

# Google Summer Of Code

## 2022 Proposal

# Blog Integration

by Rijuta Singh

<b>Section 1 : About Me</b>	<b>2</b>
<b>Section 2: Proposal Details</b>	<b>5</b>
Problem Statement	<b>5</b>
<b>Section 2.1: WHAT</b>	<b>6</b>
Technical Requirements	7
<b>Section 2.2: HOW</b>	<b>11</b>
Existing Status Quo	11
Solution Overview	12
Implementation Approach	15
Storage Model Layer Changes	16
Domain Layer	27
User Flows (Controllers and Services)	35
Web frontend changes	40
Documentation changes	49
Testing Plan	49
<b>Migration Of Blog Posts From Medium to Oppia</b>	<b>50</b>
<b>Implementation Plan</b>	<b>51</b>
Milestone 1 Table	51
Milestone 2 Table	52
Launch Plan	53
Future Work	54

## ABOUT ME:

What project are you applying for?

*I am applying for - Blog integration.*

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

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! I've always felt strongly about giving back to society and that it is one of my responsibilities to do my personal best to assist others who have been less fortunate. I had vowed from a young age to do something for the impoverished children in my community so that they may at least study and receive an education. I feel I am being educated in order to help and empower those kids.

Oppia is a community of learners and teachers dedicated to assisting everyone in learning whatever they choose in a fun and productive manner. Oppia's mission is to make the world a better place. It provides a barrier-free environment for gaining and sharing knowledge. As a result, it became my home. To find an organization like Oppia in Google Summer Of Code was something I had been waiting for.

Contributing to Oppia has increased my technical skills while also teaching me the value of working as part of a team and assisting other contributors. For the reasons it serves and the atmosphere it provides to its contributors and users, I would like to continue contributing to Oppia even after the GSoC time ends.

This project appeals to me because it is a full-stack project aimed at integrating blogging features into the Oppia web platform apart from the fact that the first chunk of the project was implemented by me in GSoC' 2021. I thought I would be able to implement the second chunk of it as soon as I finished with GSoC'2021. However, conditions and time limitations did not allow it. I found an opportunity to finish what I have started with GSoC'2022 driving. This project will not only improve the website's content, but it will also make it easier for all of its contributors, learners, and volunteers to share their experiences with the rest of the world directly from the Oppia website. This project requires me to write code for both the frontend and backend, and it also allows me to participate in the design process and finally owning code for a complete feature!

### Prior experience

Oppia was my first open-source organization, I started contributing to and is the only one till now.

I started around November 2020 and was soon selected as a member of the LaCE team and completed GSoC 2021 project - [Integrating the Oppia blog with Oppia.org](#).

I have more than 50 successfully merged PRs. These PRs are concerned with work in Python, Typescript, Angular, AngularJs, HTML/CSS, E2E tests and beam jobs. Now I feel I have a good command over Oppia's codebase though there are still many parts of it left for me to discover. I have become familiar with the approaches that the organization follows for tackling issues and various kinds of bugs.

I've been contributing to Oppia as the LaCE quality team lead since I completed GSoC'2021,. I've tried to review PRs from new contributors related to the team. I've also participated in release testing, which has given me a lot of experience with the website from the user's perspective.

S.No	PR number	PR Description
1.	<a href="#">#14523</a>	<i>This PR focussed on fixing the voiceover drop-down so that languages names appear in their respective language and also had other bug fixes. - Frontend.</i>
2.	<a href="#">#14685</a>	<i>l18N keys were added for the preference page - Internationalization.</i>
3.	<a href="#">#13683</a>	<i>This PR focussed on adding e2e tests for the complete Blog Dashboard functionality. - End to End testing.</i>
4.	<a href="#">#13278</a>	<i>This PR added a controller layer for the Blog Homepage - Backend.</i>
5.	<a href="#">#13232</a>	<i>This PR added blog model validations according to apache beam structure. - Apache Beam Jobs - Backend.</i>
6.	<a href="#">#11467</a> and <a href="#">#12038</a>	<i>The new teach page which was a part of the integration of Oppia.org with the Oppia Foundation Website was implemented in #11467 whereas responsiveness bugs of the page were fixed in #12038 - Frontend</i>

## Project size

The project is medium sized ( about 175 hours ).

