oppia/core/controllers/)

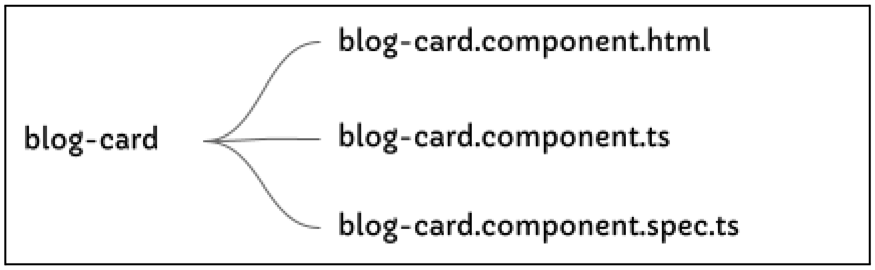
- Component frontend files in (template/pages/**blog)** with their services and test.


templates/pages/blog
blog-dashboard
blog-card
blog-home-page
blog-dashboard.mainpage.html
blog-dashboard.component.html
blog-dashboard.component.ts
blog-dashboard.component.spec.ts
blog-dashboard-backend.service.ts
blog-dashboard-backend.service.spec.ts
blog-dashboard.service.ts
blog-dashboard.service.spec.ts
blog-statistics
blog-editor
blog-homepage.mainpage.html
blog-homepage.import.ts
blog-homepage.component.html
blog-homepage.component.ts
blog-homepage.component.spec.ts
blog-homepage.service.ts
blog-homepage.service.spec.ts
blog-post-page
blog-post-page.component.html
blog-post-page.component.ts
blog-post-page.component.spec.ts

The Frontend Component files


blog-editor
blog-cke-editor.component.html
blog-cke-editor.component.ts
blog-cke-editor.component.spec.ts
blog-cke-editor-widget.initializer.ts
main-blog-editor.component.ts
main-blog-editor.component.html
main-blog-editor.component.spec.ts

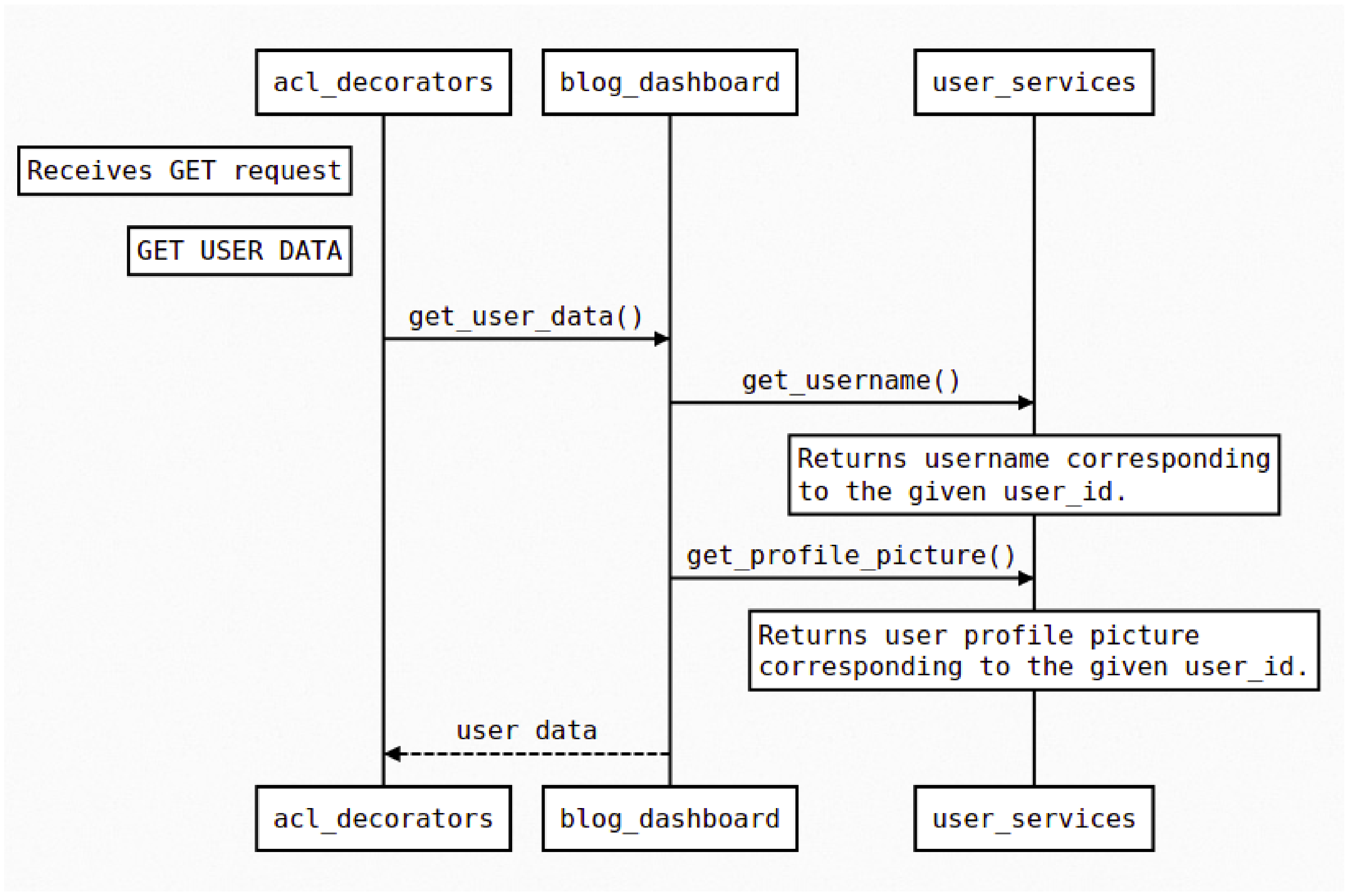
The Blog Editor files



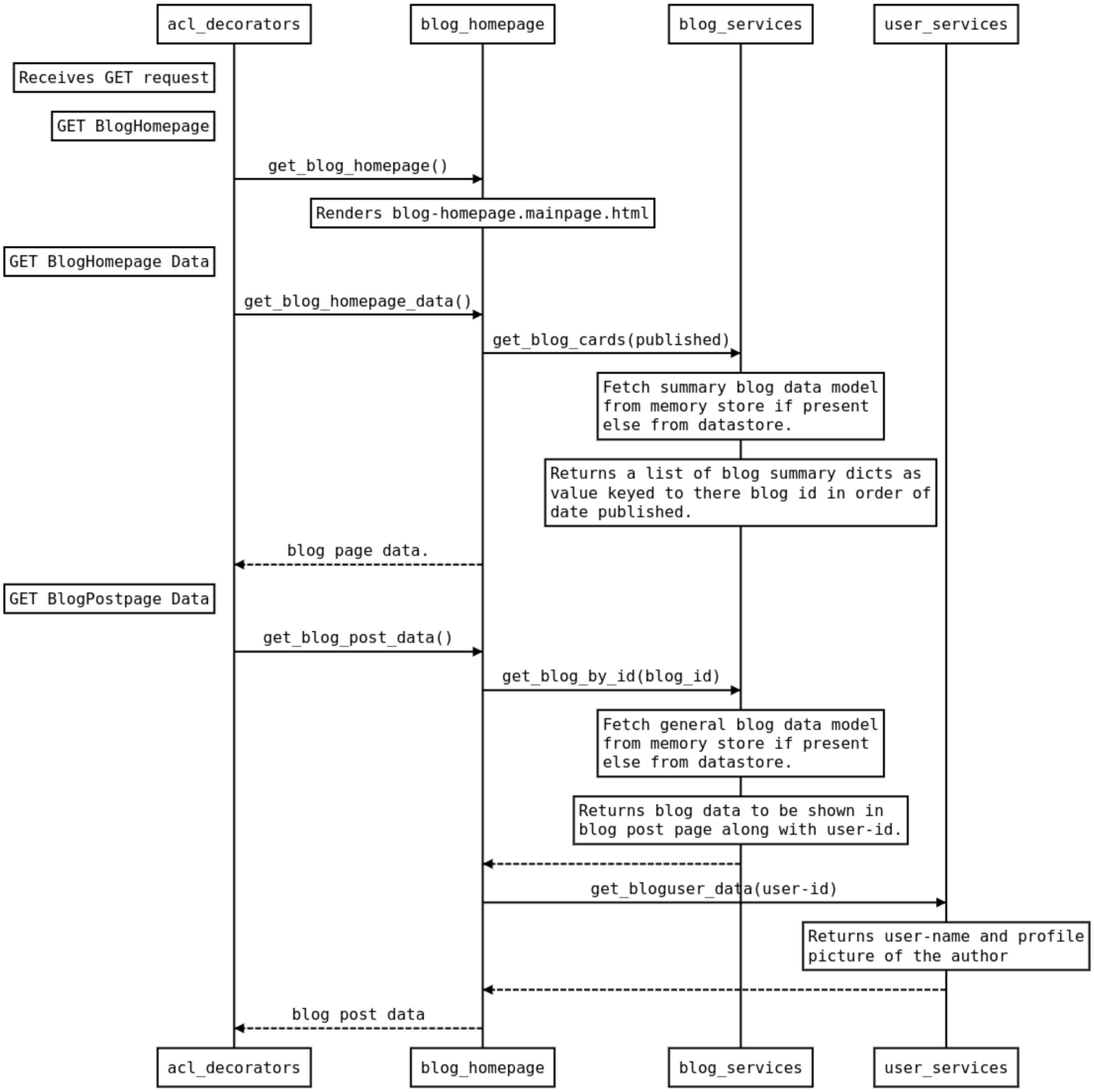
The Blog Card files

Sequence diagram :

Sequence diagram to convey system structure between controllers and domain services.



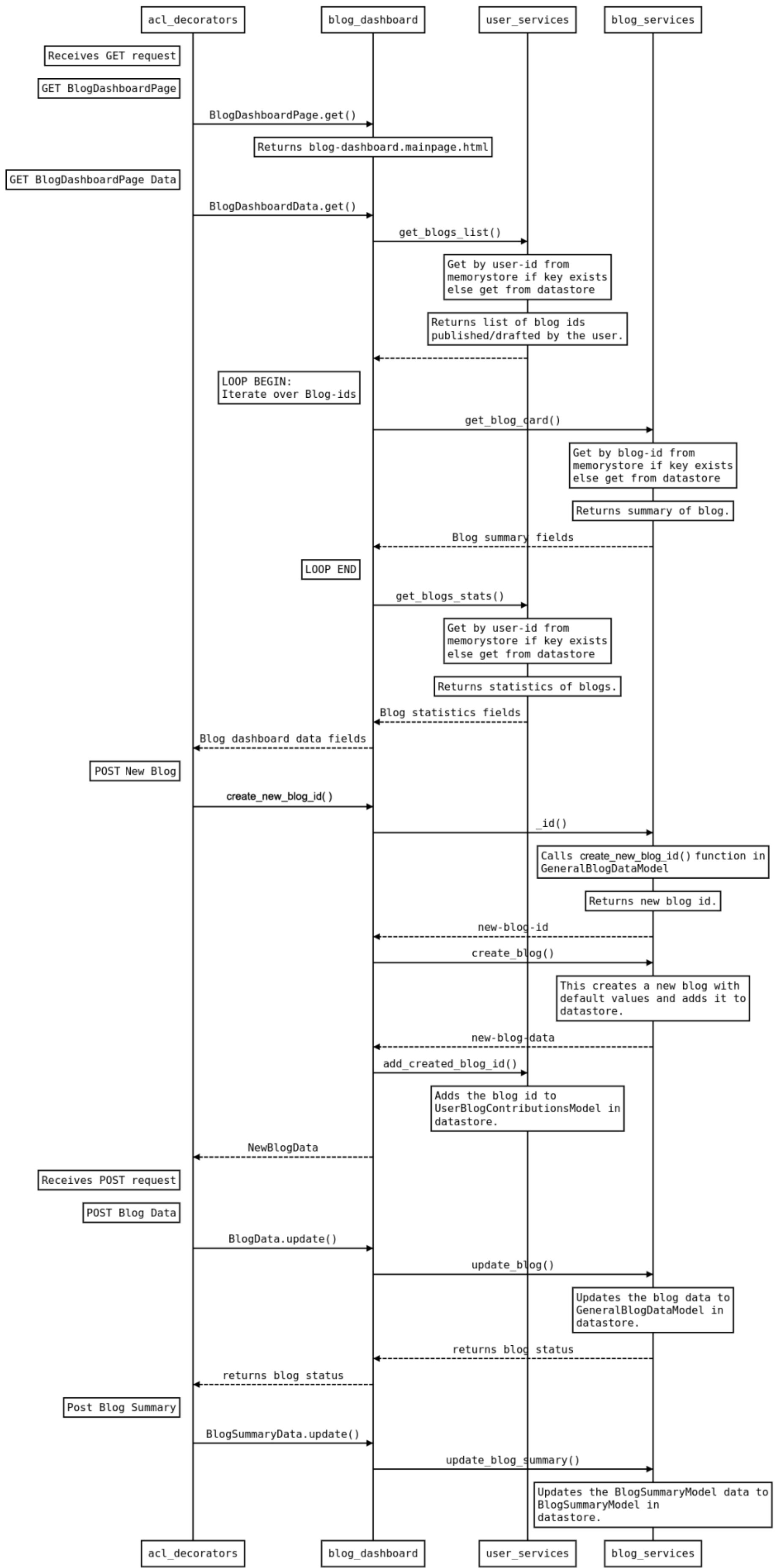
Sequence Diagram for getting user data in blog dashboard