## Project timeframe

I will be utilising an extended coding period for my project. I will be working from July 20, 2022 to Oct 20, 2022. I have my summer internship starting from mid-May and ending in mid-July. Thus to avoid clashing of project coding period and internship, I wish to utilize the extended coding period completely.

## Contact info and timezone(s)

Timezone: I will stay in India throughout the period of my project. The time zone will be Indian Standard Time (GMT+5:30).

### Contact:

Mail: [rijuta\\_s@me.iitr.ac.in](mailto:rijuta_s@me.iitr.ac.in)

Mobile no. : +91- 8699815957 (Whatsapp)

Github Profile: [Rijuta-s](#)

## Time commitment

I will be able to give around 6 hours per day starting from July 25, 2022 to August 31, 2022. After August 31, 2022 I will be able to give roughly around 4 hours per day till the end of my project that is Oct 20, 2022 due to

college semester engagements. However, taking into consideration the size of project that is roughly around 175 hours, I should be able to complete the project well within time.

## Essential Prerequisites

```
Tasks still running:
  core.controllers.editor_test (started 00:11:43)
-----
[datastore] Apr 02, 2022 12:11:56 AM io.gapi.emulators.grpc.GrpcServer$3 operationComplete
[datastore] INFO: Adding handler(s) to newly registered Channel.
[datastore] Apr 02, 2022 12:11:56 AM io.gapi.emulators.netty.HttpVersionRoutingHandler channelRead
[datastore] INFO: Detected HTTP/2 connection.
18:42:56 FINISHED core.controllers.editor_test: 72.4 secs
Stopping Redis Server(name="sh", pid=50086)...
Stopping Cloud Datastore Emulator(name="sh", pid=49995)...

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

SUCCESS   core.controllers.editor_test: 92 tests (59.1 secs)

Ran 92 tests in 1 test class.
All tests passed.

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

- I am able to run a single backend test target on my machine. (Show a screenshot of a successful test.)

```
System.import() is deprecated and will be removed soon. Use import() instead.
For more info visit https://webpack.js.org/guides/code-splitting/
@ ./node_modules/@angular/platform-browser-dynamic/fesm2015/testing.js 7:0-169 42:12-22 45:44-50 109:31-41 115:84-92 121:89-98 127:10-18
@ ./core/templates/combined-tests.spec.ts
i [wdm]: Compiled with warnings.
02 04 2022 00:18:38.601:WARN [filelist]: Pattern "/third_party/static/lamejs-1.2.0/worker-example/worker-realtime.js" does not match
02 04 2022 00:18:38.601:WARN [filelist]: Pattern "/home/rijuta/oppia/oppia/extensions/interactions/**/*.directive.html" does not match
Chrome Headless 99.0.4844.84 (Linux x86_64): Executed 7370 of 7370 SUCCESS (2 mins 39.296 secs / 2 mins 20.843 secs)
TOTAL: 7370 SUCCESS
TOTAL: 7370 SUCCESS
02 04 2022 00:21:37.112:WARN [launcher]: ChromeHeadless was not killed in 2000 ms, sending SIGKILL.
Done!
```

- I am able to run all the frontend tests at once on my machine. (Show a screenshot of a successful test.)

```
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 preferred audio language of the learner
.   ? should change preferred 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.
```

- I am able to run one suite of e2e tests on my machine. (Show a screenshot of a successful test.)

## Other summer obligations

I have my summer internship starting from mid of May to mid of July. Hence I prefer to work in extended coding period that is from July 20, 2022. From 10th August, my college will resume however the work load in college will be light in August and will demand only 3-4 hours per day, 5 days a week. Hence I will be able to work easily on my project till August 31, 2022. In September, I will have my mid term examinations which will go on for about a week. So I might get a bit slow for 15 days, but will pick up pace in the following mid term holidays working around 7-8 hours per day. In October , college will remain light allowing me to work on my project easily.

## Communication channels

Meeting with mentor: 2 times per week (flexible) [ Google-Meet or any other platform ]

I will try to provide daily updates as much as possible to ensure smooth working through out the project..

[ I check my email regularly and can be contacted on any of the above-mentioned platforms.]

# Section 2: Proposal Details

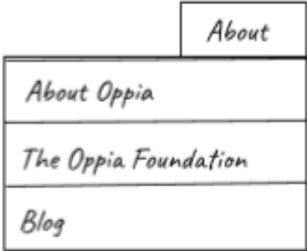
## Problem Statement

<b>Link to PRD (or N/A if there isn't one)</b>	N/A
<b>Target Audience</b>	Everyone ! Blog posts will be used to spread information about oppia and its volunteers' efforts across all social media platforms, specially on the website itself. Thus, Oppia users, developers, and volunteers will not be the only ones reading blog entries on the Blog Homepage but the entire world!
<b>Core User Need</b>	Oppia.org's blogs are currently hosted on Medium, a distinct website. Having all of the blog posts on Oppia's website will allow team members and users to share their tales with the rest of the world from our website itself. This will help readers to view our website without much difficulty or confusion. It will, in short, assist our marketing and advertising team in driving traffic to our website, as well as saving our readers time if they desire to traverse the website after viewing it.  Currently, Blog Dashboard functionality exists in our codebase allowing authors to write blog posts, however they are not accessible to other people. Also, authors can by no means see the number of views of their blog posts as no such functionality of tracking views is inculcated in the codebase making it difficult for the writers to know the impact and likeability of their content.

<b>What goals do we want the solution to achieve?</b>	<p>One of our key objectives is to increase traffic to our website and to give a hassle-free experience for our blog post readers! It will be a simple approach for new users who come to read blog posts to find their way to our website, eliminating friction.</p> <p>Our developers, volunteers, and other users will be able to share their thoughts and experiences directly from our website.</p> <p>A statistics tab to allow blog post authors to have an insight about how many people read and viewed their blog posts.</p>
---	--

## Section 2.1: WHAT

### Key User Stories and Tasks

#	Title	User Story Description (role, goal, motivation) <i>"As a ..., I need ..., so that ..."</i>	Priority <sup>1</sup>	List of tasks needed to achieve the goal (this is the "User Journey")	Links to mocks / prototypes, and/or PRD sections that spec out additional requirements.
1.	Reading a blog post	As a blog post reader, I should be able to navigate to Blog Home Page via navbar and footer, and should be able to read the blog post of my choice selecting from the available list of blog posts.	Must Have	Selecting the 'Blog' navbar item from 'About' drop down to land on the 'Blog Home Page' .	
				Users should be able to scroll freely and use pagination , filter blog posts by tags and perform searches for them on the Blog Home Page.	<a href="#">Link to Blog Home page</a> <a href="#">Link to Blog Search Results/Filter By Page</a>
				Clicking on the desired blog post card on the Blog HomePage / Search Results page should take you to the respective blog post page.	<a href="#">Link to Blog Post Page</a> <ul style="list-style-type: none"> <li>- Navbar will remain the same as it is on the other parts of the site.</li> </ul>
2.	Visiting Author Profile Page	As a blog post reader, I should be able to visit the author profile page, read the author's bio and see all the blog posts written by the author.	Should Have	Navigate to any blog post page.	<a href="#">Link to Blog Post Page</a>
				Click on Author's Profile Picture or Name to land onto Author's profile page.	<a href="#">Link to Author Page</a>
				Use pagination to see all the blog posts written by the author in chronological order and clicking on any blog post card	

				should navigate to the blog post page.	
3.	Viewing Stats	As a blog post editor/ admin, I should be able to visit the statistics tab of the blog dashboard. I should be able to see the total number of views, reads and reading time on each blog post along with the total number of views, reads and reading time on all the blog posts by me.	Must Have	Login as Blog Post Editor / Blog Admin	
				From Profile dropdown, navigate to blog dashboard.	<a href="#">Link to Statistic tab</a>
				Select the Statistics tab to see stats of each blog post. <ul style="list-style-type: none"> <li>Number of views on each blog post as well as on all the blog posts written by me</li> <li>Number of reads on each blog post as well as on all the blog posts written by me</li> <li>Number of users vs Reading time of blog posts on each blog post as well as on all the blog posts</li> </ul>	<ul style="list-style-type: none"> <li>Views: <a href="#">Link</a></li> <li>Reads: <a href="#">Link</a> (Formatting will be same as views chart: Position of Views, Reads and Reading time buttons)</li> <li>Users vs Reading time: <a href="#">Link</a></li> </ul>

## Technical Requirements

### DESCRIPTIONS OF VARIABLE NAMES / CONSTANTS USED :

- author\_publicly\_viewable\_name : publicly\_viewable\_name of the author that he/she can input from the blog dashboard. It will be the same as the username of the author that they enter when they sign up until edited by the user on the blog dashboard. It can be changed from the user preferences page by the author.
- author\_username: username of the author entered while they log in /sign up in Oppia.
- blog\_post\_url : blog post url is generated using blog post title and its unique id which is a 12 character randomly generated hash.