acl_decorators
blog_homepage
blog_services
user_services
Receives GET request
GET BlogHomepage
get_blog_homepage()
Renders blog-homepage.mainpage.html
GET BlogHomepage Data
get_blog_homepage_data()
get_blog_cards(published)
Fetch summary blog data model from memory store if present else from datastore.
Returns a list of blog summary dicts as value keyed to there blog id in order of date published.
blog page data.
GET BlogPostpage Data
get_blog_post_data()
get_blog_by_id(blog_id)
Fetch general blog data model from memory store if present else from datastore.
Returns blog data to be shown in blog post page along with user-id.
get_bloguser_data(user-id)
Returns user-name and profile picture of the author
blog post data
acl_decorators
blog_homepage
blog_services
user_services

Sequence Diagram for blog homepage

To have a clear view of the Sequence diagram for Blog Dashboard Implementation: visit the Sequence diagram.

Snap of the diagram.


acl_decorators
blog_dashboard
user_services
blog_services
Receives GET request
GET BlogDashboardPage
BlogDashboardPage.get()
Returns blog-dashboard.mainpage.html
GET BlogDashboardPage Data
BlogDashboardData.get()
get_blogs_list()
Get by user-id from memorystore if key exists else get from datastore
Returns list of blog ids published/drafted by the user.
LOOP BEGIN: Iterate over Blog-ids
get_blog_card()
Get by blog-id from memorystore if key exists else get from datastore
Returns summary of blog.
Blog summary fields
LOOP END
get_blogs_stats()
Get by user-id from memorystore if key exists else get from datastore
Returns statistics of blogs.
Blog statistics fields
Blog dashboard data fields
POST New Blog
create_new_blog_id()
_id()
Calls create_new_blog_id() function in GeneralBlogDataModel
Returns new blog id.
new-blog-id
create_blog()
This creates a new blog with default values and adds it to datastore.
new-blog-data
add_created_blog_id()
Adds the blog id to UserBlogContributionsModel in datastore.
NewBlogData
Receives POST request
POST Blog Data
BlogData.update()
update_blog()
Updates the blog data to GeneralBlogDataModel in datastore.
returns blog status
returns blog status
Post Blog Summary
BlogSummaryData.update()
update_blog_summary()
Updates the BlogSummaryModel data to BlogSummaryModel in datastore.
acl_decorators
blog_dashboard
user_services
blog_services

- Apart from the creation of new files, the following existing files will be modified (for details see Implementation approach) :

  - **feconf.py** to add new constants (for example: user role ID, blog status).
  - **role_services.py** to add the new role.
  - **models.py** in core/platform to include the newly created blog storage model.
  - **Files related to user storage :**
    Inside storage/users :
      - gae_models.py
      - gae_models_test.py

    Inside domain :
      - user_services.py
      - user_services_test.py
      - user_domain.py
      - user_domain_test.py
      - user_validators.py
      - user_jobs_one_off.py
  - **main.py** will be added with new routes.
  - **app_dev.yaml and app.yaml** will be modified to remove the plugins for cke-editor from the list that removes them from build. (See app_dev.yaml and app.yaml)
  - Webpack.common.config.ts, lighthouserc.js, and lighthouserc-accessibility.js will be modified to add the new blog pages.
  - **components/top_navigation_bar/top_navigation_bar_directive.html:** Another element to be added to the dropdown called Blog Dashboard. It will only be visible if the user has "blog_editor" role.
  - To **add new config property** (to add tags which can be then assigned to blogs):
    - admin-backend-api.service.ts
    - config_domain.py
    - config_services.py
    - admin.py and admin_test.py (controllers for admin page)
    - Other related files like config_validators
  - To create a **search handler** for the blogs-
    In core/domain -
      - search_services.py (and respective test file).

Implementation Approach-

INTRODUCING NEW CONSTANTS:

New constants will be added inside the **feconf.py** file some of which are:

- ○ NEW_BLOG_URL : '/blog_editor_handler/create_new_blog'
  1. This URL will handle POST requests which will hit whenever the "New-Post" button or "Create Your Blog" is clicked.
- ○ BLOG_EDITOR_DATA_URL_PREFIX: '/blog_editor_handler/data'
  1. This URL will handle PUT requests in case of updating the blog.
  2. It will also manage GET requests to populate the blog's data in the blog editor.
- ○ BLOG_EDITOR_URL_PREFIX: '/blog_editor'
  1. This URL will handle GET requests to fetch tags, and other data on the blog dashboard's editor.(mainly to get the predefined tags set by admin)
- ○ BLOG_DASHBOARD_DATA_URL : '/blog_dashboard/data'
  1. This URL will handle GET requests to fetch the data for blog dashboard.(published and draft blog cards and the user data)
  2. It will also respond to DELETE requests to delete a particular blog.
- ○ ROLE_ID_BLOG_EDITOR : 'BLOG_EDITOR'
- ○ BLOG_DASHBOARD_URL: '/blog_dashboard' to render the mainpage.html on GET request.

1. BLOG STORAGE MODELS:

Under core/storage, a new folder named blog with the following files will be created:

○ **__init__.py:** An empty file to initialize the blog model.
○ **gae_models.py:** Inside it models for Oppia's blogs will be added containing the blog's content, author-id, and other data storage classes.
○ **gae_models_test.py:** A file containing tests for corresponding gae_models. "Gae_model.py" for blog storage will be completely tested inside this file

BLOG STORAGE MODELS:

Inside gae_models.py:
There will be 3 classes namely :

1. ClassBlogDataModel
2. ClassBlogSummaryDataModel
3. ClassBlogRightsModel

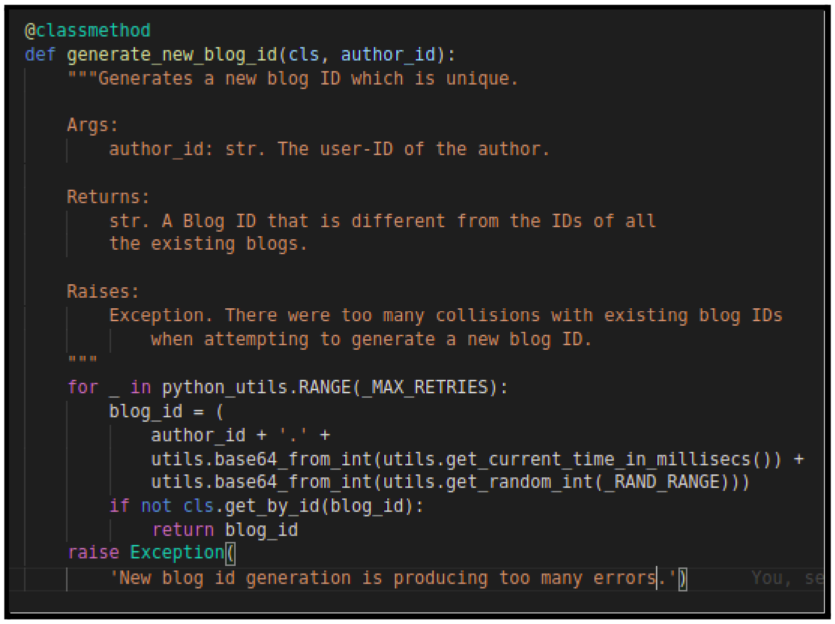
In ClassBlogDataModel: The main schema of the blog and its associated functions. We do not require a versioned model for blogs.Thus this will be basic base models.It will have the following fields:

Each instance of this model will be keyed by blog-id which will be of the form [author_id].[generated string]. Blog-id will be unique for every blog.

Field Name	Property type	To store..	Required/Indexed
author_id	String Property	The user_id of the author	Required and Indexed
title	String Property	Title of Blog	Required and Indexed
content	JSON Property	The content entered in blog-RTE	Required but **not** indexed
url_fragment	String Property	The unique URL fragment for the blog.	Required but **not** indexed
time_publish	DateTimeProperty	Storing time of publish	Required and Indexed
time_updated	DateTimeProperty	Storing the time when the blog was last updated	Required but **not** indexed

For this data model following policies will be applied:

For export :
'get_export_policy' class method will be defined. The model contains the user-id of the author and it will be exported.
For deletion :
'get_deletion_policy' class method will be defined. The model contains data to locally pseudonymize corresponding to a user - author_id.
For model association to user :
'get_model_association_to_user()' class method will be defined inside which, under which model will be exported as one instance per user.
It will have the following class method to generate a unique string to ensure that no 2 blogs have same blog_id :



```
@classmethod
def generate_new_blog_id(cls, author_id):
    """Generates a new blog ID which is unique.

    Args:
        author_id: str. The user-ID of the author.

    Returns:
        str. A Blog ID that is different from the IDs of all
        the existing blogs.

    Raises:
        Exception. There were too many collisions with existing blog IDs
            when attempting to generate a new blog ID.
    """
    for _ in python_utils.RANGE(_MAX_RETRIES):
        blog_id = (
            author_id + '.' +
            utils.base64_from_int(utils.get_current_time_in_millisecs()) +
            utils.base64_from_int(utils.get_random_int(_RAND_RANGE)))
        if not cls.get_by_id(blog_id):
            return blog_id
    raise Exception(
        'New blog id generation is producing too many errors.')       You, se
```

In ClassBlogSummaryDataModel: It can be used whenever the content of the blog is not required. It will mainly populate data in the Blog Cards and will be used during searches.

Field Name	Property type	To Store..	Required/Indexed
author_id	String Property	The user_id of the author	Required and Indexed
title	String Property	Title Of Blog	Required and Indexed
summary	Text Property	The summary of the blog.	Required but **not** indexed
thumbnail_filename	String Property	Storing the unique URL fragment for the blog.	Required but **not** indexed
time_publish	DateTimeProperty	Storing time of publish	Required and Indexed
time_updated	DateTimeProperty	Storing the time when the blog was last updated	Required but **not** indexed
tags	String Property	Storing the tags	Required and Indexed

For this data model following policies will be applied:
For export :
'get_export_policy' class method will be defined. The model contains the user-id of the author and it will be exported.
For deletion :
'get_deletion_policy' class method will be defined. The model contains data to locally pseudonymize corresponding to a user - author_id.
For model association to user :
'get_model_association_to_user()' class method will be defined inside which, under which model will be exported as one instance per user.

In ClassBlogsRightsModel: To store rights related to a model. And also to track blog status.

Field Name	Property type	To Store..	Required/Indexed
editor_ids	String Property	The user_ids which have edit access to the blog.	Required and Indexed
blog_status	Boolean Property	published/draft status. Will be true if published.By default, it will be false.	Required and Indexed

For this data model following policies will be applied:
For export :
'get_export_policy' class method will be defined. The model contains the user-ids of the editors and it will be exported.
For deletion :

‘get_deletion_policy’ class method will be defined. The model contains data to locally pseudonymize corresponding to users - editor_ids.

For model association to user :

‘get_model_association_to_user()’ class method will be defined inside which, under which model will be exported as one instance per user.

2. ENLISTING STORAGE MODELS TO MAKE THEM VALID:

In platform/models.py:

○ The blog models have to be added to the ‘NAMES’ list so that they can be enlisted as valid model names.

○ In the function import_models() in class _Gae, the above models have to be added to the if-elif statements to import the created gae_models.py file for each in core/storage.

3. ADDING MODELS IN USER STORAGE MODEL :

Class UserBlogContributionsModel: It will be added to store IDs of blogs created by the user and will be used when a user visits the blog-dashboard to populate the dashboard with the blog created by the user.Instances will be keyed by **user-id**.

Field Name	Property type	To Store..	Required/Indexed
blog_ids	String Property	The blog_ids of blogs created by user	Required and Indexed

To handle users ability to export and delete model:

For export :

‘get_export_policy‘ class method will be defined. It contains data to be exported corresponding to the user- blog-ids. However it **does not** contain any specific user detail (such as user id)

For deletion :

‘get_deletion_policy’ class method will be defined. The models contain data to be deleted corresponding to the user : blog-id fields.

For model association to user :

‘get_model_association_to_user()’ class method will be defined inside which, under which model will be exported as one instance per user.

4. CONTROLLERS:

1. blog_dashboard.py:

- **Class BlogDashboard:** To render the frontend html page when the URL for the blog dashboard is called. This will fetch the global information about blogs created by the user (using the id) and render their respective html page for the dashboard using the function self.render_template() .

- **Class New Blog:** It will be a type of post request handler that will create a new blog with default values for all fields (will raise a call to blog_domain.create_new_blog()) through which a unique blog_id will be generated(blog_service.create_new_blog_id). The blog_id of the newly created blog is returned.

- **Class BlogDashboardDataHandler:** It will be completely responsible for loading data on the dashboard. This will reduce the number of "GET" requests to be placed and also will make it easier as only one handler will be loading the page data initially, fetch the user_id, profile picture and using get_blogs_list(user_id) to get the list of blog_ids and then if the length of the list is greater than 0, it will be fetching both blogsRights and blogSummaries

- **Class BlogPostHandler:** It is to be created with the function get_blog_data() which will return a dictionary of values based on the blog_id and this will be called by a get() function which will then return the values to the frontend components. It will also have put() and delete() functions for updating and deleting the blog.

- **Class BlogTagsHandler :** It will contain the get() function to retrieve the list of blog tags to populate the dashboard with the tags which can be used to categorize the blog. This controller will make calls to config_domain.py .

2. blog_dashboard_test.py:

It will be the test file to completely test blog_dashboard.py. Some test functions are:

1. test_access_blog_dashboard_page(): This will make sure that users with the blog-editor role can only access the dashboard.

2. test_editable_blog_handler_put_can_access(): This will make sure that only blog-editors can do a put request to the 'blog post handler'.

3. test_publish_blog(): This would test the working of the blog publication process. Thus a complete testing of the separate blog dashboard components will be done here.

4. test_fetch_blog_handler(): This will create a few blogs (some published and some drafts) then assign them to a user and test whether the 'FetchBlogsHandler' returns the correct set of blogs with their correct status.

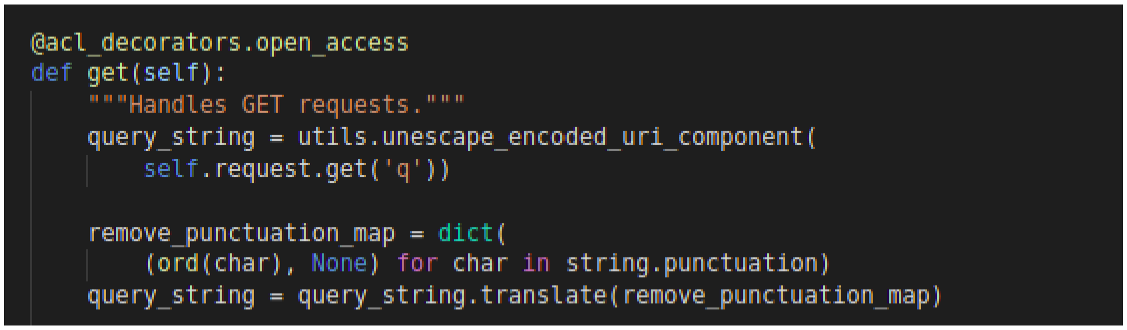
3. blog_home_page.py:

- **Class BlogHomepage:** To render the frontend html page when the URL for the blog homepage is called. It will fetch the information about all the blogs published till date and render the Html page for the homepage using the function self.render_template() .

- **Class BlogHomepageHandler:** This will have the get_all_cards() function which will return the published blogs for the blog homepage page.

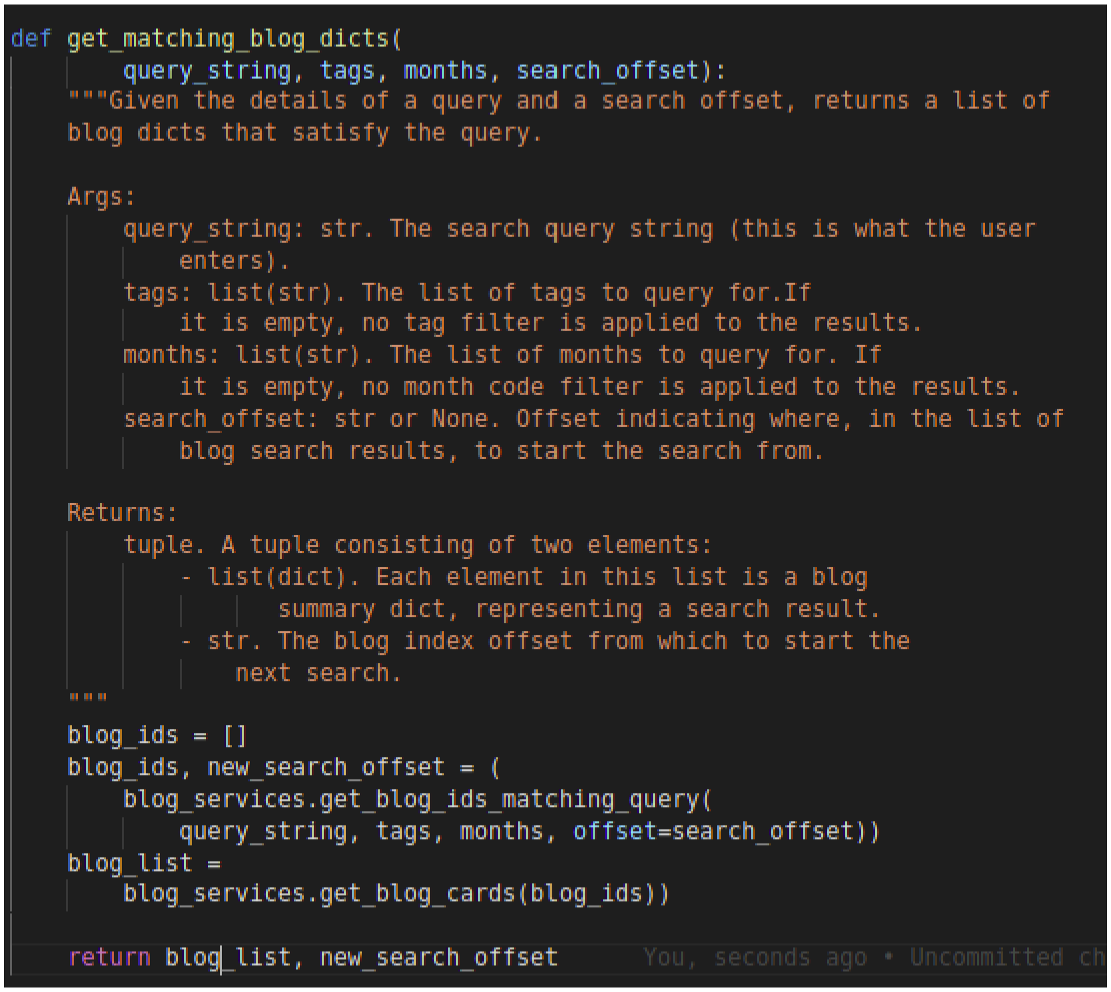
- **ClassBlogpostHandler:** get_blog_data() function which will return a dictionary of values based on the blog_id and this will be called by a get() function which will then return the values to the frontend for Blog-Post-Page.

- **ClassSearchHandler:** This will have a get() function which will return the list of blogs dicts satisfying the user's query. It will call get_matching_blog_dicts() as defined above.
  It will format the query string into a form which can be used for searching,that is,all punctuations will be removed and will be replaced by spaces. Any other required formatting will also take place here .It will supply all the parameters provided in proper order to get_matching_blog_dicts() function .Snap of a part of function :



```
@acl_decorators.open_access
def get(self):
    """Handles GET requests."""
    query_string = utils.unescape_encoded_uri_component(
        self.request.get('q'))

    remove_punctuation_map = dict(
        (ord(char), None) for char in string.punctuation)
    query_string = query_string.translate(remove_punctuation_map)
```

- **Function get_matching_blog_dicts():** Given the details of a query i.e title, metatags or categories, authors name, and a search offset, it will return a list of blog dicts that satisfy the query.



```python
def get_matching_blog_dicts(
        query_string, tags, months, search_offset):
    """Given the details of a query and a search offset, returns a list of
    blog dicts that satisfy the query.

    Args:
        query_string: str. The search query string (this is what the user
            enters).
        tags: list(str). The list of tags to query for.If
            it is empty, no tag filter is applied to the results.
        months: list(str). The list of months to query for. If
            it is empty, no month code filter is applied to the results.
        search_offset: str or None. Offset indicating where, in the list of
            blog search results, to start the search from.

    Returns:
        tuple. A tuple consisting of two elements:
            - list(dict). Each element in this list is a blog
                  summary dict, representing a search result.
            - str. The blog index offset from which to start the
                next search.
    """
    blog_ids = []
    blog_ids, new_search_offset = (
        blog_services.get_blog_ids_matching_query(
            query_string, tags, months, offset=search_offset))
    blog_list =
        blog_services.get_blog_cards(blog_ids))

    return blog_list, new_search_offset      You, seconds ago • Uncommitted ch
```

4. blog_home_page_test.py:

It will be a test file to completely test blog_home_page.py. Some functions in it are :

1. test_blog_homepage_handler(): First, a few sample blogs can be created with some random publishing dates, and then, the handler function can be tested to make sure it returns the correct blog with the correct date.

2. test_blog_post_page_handler(): The blogs created above will be checked for returning correct values in Blog-Post-Page .