```

466 def generate_url_fragment(title, blog_post_id):
467     """Generates the url fragment for a blog post from the title of the blog
468     post.
469
470     Args:
471         title: str. The title of the blog post.
472         blog_post_id: str. The unique blog post ID.
473
474     Returns:
475         str. The url fragment of the blog post.
476     """
477     lower_title = title.lower()
478     hyphenated_title = lower_title.replace(' ', '-')
479     lower_id = blog_post_id.lower()
480     return hyphenated_title + '-' + lower_id

```

For example if the title of the blog post is 'Global Education and Oppia' and its blog post id is '12DE45FGuthE' then the url fragment will be : '/global-education-and-oppia-12de45fguthe'

#### ALREADY PRESENT CONSTANTS IN CODEBASE:

These constants are already present in feconf.py :

1. BLOG\_HOMEPAGE\_URL : '/blog'
2. BLOG\_HOMEPAGE\_DATA\_URL : '/blogdatahandler/data'
3. AUTHOR\_SPECIFIC\_BLOG\_POST\_PAGE\_URL\_PREFIX : '/blog/author'

#### Additions/Changes to Web Server Endpoint Contracts

	Endpoint URL	Request type (GET, POST, etc.)	New / Existing	Description of the request/response contract (and, if applicable, how it's different from the previous one)
1.	/blog	GET	New	This get request will render the template of the blog homepage if the blog homepage is accessible.
2.	/blog/author/<author_username>	GET	New	This get request will render the template of the author profile page if the blog homepage is accessible.
3.	/blog/<blog_post_url>	GET	New	This get request will render the template of the blog post page.
1.	/blogdatahandler/data	GET	Existing	<p>This get request will populate data on the Blog HomePage. It will fetch all the published blog post summaries which will be used to populate blog post cards on the blog homepage in chronological order.</p> <p>It currently misses the total number of published blog posts.</p> <p>Final response dict contains the following data fields -</p> <ol style="list-style-type: none"> <li>1. list_of_default_tags : the list of default tags to be used in the tags filter on the blog homepage.</li> <li>2. blog_post_summary_dict : a list of published blog post summary dicts in order of date published which will be used to populate blog post cards.</li> <li>3. total_published_blog_posts : number to be displayed on the top the blog post page</li> </ol>
4.	/blogpostdatahandler/<blog_post_url>	PUT	New	This request will update the number of views, number of reads and reading time summary models and raw event log models based on the type of payload.
5.	/blog/searchhandler/data	GET	New	<p>This get request will be used in order to filter blog posts by tags or to search for them using keywords in title.</p> <p>Webpage URL will be :</p> <p>Arguments:</p>



	/blog/search/find?q=<search_query>&tags=<tags>			q: string ( search query string ) tags: list of strings offset: int ( search offset )
6.	/blogdashboardhandler/data/statistics	GET	New	This get request will fetch data for the statistics tab on the blog dashboard page from summary models using blog_post_id and type of chart required( views, reads, reading time)

## Calls to Web Server Endpoints

#	Endpoint URL	Request type (GET, POST, etc.)	Description of why the new call is needed, or why the changes to an existing call is needed
1.	/blogpostdatahandler/<blog_post_url>	GET	It will fetch blog post data to be visible on the blog post page.  Response dict contains the following fields - <ul style="list-style-type: none"> <li>1. profile_picture_data_url : Profile picture of the author of the blog post.</li> <li>2. blog_post_dict : A dict containing data fields that will be used to display the blog post's content.</li> <li>3. summary_dicts : A list of summary dicts that will be used to blog post cards in the 'Suggested for You' section.</li> </ul>