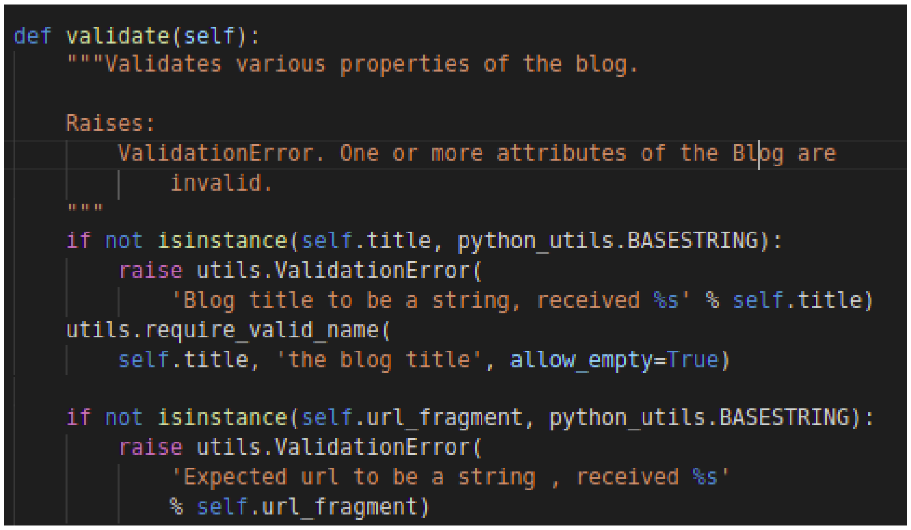
3. test_search_handler(): Multiple query strings will be created and checked for returning correct blogs from the above-created blogs.

5. DOMAIN:

Inside the 'domain' directory, new files will be created namely:

1. **blog_domain.py:**

- **class Blog:** It would be present to handle all functions related directly to the Blog model. The constructor will create a Blog object having all related fields and return it.
  Some functions (class methods) inside it will be :
  1. __init__() : Initializes a Blog domain object.
  2. blog_id() : Returns the Blog ID corresponding to the blog.
  3. require_valid_blog_id(): It validates the blog-id.
  4. to_dict() : It will return a dict representation of the blog object.
  5. from_dict(): It will return a Blog domain object from a dict.
  6. **create_new_blog():** function would just call the constructor with all default values of the Blog.
  7. validate(): To validate the domain object.



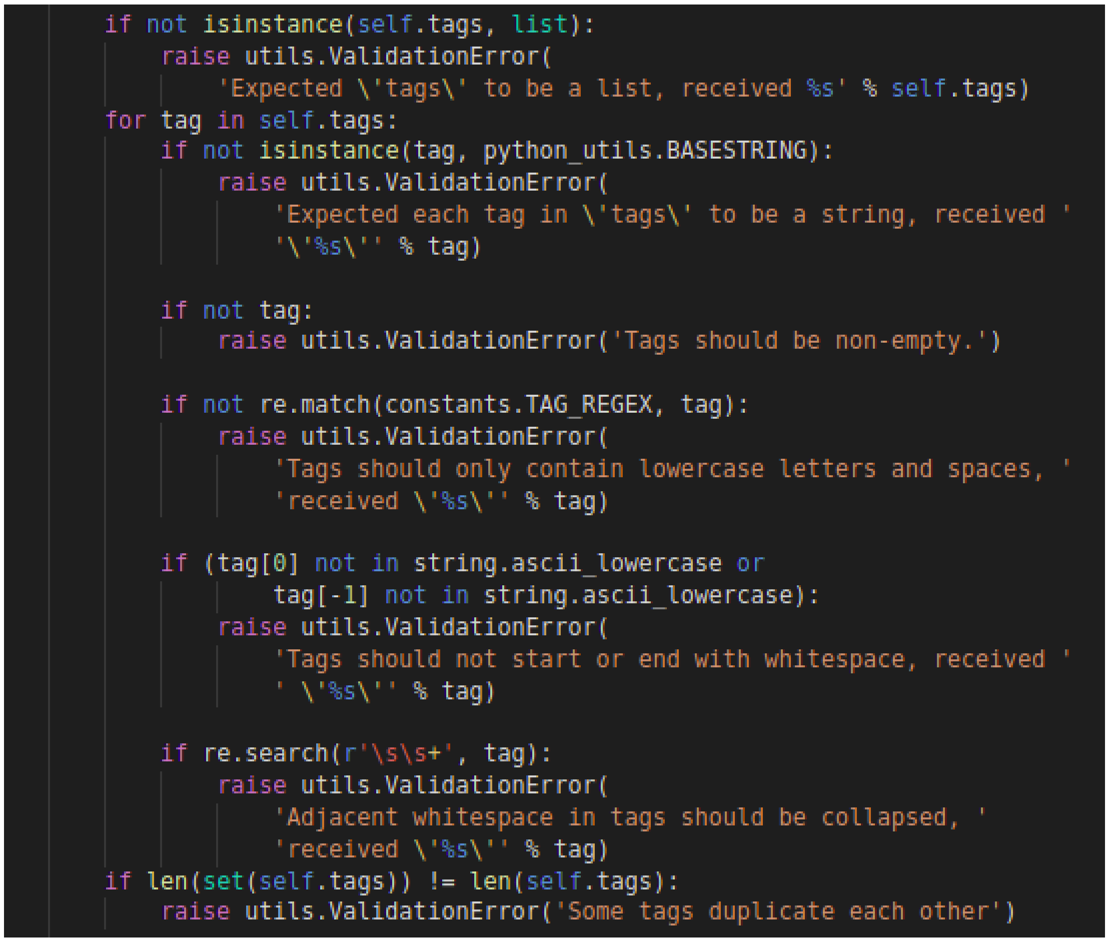
```
def validate(self):
    """Validates various properties of the blog.

    Raises:
        ValidationError. One or more attributes of the Blog are
            invalid.
    """
    if not isinstance(self.title, python_utils.BASESTRING):
        raise utils.ValidationError(
            'Blog title to be a string, received %s' % self.title)
    utils.require_valid_name(
        self.title, 'the blog title', allow_empty=True)

    if not isinstance(self.url_fragment, python_utils.BASESTRING):
        raise utils.ValidationError(
            'Expected url to be a string , received %s'
            % self.url_fragment)
```

Similar validations will be added for other required fields

- **class BlogSummary:** It would be present to handle all functions directly related to the Blog summary model.

  Some functions (class methods) inside it will be :
  1. __init__() : Initializes a Blog Summary domain object.
  2. to_dict(): It will return a 'dict' representation of the blog summary object.
  3. from_dict(): It will return a Blog summary domain object from a 'dict'.
  4. create_new_blog_summary(): function would just call the constructor to initialize.

5. validate(): To validate the domain object. To validate the tags list :



```
if not isinstance(self.tags, list):
    raise utils.ValidationError(
        'Expected \'tags\' to be a list, received %s' % self.tags)
for tag in self.tags:
    if not isinstance(tag, python_utils.BASESTRING):
        raise utils.ValidationError(
            'Expected each tag in \'tags\' to be a string, received '
            '\'%s\'' % tag)

    if not tag:
        raise utils.ValidationError('Tags should be non-empty.')

    if not re.match(constants.TAG_REGEX, tag):
        raise utils.ValidationError(
            'Tags should only contain lowercase letters and spaces, '
            'received \'%s\'' % tag)

    if (tag[0] not in string.ascii_lowercase or
            tag[-1] not in string.ascii_lowercase):
        raise utils.ValidationError(
            'Tags should not start or end with whitespace, received '
            ' \'%s\'' % tag)

    if re.search(r'\s\s+', tag):
        raise utils.ValidationError(
            'Adjacent whitespace in tags should be collapsed, '
            'received \'%s\'' % tag)
if len(set(self.tags)) != len(self.tags):
    raise utils.ValidationError('Some tags duplicate each other')
```

2. blog_domain_test.py:

It would be the test file with test functions to ensure that the model objects are created, modified, and updated in a proper state and are correctly defined. blog_domain.py will be completely tested in it.

3. blog_services.py:

Some function inside it will be:

- create_new_blog_id(): It will call the create_new_blog_id() function in GeneralBlogDataModel (part of base_model).

- create_blog (): This will create a new BlogModel object with all the relevant fields and save it to the Database.

- create_blog_summary(): This will create the Blog summary model object of the blog being created.

- get_blog_by_id (blog_id): This would return the blog content and other fields.

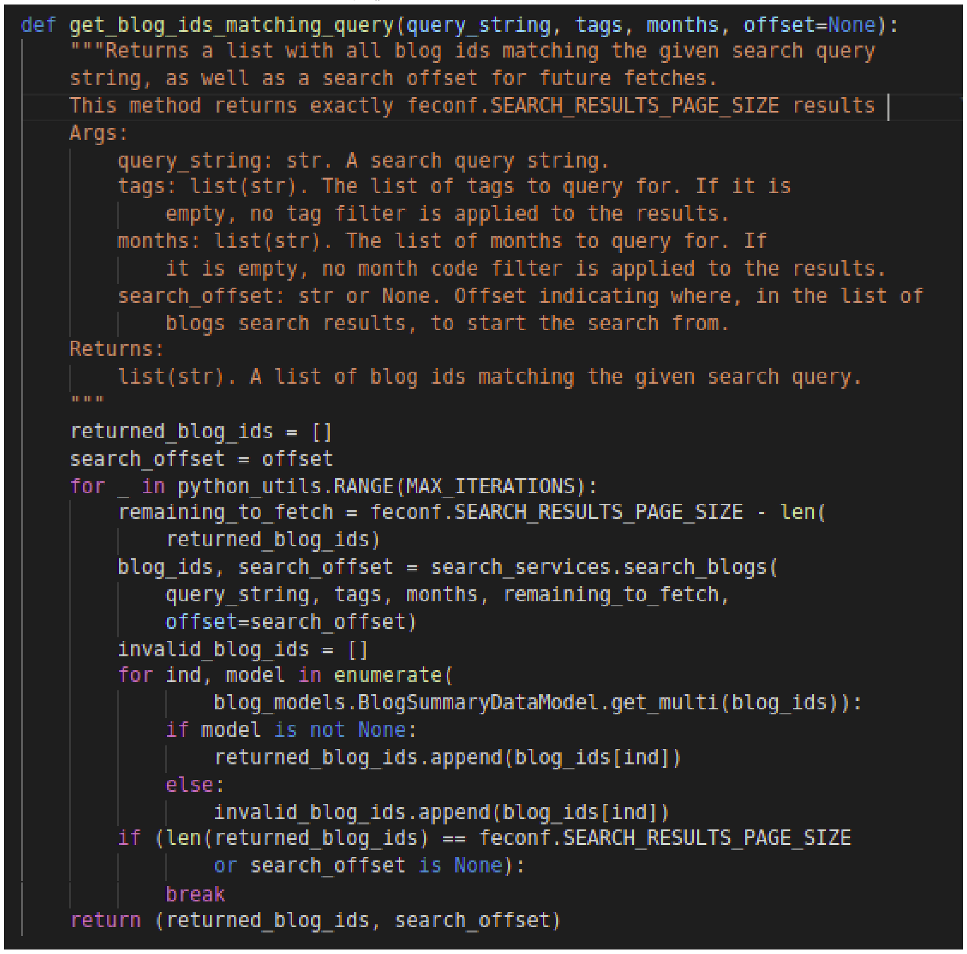
- get_blog_id_by_month (month): This would return the list of blogs by the month of them being published.

- get_blog_id_by_tags (list(tags)): This would return the list of blogs by the tags assigned to them.

- update_blog (): update the blog models with changes when edited.

- get_blog_cards (status): Returns a dict with blogSummary domain objects as values, keyed by their blog_id.This function will take status as parameter.

- get_blog_ids_matching_query(): A list of blog titles is returned corresponding to a given query.

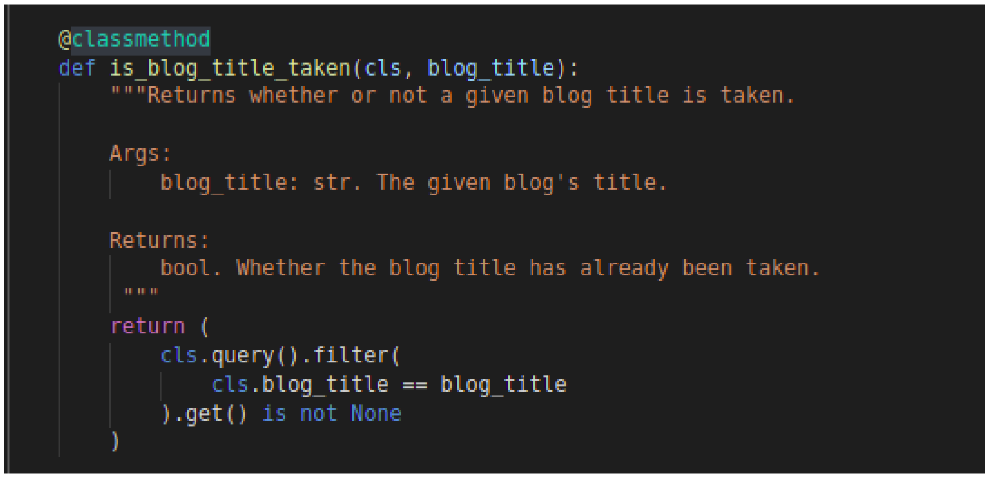


```python
def get_blog_ids_matching_query(query_string, tags, months, offset=None):
    """Returns a list with all blog ids matching the given search query
    string, as well as a search offset for future fetches.
    This method returns exactly feconf.SEARCH_RESULTS_PAGE_SIZE results
    Args:
        query_string: str. A search query string.
        tags: list(str). The list of tags to query for. If it is
            empty, no tag filter is applied to the results.
        months: list(str). The list of months to query for. If
            it is empty, no month code filter is applied to the results.
        search_offset: str or None. Offset indicating where, in the list of
            blogs search results, to start the search from.
    Returns:
        list(str). A list of blog ids matching the given search query.
    """
    returned_blog_ids = []
    search_offset = offset
    for _ in python_utils.RANGE(MAX_ITERATIONS):
        remaining_to_fetch = feconf.SEARCH_RESULTS_PAGE_SIZE - len(
            returned_blog_ids)
        blog_ids, search_offset = search_services.search_blogs(
            query_string, tags, months, remaining_to_fetch,
            offset=search_offset)
        invalid_blog_ids = []
        for ind, model in enumerate(
                blog_models.BlogSummaryDataModel.get_multi(blog_ids)):
            if model is not None:
                returned_blog_ids.append(blog_ids[ind])
            else:
                invalid_blog_ids.append(blog_ids[ind])
        if (len(returned_blog_ids) == feconf.SEARCH_RESULTS_PAGE_SIZE
                or search_offset is None):
            break
    return (returned_blog_ids, search_offset)
```

- register_blog_view(): This will increase the count of the total views on the blogs by the user. It will call user_services.py to update the UserBlogViewsModel.

- save_blog ():
    1. It will call the create_blog() function which will create and save the blog in the database irrespective of it being published or not.
    2. It will also call user_services.py to update the user storage and register the new blog authored by the author.

- add_blog_tag (): Appends the tags to the list of tags in the datastore model.
- delete_blog_tag (‘tag’): Will remove the given tag from the list of tags.
- get_blog_tags (): Returns a list of blog tags
- is_blog_title_taken () : It will take the title of the blog as parameter and will raise a call to the following class method in class BlogDataModel in storage/blog and will return boolean values depending on the result :



```python
@classmethod
def is_blog_title_taken(cls, blog_title):
    """Returns whether or not a given blog title is taken.

    Args:
        blog_title: str. The given blog's title.

    Returns:
        bool. Whether the blog title has already been taken.
    """
    return (
        cls.query().filter(
            cls.blog_title == blog_title
        ).get() is not None
    )
```

- is_blog_tag_valid(): it will take the tags attached to the blog and verify that all the tags are a part of those tags which are added by admin using the config tab.

4. blog_services_test.py:

It would be the test file with test functions to ensure that the model services are completely tested . Some of the test functions inside it will be:

1. test_no_errors_are_raised_when_creating_new_blog(): The blog creation process can be tested by using this function. It will raise a call to blog_domain.create_new_blog(). A new blog should be created with provided values with errors.

2. test_fetching_blog_by_id(): The get_blog_by_id(user_id) function can be tested to make sure it returns the correct blog list.

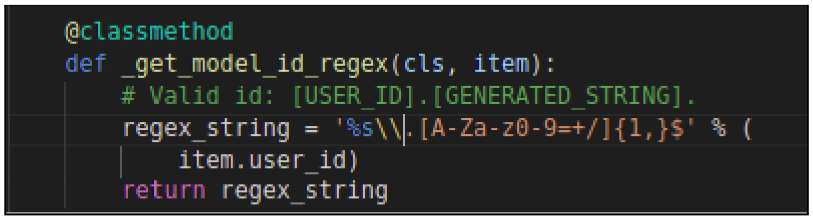
3. test_search_blog_by_query(): The get_blog_id_by_tags(), get_blog_id_by_month(), get_blog_ids_matching_query() functions can be tested inside this. It will call get_blog_ids_matching_query() which will then call get_blog_id_by_tags() and get_blog_id_by_month().

Also inside the domain directory already existing user_domain.py, user_services.py will be modified and other files related to them too.

5. blog_validators.py:

The blog storage model classes will be validated inside this file.
Some classes inside it will be :

- <u>BlogModelValidator:</u> Class will have functions to validate general blog models. It will have functions like:
    1. _get_model_id_regex:



```python
@classmethod
def _get_model_id_regex(cls, item):
    # Valid id: [USER_ID].[GENERATED_STRING].
    regex_string = '%s\\.[A-Za-z0-9=+/]{1,}$' % (
        item.user_id)
    return regex_string
```

2. _get_external_id_relationships: To validate the user id (author of the blog).
3. _get_model_domain_object_instance: To get the model object instance.
3. _validate_has_title: validates title of the blog .
4. _validate_date_publish: validates the date and time of publication of blog or last saved .
5. _validate_model_id: validates model id using the id returned in _get_model_id_regex.
6. _validate_date_updated: validates the date and time of the last update of blog .

- <u>BlogSummaryModelValidator:</u> Class will have functions to validate general blog summary models. It will have functions like:
    1. _get_model_id_regex: to validate the model id.
    2. _get_external_id_relationships: to validate the user id.
    3. _validate_datetime: validates the date and time of publication and editing of blog. It should be less than the current date and time.
    4. _validate_has_summary: validates the summary of the published blog.It should not be empty.
    5. _validate_is_published: validates the status of the blog to be published.
    6. _validate_model_id: validates model id using the id returned in _get_model_id_regex.

6. blog_validators_test.py:

This will have the test functions to validators functions. The “blog_validator.py” will be completely tested inside it.

7. blog_jobs_one_off.py:

This will have jobs to audit the blog models. Currently it will be an empty file, mostly.

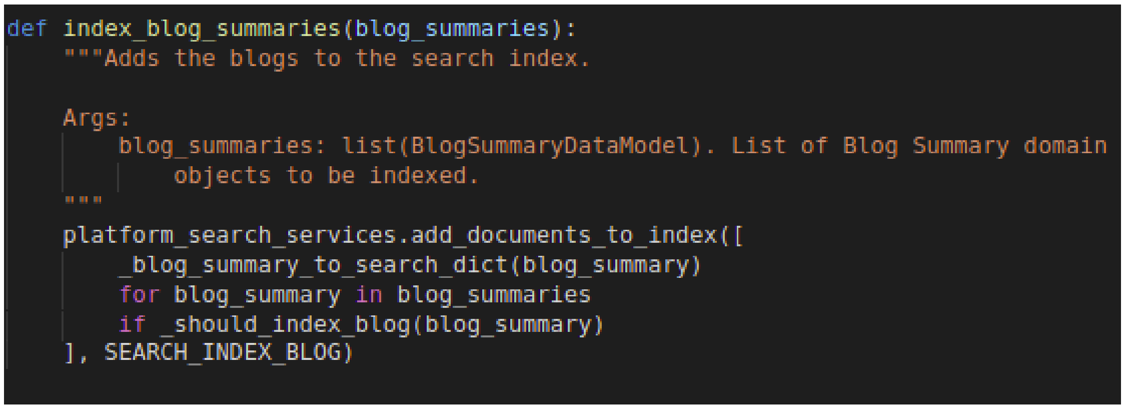
8. blog_jobs_one_off_test.py:

This will have test functions to test blog_jobs_one_off.py.

9. search_services.py:

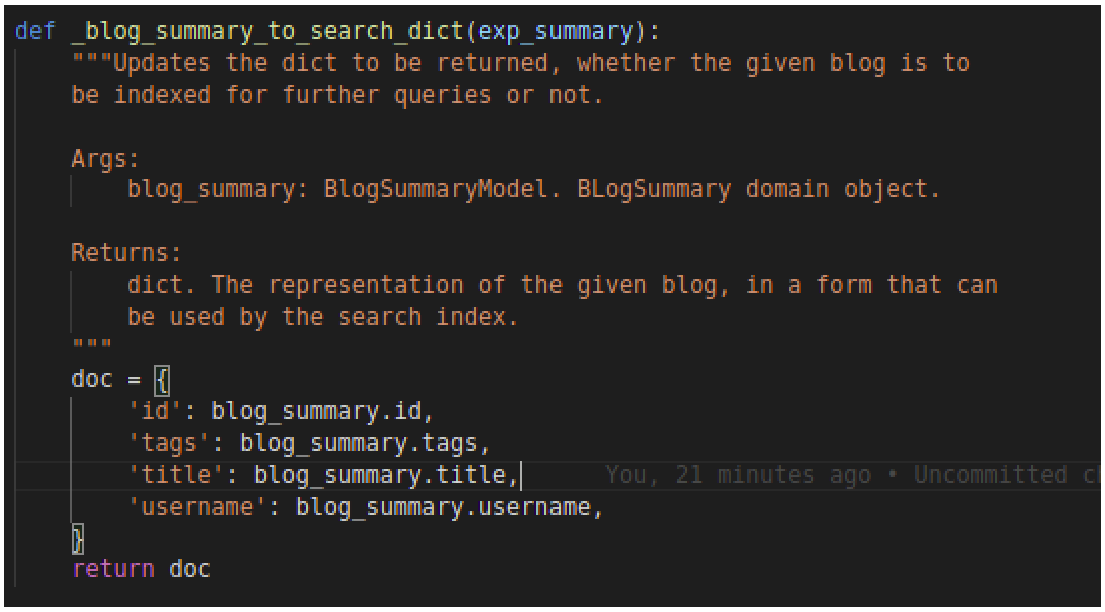
To allow user to search through various blogs, search_services will be modified accordingly and following functions will be added:

- index_blog_summaries():



```
def index_blog_summaries(blog_summaries):
    """Adds the blogs to the search index.

    Args:
        blog_summaries: list(BlogSummaryDataModel). List of Blog Summary domain
            objects to be indexed.
    """
    platform_search_services.add_documents_to_index([
        _blog_summary_to_search_dict(blog_summary)
        for blog_summary in blog_summaries
        if _should_index_blog(blog_summary)
    ], SEARCH_INDEX_BLOG)
```

- _blog_summary_to_search_dict():



```
def _blog_summary_to_search_dict(exp_summary):
    """Updates the dict to be returned, whether the given blog is to
    be indexed for further queries or not.

    Args:
        blog_summary: BlogSummaryModel. BLogSummary domain object.

    Returns:
        dict. The representation of the given blog, in a form that can
        be used by the search index.
    """
    doc = {
        'id': blog_summary.id,
        'tags': blog_summary.tags,
        'title': blog_summary.title,       You, 21 minutes ago • Uncommitted c
        'username': blog_summary.username,
    }
    return doc
```

- clear_blog_search_index() : To clear blog search index.
- search_blogs(query, tags, months, offset=None): Will perform the search taking in SEARCH_INDEX_BLOG obtained from index_blog_summaries.

10. user_domain.py:

- **Class UserBlogContribution:** It would be present to handle all functions related directly to the User Blog Contributions model. The constructor will create an object having all related fields and will return it.

Some functions (class methods) inside it will be :

1. __init__() : Initializes the domain object.
2. validate(): To validate the domain object (user_id, published_blog_ids, draft_blog_ids)
3. add_blog_id_to_list(): It will add the created blog id to the end of respective list depending upon their status(published/ draft).
4. remove_blog_id(): It will remove the blog_id from the lists in case the blog is deleted by the user.

11. User_services.py:

- add_created_blog_id(): Adds an blog_id to a user_id's UserBlogContributionsModel of the created blog.
- get_blogs_list (user_id): This would return the list of blogs along with their status created by the user.
- remove_blog_id(): Will remove the blog_id from their respective list when deleted.

12. user_services_test.py:

This file will be modified to test the above newly created functions. This can be done by creating few blogs and then editing and deleting them. All the functions introduced will ensure that the service file is completely tested for introduced functions.

Accordingly, the validators file for user's storage models and jobs_one_off.py file for users will be modified for the new storage model classes introduced and these will be tested in corresponding test files.

6. ADDING ROUTES:

main.py: New routes are to be created to create new blogs in the database as well as for viewing the blogs.

- /blog_dashboard/<user-id>: This will call the BlogDashboard class in blog_dashboard.py, to initialize the blog dashboard with the data in the backend corresponding to the passed user_id.

- /create_new_blog : This route will call a class that is a part of controllers/blog_dashboard.py. The class called NewBlog will be declared and it will call the required functions from blog_domain.py and blog_services.py in core/domain to create a new blog. This will serve the data to the blog editor when a new blog is to be created.

- /blog_dashboard/blog_editor/<blog-id> : This route will call class BlogPostHandler from blog_dashboard.py to provide data to be edited in blog-editor.

- /blog_page/<blog_id>: This will call the class BlogPostHandler in blog_home_page.py, to initialize the blog page with the data in the backend corresponding to the passed blog_id.
- /blogs: This will call the class BlogHomepage in blog_home_page.py to initialize the blog homepage.It will call all the other necessary functions from blog_domain.py and blog_services.py.
- /blogs/search : It will call the Search-Handler class inside the blog homepage when a user inputs a query to display specific blogs.
- /blog_dashboard/<user-id>/get_stats: It will call class TotalViewsHandler of blog_dashboard.py which will call related functions in core/domain files- user_domain.py and user_services.py to feed the statistics page of the blog_dashboard.

COMPONENT FILES (templates/pages/blog):

FOR BLOG DASHBOARD:

1. Inside blog-dashboards.component.html:

1. It will have a button - "new post" which on clicking will call createNewBlog() function in the blog-dashboards.component.ts . It will be visible only when the user has created any blogs before.
2. On the bar below user information, there will be -
    1) Drafts - Clicking on it will call showDrafts() in the component.ts file and will enable blog cards that are yet to be published to be visible.
    2) Published - Clicking on it will call showPublished() in the component.ts file and will enable blog cards that are published to be visible.
    3) ** Statistics - The front end of the statistics tab that is shown in mocks will be implemented afterwards and its implementation is detailed in "Future Work".

Each of the two tabs' HTML will be written inside separate section tags which will have **" *ngIf "** to make only the desired section visible. Inside each section **" *ngFor "** will be used to display the blog cards which will iterate over the list of blog-summaries defined in BlogDashboard.component.ts.

In case the user has not worked on any blogs yet,(no drafts and published blogs are present for the user in the datastore) then "activeTab" value from component.ts will be "none" which will hide all the three tabs and then " Create Your Blog" button will be visible in the main area of the dashboard with the

The HTML of blog cards will be defined in a separate component so that the same code can be reused again and again.

2. **Inside blog-dashboards.component.ts:**

It will contain typescript functions which will serve as the backbone of the html.
Some functions inside it are-

1. ngOnInIt(): It will set the default values of variables and will call the necessary functions required to initialize and populate the values on the page. Some variable inside it will be:-

   - activeTab : It will have the default value as "published" if the length of the list of publishedBlogs is **not** zero. If it is zero, then the value will be "drafts". In case both are zero, the active tab will be "none".
   - publishedBlogs : It will be a list containing blog summary dictionaries as its item.The values will be provided by the *blog-dashboard.service.ts* .
   - draftBlogs : It will be a list containing blog summary dictionaries as its item.The values will be provided by the *blog-dashboard.service.ts* .
   - *username : It will be the user-name that will be displayed on the dashboard.*

2. showDrafts(): It will be called on clicking the draft button on the page. This will change the value of "activeTab" flag to "drafts".This will inturn resolve the statement defined in the html for *ngIf in drafts section to return true, making the drafts section visible.

3. showPublished()**:** It will be called on clicking the publish button. This will change the value of "activeTab" flag to "published".This will inturn resolve the statement defined in the html for *ngIf in the published section to return true, making the published section visible.

4. createNewBlog(): It will initialize the process for creating a new blog.It will make the blog dashboard's blog editor visible which will be populated with the default values for the blog field.

- **Note: Only the blog cards of the active tab will be loaded initially. But once loaded they will remain in the view. This will ensure no unnecessary calls are made. [example -If the user does not shift to drafts section no draft blog cards will be loaded. Hence reducing loading speed on the initial page load and also there will be no unnecessary data in the memory store.] Thus according to active tab value, call will be raised to blog-dashboard.service.ts**

3. **Inside blog-dashboard.component.spec.ts:**

It will contain test functions which will ensure that the .ts file is completely and efficiently tested.

4. **Inside blog-dashboard-backend-api.service.ts:**

   It will have a class that will contain functions which will provide data to the frontend from backend.These functions will place http requests to controllers. Both post and get requests will be placed in order to retrieve and store data from and to the datastore.
   The class name will be "**BlogDashboardBackendApiService**".
   It will have the following functions:

   - _fetchBlogDashboardDataAsync() : It will return a promise containing data to be displayed on the dashboard. When the blog dashboard is loaded by the user it will place a request on "/blog_dashboard/data" handler.

   - _fetchBlogCardsAsync() : Taking user-id as parameter it will place a get request to the "FetchBlogsHandler" in the controller layer.

   - _fetchBlogPostPageDataAsync() : It will take blog-id as parameter and will place a get request to "BlogPostHandler".

   - deleteBlog : parameter will be blog-id and will place a delete request in the "BLOG_DASHBOARD_DATA_URL"

   - updateBlog : parameter will be blog-id and will place a put request in the "BLOG_DASHBOARD_DATA_URL"

   - _createNewBlogIdAsync() : It does a POST request to the backend () to actually create the Blog. This returns the id of the created blog. Now, the url is redirected to '/blog_editor/create/<id>' using UrlInterpolationService. This URL is then redirected by the backend to render the html page at blog_editor folder to show the blog-editor.

   - _createBlogSummaryAsync():It does a POST request to the backend () to actually create the Blog summary.

   - _createBlogDataAsync(): It will perform a post request with the data for "BlogDataModel" to the handler.

   - _fetchBlogTagsListAsync(): It will perform a get request to retrieve all the pre-defined tags which can be assigned to blogs.

   - publishBlogs() : It will update the blog to the backend with content and will make the blog status- "publish".

5. **Inside blog-dashboard-backend-api.service.spec.ts:**
   It will contain test functions that will ensure that the .ts file is completely and efficiently tested.

6. **Inside blog-dashboard.service.ts:**
   It will have all the other necessary functions required for the blog dashboard such as "fetching_user_data" to be displayed on the dashboard. It will also be used to provide functions required for routing between blog editors , the blog dashboard and the blog post page.

7. Inside blog-dashboard.service.spec.ts:

It will contain test functions which will ensure that the .ts file is completely and efficiently tested.

8. Inside blog-editor folder:

As stated above in the product design section, In order to avoid mingling of features and plugins between RTE and the blog-editor, I will be directly using the cke-editor. To achieve this new files will be created
New files -

- blog-cke-editor.component.html
- blog-cke-editor.component.ts
- blog-cke-editor.component.spec.ts
- blog-cke-editor-widget.initializer.ts

- main-blog-editor.component.html
- main-blog-editor.component.ts
- main-blog-editor.component.spec.ts

The code inside the first 4 files will be similar to the files inside ‘core/templates/components/cke-editor-helper’ present in our codebase.

Toolbar configuration in will be as follows in the blog-cke-editor.component.ts to load plugins and features :

```
CKEDITOR.editorConfig = function( config ) {
    config.toolbar = [

        { name: 'clipboard', items: [ 'Cut', 'Copy', 'Paste'] },
        { name: 'editing', items: [ 'Find', 'Replace', '-', 'SelectAll', '-', 'Scayt' ] },
        { name: 'basicstyles', items: [ 'Bold', 'Italic', 'Underline', 'Strike', 'Subscript', 'Superscript',
'-'] },
        { name: 'paragraph', items: [ 'NumberedList', 'BulletedList', '-', 'Outdent', 'Indent','-',
'JustifyLeft', 'JustifyCenter', 'JustifyRight', 'JustifyBlock', '-', 'BidiLtr', 'BidiRtl', 'Language' ] },
        { name: 'insert', items: [ 'Image', 'Table', 'Smiley', 'SpecialChar','Iframe', 'EmojiPanel' ] },
        '/',
        { name: 'styles', items: [ 'Styles', 'Format', 'Font', 'FontSize' ] },
        { name: 'colors', items: [ 'TextColor', 'BGColor' ] },
    ];
};
```

Inside main-blog-editor.component.html:

1. It will first contain the button to upload the thumbnail image. It will call “uploadThumbnail()” function in the ts file.Its functionality can be borrowed from the current topic thumbnail uploader.
2. It will then have the blog-cke editor.
3. It will have a side pane in which tags under which blog can be categorized will be visible. Clicking on each tag will call a function addTag() which will append the name of the tag in the tags list in the ts file for the blog.

4. It will have " save as draft " and "publish" buttons which will call saveDraft() and publishBlog() functions respectively in its ts file.
5. It will have an "eye" icon which will call the loadPreview() function.This will make the preview of the blog card visible.

Inside main-blog-editor.component.ts:

It will contain typescript functions which will serve as the backbone of the blog editor.

Some functions inside it are-

1. ngOnInIt(): It will set the default values of variables and will call the necessary functions required to initialize and populate the values on the page.
   - availableTags : It will be containing the list of predefined tags among which the user can choose.
   - *username : It will be the user-name that will be displayed on the editor side pane.*

2. SaveDraft(): It will call the updateBlog function in the blog-dashboard-backend-api.service which will the place necessary requests to the controller to update the blog in the backend.In this case the status of the update blog in the storage model will is "STATUS_BLOG_IS_DRAFT" .

3. publishBlog(): It will also call the updateBlog function in the service but in this case the status of the blog will change to "STATUS_BLOG_IS_PUBLISHED".

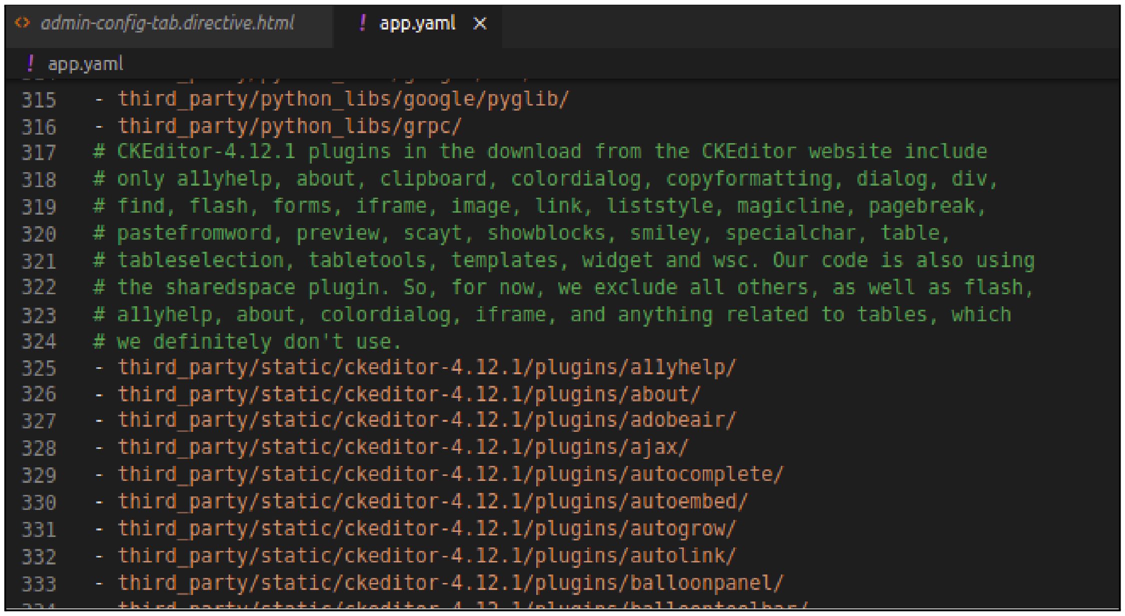
4. loadPreview(): It will make the preview of the blog card visible by providing values to blog-card.component files.

Inside main-blog-editor.component.spec.ts:

It will contain test functions which will ensure that the .ts file is completely and efficiently tested.

Inside app.yaml and app_dev.yaml:

All the plugins required for CKEditor will have to be included in the build hence they will be removed from the list which removes them from build(below is the SS showing that they removed from build).



```
  - third_party/python_libs/google/pyglib/
  - third_party/python_libs/grpc/
  # CKEditor-4.12.1 plugins in the download from the CKEditor website include
  # only allyhelp, about, clipboard, colordialog, copyformatting, dialog, div,
  # find, flash, forms, iframe, image, link, liststyle, magicline, pagebreak,
  # pastefromword, preview, scayt, showblocks, smiley, specialchar, table,
  # tableselection, tabletools, templates, widget and wsc. Our code is also using
  # the sharedspace plugin. So, for now, we exclude all others, as well as flash,
  # allyhelp, about, colordialog, iframe, and anything related to tables, which
  # we definitely don't use.
  - third_party/static/ckeditor-4.12.1/plugins/allyhelp/
  - third_party/static/ckeditor-4.12.1/plugins/about/
  - third_party/static/ckeditor-4.12.1/plugins/adobeair/
  - third_party/static/ckeditor-4.12.1/plugins/ajax/
  - third_party/static/ckeditor-4.12.1/plugins/autocomplete/
  - third_party/static/ckeditor-4.12.1/plugins/autoembed/
  - third_party/static/ckeditor-4.12.1/plugins/autogrow/
  - third_party/static/ckeditor-4.12.1/plugins/autolink/
  - third_party/static/ckeditor-4.12.1/plugins/balloonpanel/
```

FOR BLOG CARD:

1. **Inside blog-card.component.html:**

   It will contain the code for the blog-card which can be used in both blog-dashboard and the blog-homepage. It will have the following parts:
   1. A picture tag for displaying the thumbnail image.
   2. A div containing a picture tag for displaying the user profile image of the author. Along with it in the same line user-name of the author will be displayed. Below it the time of publishing the blog post will be displayed.
   3. A div containing the blog summary text.

On clicking the blog card , a function called “loadBlogPost()” will be called and the page will redirect to ‘blog post’ page of the blog card.

2. **Inside blog-card.component.ts:**

   It will contain the typescript functions such as :

   1. ngOnInIt(): It will set the default values of variables and will call the necessary functions required to initialize and populate the values on the page.
   2. loadBlogPost(): It will pass the blog-id to a function in “blog-dashboard.service.ts” which will load the blog post page for the given id (This function will then call _fetchBlogPostPageDataAsync() in the “blog-dashboard-backend-api.service.ts”).
   [NOTE : BLOG POST PAGE IS A PART OF FUTURE WORK]

3. **Inside blog-card.component.spec.ts:**

   It will contain the unit tests for complete and efficient testing of blog-card.component.ts.

NOTE : Blog homepage ,Blog Post Page, And the statistics tab frontend will be implemented afterwards and is a part of future works.

TO ENABLE ADMIN ADD PRE-DEFINED TAGS:

1. In domain/config_domain.py:
   - In class Registry: A new config property namely “Predefined_Blog_Tags” will be added.
     It will be of the following schema: SET_OF_STRINGS_SCHEMA
     This will automatically introduce the required type of input in the admin page’s config tab.
2. In templates/domain/admin :
   - admin-backend-api.service.ts: In interface ConfigPropertyValues,a new field “Predefined_Blog_Tags” with “string[ ]” value will be added.

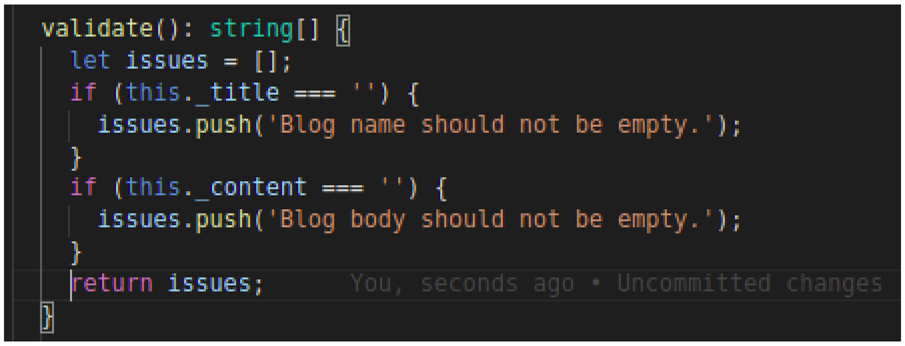
ADDING NEW ROLE:

A new blog-editor role will have to be added. Therefore the following steps will be followed to achieve this:

- As specified in changes made to feconf.py the following role will be added- ROLE_ID_BLOG_EDITOR : 'BLOG_EDITOR'
- In core/domain/role_services.py : Under 'UPDATABLE_ROLES' and 'VIEWABLE_ROLES' it will be added.
    - In 'HUMAN_READABLE_ROLES' corresponding string will be added : "blog editor".
    - In 'PARENT_ROLES' - key and value will be added.
    - In 'ROLE_ACTIONS' - key and value will is added.
    - Actions will be added for the role added -
        1. ACTION_EDIT_OWNED_BLOG = 'EDIT_OWNED_BLOG'
        2. ACTION_PUBLISH_OWNED_BLOG= 'PUBLISH_OWNED_BLOG"
        3. ACTION_UNPUBLISH_OWNED_BLOG = 'UNPUBLISH_OWNED_BLOG'
        4. ACTION_DELETE_OWNED_BLOG = 'DELETE_OWNED_BLOG'
- Corresponding decorator for the role action will be added in controllers/acl_decorators.py

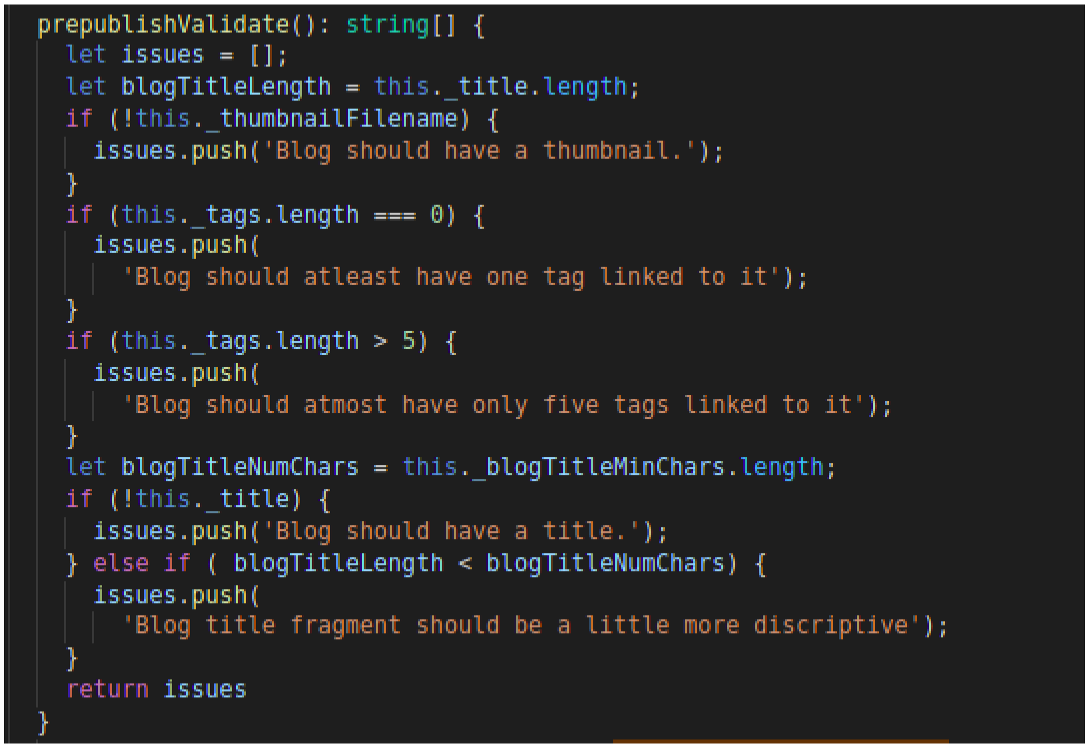
<u>All Validations</u> (front end and backend):

In frontend : Inside blog.model.ts : It is the model class that will create the frontend model for the main blog domain object. It will have a validate() method defined which will validate the fields of the object to be saved as draft. It will also have isTitleValid() and prepublishValidate() method to validate the fields of blog being published.



```
validate(): string[] {
  let issues = [];
  if (this._title === '') {
    issues.push('Blog name should not be empty.');
  }
  if (this._content === '') {
    issues.push('Blog body should not be empty.');
  }
  return issues;        You, seconds ago • Uncommitted changes
}
```

validate() method will ensure that the title and the content fields are not empty when the blog is being saved(as draft).



```
prepublishValidate(): string[] {
  let issues = [];
  let blogTitleLength = this._title.length;
  if (!this._thumbnailFilename) {
    issues.push('Blog should have a thumbnail.');
  }
  if (this._tags.length === 0) {
    issues.push(
      'Blog should atleast have one tag linked to it');
  }
  if (this._tags.length > 5) {
    issues.push(
      'Blog should atmost have only five tags linked to it');
  }
  let blogTitleNumChars = this._blogTitleMinChars.length;
  if (!this._title) {
    issues.push('Blog should have a title.');
  } else if ( blogTitleLength < blogTitleNumChars) {
    issues.push(
      'Blog title fragment should be a little more discriptive');
  }
  return issues
}
```

prepublishValidate() method will ensure that the blog has at least one tag and at most five tags selected. It will ensure that a thumbnail is selected and that the title is a little descriptive (by validating that it is longer than a certain number of chars).

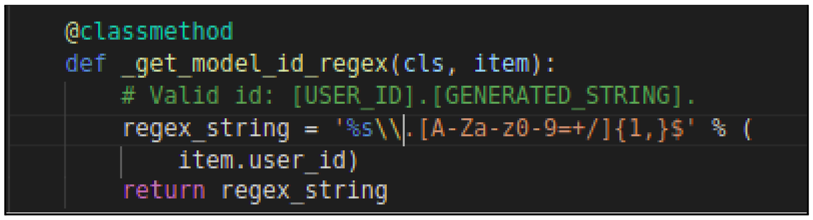
The function isTitleValid() will use is_blog_title_taken() in the blog_services.py to validate the title only if it is unique.

In backend:

1. Inside class blog in blog_domain.py - validate() function will exist that will validate the properties of the blog domain object.
   It will validate the **author_id,title,content** and **url_fragment** by ensuring that they are **string** and also that they are **not empty**. **url_fragment** will also be validated when it is **unique**. Inside the **blog_services.py** it will be ensured that whenever a new title is being stored it is **unique**.
2. Inside class blogSummary in blog_domain.py - validate() function will exist that will validate **author_id, title** and summary are not empty and are strings.
   The thumbnail_filename will also be validated using "utils.require_valid_thumbnail_filename(thumbnail)".It will also validate the tags to be a list where each tag is a string.
   The length of summary will also be checked and validated by ensuring that it doesn't exceed the set limit of characters.
   In the service layer we will ensure that all the selected tags are a part of the list of tags set by the admin.
3. Inside classblogRights , validate() will check if the editor_id is string and is not empty and that the status is a boolean value.
4. Inside blog_validator.py there will be 3 classes to validate their respective models.
   All three will be using "BaseModelValidator)

   Inside blog_validators.py :

- BlogModelValidator: Class will have functions to validate general blog models. It will have functions like:
    1. _get_model_id_regex:



```
@classmethod
def _get_model_id_regex(cls, item):
    # Valid id: [USER_ID].[GENERATED_STRING].
    regex_string = '%s\\.[A-Za-z0-9=+/]{1,}$' % (
        item.user_id)
    return regex_string
```

    2. _get_external_id_relationships: function will ensure that correct ids are returned when external models are called.
    3. _get_model_domain_object_instance: To get the model object instance.

- BlogSummaryModelValidator: Class will have functions to validate general blog summary models. It will have functions like:
    1. _get_external_id_relationships: function will ensure that correct ids are returned when external models are called

2. _validate_has_summary: validates the summary of the published blog.It should not be empty.
3. _validate_model_time_fields: Validates the last_updated and created_on fields of a model instance.
4. _get_model_id_regex: Will ensure that the id is in the expected form.

- BlogRightsModelValidator:
  1. _get_model_id_regex: Will ensure that the id is in the expected form.
  2. _get_external_id_relationships: Function will ensure that correct ids are returned when external models are called.
  3. validate_blogrights_blog_ids_in_author_blog_ids : A validator that can be written to check that all the blogs for a given user in the blogsRights model have there blogIds in the userContributionsModel for the given user.
  4. _get_custom_validation_functions: It will return "_validate_blogrights_blog_ids_in_author_blog_ids".

Testing Approach

All the files and functions that will be introduced will be thoroughly tested.

1. **End to End tests:**
   All automated tests will be written for both blog dashboard will be inside **blogPage.js** in tests/protractor_utils and **blog.js** in tests/protractor_desktop.

2. **Karma tests:**
   All the component.ts and services.ts files will be accompanied by their respective spec files. This will ensure all the frontend files are tested.

3. **Backend tests:**
   All the backend files will be accompanied by their test files.

4. **Lighthouse tests:**
   All the new pages url will be added to the lighthouserc.js, and lighthouserc-accessibility.js to perform accessibility tests on the webpage.

MIGRATING BLOGS FROM MEDIUM TO OPPIA:

I will be doing a manual transfer as medium though allows to export blog but renders the content of the blog in a form of html file.Thus a manual transfer will be far more simpler whereas a trying an automated approach will require other functionality in the codebase.The fact that there are some free plugins available which allow rendering of medium blogs support wordpress editor format after being provided the html files in zip format(which is downloaded from medium itself) will contain few unnecessary fields as claps, feedback thread which aren't being handled by the oppia's blog interface.Moreover, the achieved formatting won't be same as that expected. **Our editor might not support the html tags the way intended as we do not support the features of the medium's blog editor in the same format.** Automated transmission might sound easy to achieve and will definitely require less time but can have major unseen issues.Manual transmission will not require more than 4 days and will be quite easy to achieve.Also manual transmission guarantees that all blogs will be shifted with no formatting errors which is hard to promise in an automated transmission.

Milestones

The parts of the product that will be completely covered within the GSoC timelines are

1) BLOG DASHBOARD [Complete backend and frontend except frontend of statistics tab]
2) BLOG EDITOR [Both frontend and backend]
3) BLOG HOMEPAGE [Only Backend will be done[controllers], No frontend will be done]
4) BLOG-CARD
5) MIGRATING BLOGS FROM MEDIUM TO OPPIA

Community Bonding Period:

In the community bonding period, I will finalize all the technical and design details with my mentor. I have been contributing to this community for more than 6 months now, I'm much familiar with the workflow in the codebase. Thus, I will start working on the project, if my Mentor allows, to ensure I will be able to cope up with any difficulties in the future.

Milestone 1: Complete storage models and other related backends along with their tests for blog dashboard and editor.

The coding period begins from **7th June** and ends on 12th of July.

Key Objective:

1. In the first milestone storage model for blogs and changes in user storage model will be done.
2. All the domain files and controllers will be added along with the validators for all the new files as well as for those modified.
3. Admin page's Config tab will be modified to enable the admin to add predefined tags.All related changes in backend and front end will be completed.

No.	Description of PR	Prereq PR num	Target date for PR submission	Target date for PR to be merged
1.1	Adding blog and user storage models in core/storage and enlisting them in platform/models.py to make them valid. ○ As a result 3 new files inside a folder named blogs in the core/storage folder are added. ○ User storage file has a new model.	none	10th June	16th june E.O.D

1.2	Adding domain files for blog storage models i.e: 1) blog_domain.py 2) blog_services.py 3) blog_validators.py 4) blog_jobs_one_off.py(EMPTY FILE) These will be accompanied by their tests. ○ As a result 8 new files will be added (4 above mentioned files + 4 test files of these files.)	1.1	16th June	22nd june E.O.D
1.3	Modifying domain files for adding new model in user's storage models.Files that will be modified in it will be: 1) user_domain.py 2) user_services.py 3) user_validators.py These changes will be accompanied by their testing in their respective test files.	none	18th June E.O.D	23rd June E.O.D
1.4	New role blog editor will be added and role action will added, accordingly feconf.py will be modified Changes to controllers/acl_decorators.py Controllers for the blog dashboard will be done in this PR. 2 new files will be added in core/controllers namely ○ blog_dashboard.py ○ blog_dasboard_test.py	1.2 1.3	25th June	30th June E.O.D
1.5	Modifications in search_services.py will be done to build search handler for blogs.	1.1 1.4	28th June	2nd July E.O.D
1.6	Controllers for blog homepage will be done in this PR 2 new files will be added in core/controllers namely ○ blog_homepage.py ○ blog_homepage_test.py	1.2 1.3 1.5	2nd July	6th July E.O.D
1.7	Enabling Admin to add predefined tags via the config tab. Changes in the following files will be done: ○ config_domain.py ○ config_validators.py ○ admin-backend-api.service.ts	none	4th July E.O.D	8th July

	As a result both frontend and backend for the config tab in the admin page are done. All the introduced fields will have their tests in their respective tests file.			
1.8	All reported errors and bug fixes will be done.		6th-11th	12th July

MILESTONE ENDS ON 12TH JULY.
JULY 16TH IS THE MILESTONE'S EVALUATION DEADLINE.

Milestone 2 : Complete frontend with their tests for blog dashboard and editor. Making it accessible to the users via drop-down.

The milestone begins on 13th July and ends between 16th-23rd August.

Key Objective:

1. In this milestone all the frontend files for the blog dashboard and the editor will be implemented.
2. Frontend of blog-card will be done to load the blog-cards in the blog dashboard.
3. Blogs from the medium will be exported to oppia.

Last week is kept empty to handle bugs in the project.
All the PRs from milestone one will be required for PRs in milestone two.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Basic Structure for Blog Interface will be created Empty folders will be added. Blog folder under core/templates/pages will have 3 folders namely- blog-dashboard, blog-homepage and blog-card. ● Inside blog-dashboard, in blog-editor folder blog-cke-editor files will be done. ○ blog-cke-editor.component.html ○ blog-cke-editor.component.ts ○ blog-cke-editor.component.spec.ts ○ blog-cke-editor-widget.initializer.ts ● All the required plugins will be included in build and therefore **app_dev.yaml and app.yaml** will be modified.	none	18th July E.O.D	23rd July E.O.D

2.2	Services required for the blog dashboard are done. Following files inside blog/blog-dashboard in core/templates/pages will be added: ○ blog-dashboard-backend-api.service.ts ○ blog-dashboard.service.ts These will be accompanied by their test files.		21st July	26th July E.O.D
2.3	Blog-Card frontend is complete.	2.2	24th July	29th July E.O.D
2.4	Blog-Editor frontend is complete. ○ main-blog-editor.component.html ○ main-blog-editor.component.ts ○ main-blog-editor.component.spec.ts	2.1 2.2 2.3	27th July E.O.D	1st August
2.5	Blog-Dashboard frontend is complete. Following new files will be created: ○ blog-dashboards.component.html ○ blog-dashboard.component.ts ○ blog-dashboard.component.spec.ts ○ blog-dashboard.component.mainpage.html	2.1 2.2 2.3 2.4	2nd August	7th August
2.6	Adding E2E tests and making the blog-dashboard accessible from the drop-down. Adding new pages to lighthouserc.js and lighthouserc-accessibility.js.	2.1 2.2 2.3 2.4 2.5	5th August	10th August
2.7	Adding all the blogs present on medium to Oppia (NO CODE MAJOR CODE ADDITION PR WILL BE RAISED THUS BUG AND ERRORS CAN BE ALSO FIXED IN THIS TIME)		(a period of 5 days(7-13)to handle any unseen problem in manual transmission)	13th August
2.8	All errors and bug fixes in the project will be done.			18th August

With the current target date , I can make my final submission on the 19th. The final last submission date is 23rd August. Thus in the worst case I will still have a period of 4 days to complete and fix things.

Optional Sections

Privacy

No new user data will be collected. Already saved username and profile picture are the sensitive information being used in the blog-dashboard,blog card and the blog homepage to show author details.However this is the data already added by the user.

Security

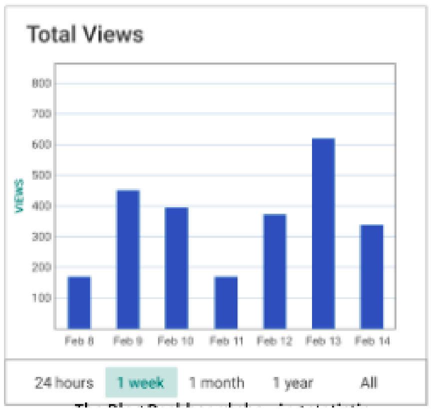
No security issues are related as such. Users can only view the name and profile picture of the author and the date of publishing of the blog. Thus no security issues will arise due to the project.

Accessibility (if user-facing)

The project will enable the team members to share their thoughts and stories by directly using the blog dashboard and thus not going for any other site. Using the blog-editor, users will be able to write engaging blogs. After the features mentioned in “Future Work” are implemented all the users of Oppia will be able to know about the latest features of the site, and the stories of the team-members , volunteers and students from our website itself.

Future Work

I am keenly interested in completing the blog interface. Thus after this project is completed , Blog-homepage and statistics tab as shown in the product design will be implemented.
In the statistics tab in the blog-dashboard, count of views on each blog and collectively count on all the blogs will be represented graphically.Representation of total views is shown in the picture.



Blog-Homepage will be implemented so that all the users are able
to read the blogs written by our members. It will be made accessible from the About drop-down as stated in the product design section.

Also in future , I will make both blog homepage and blog dashboard accessible for all devices with all kinds of screen sizes.