2.	/blog/authordatahandler/<author_username>	GET	This get request will fetch data to populate author profile page.  Response dict contains the following data fields - <ul style="list-style-type: none"> <li>1. author_name : Username of the author to be displayed.</li> <li>2. profile_picture_data_url : url to fetch profile picture of the author.</li> <li>3. author_bio : Bio of the author entered by them on the user profile page.</li> <li>4. summary_dicts : a list of published blog post summary dicts of the blog posts written by author in order of date published which will be used to populate blog post cards on the author profile page.</li> <li>5. total_blog_posts_count : Total number of blog posts by the author.</li> <li>6. offset: search offset to get the next group of blog posts.</li> </ul>
3.	/blogdashboardhandler/data	GET	This get request will be modified to get 'publicly_viewable_name' in the response dict instead of username of the author.
4.	/blogdashboardhandler/data	PUT	The put request will be added to the handler to update 'publicly_viewable_name' in the 'UserSettingsModel'

## UI Screens/Components

Note : The [blog post card](#) have already been implemented in the first part of the Blog Integration Project. Minor changes have to be done to include blog post tags in the blog card. After blog post tags are added to the blog cards, It will look like this: [link](#)

Mobile view mocks : [link](#)

Blog Dashboard (V1) mocks: [link](#)

Blog Dashboard (V2) mocks: [link](#)

#	ID	Description of new UI component	i18n required?	Mock/spec links	A11y requirements
1.	Blog Home Page	<p>The main landing page of the blog. It will be visible to everyone. It will showcase all the blog post cards in chronological order.</p> <ul style="list-style-type: none"> <li>- A maximum of 10 blog post cards will be visible on each page.</li> <li>- A search field to allow user to perform search queries on the blog posts will be present.</li> <li>- A filter to allow user to see blog posts category wise (tags).</li> </ul>	<p>Yes</p> <p>All headings such as - 'Latest Post'; 'Welcome to Oppia' etc, subheadings, button texts ( month names, tags ) and other text.</p>	<a href="#">Blog Home page</a>	<p>Yes</p> <p>Page should be keyboard navigable. All the text should be read by the screen reader.</p>
		<p>The same page will also showcase all the related blog posts in case search is performed or filters are selected ( month published or tags ).</p>	<p>Yes</p> <p>All the heading, subheading , button texts visible on the page.</p>	<a href="#">Blog Search Results/Filter By Page</a>	<p>Yes</p> <p>Page should be keyboard navigable. All the text should be read by the screen reader.</p>
2.	Blog Post Page	<p>The blog post page which will showcase the blog post content, its author and also a 'Suggested for You' section containing cards of other blog posts.</p> <ul style="list-style-type: none"> <li>- Clicking on the share icon will open a popup showing icons to share the link on different platforms such as twitter and facebook and an option to copy the url of the blog post which can then be shared anywhere.</li> <li>- Additionally, based on the author and tags selected by the user, up to two blog recommendations will be displayed at the bottom ('Suggested For You' section). <ul style="list-style-type: none"> <li>- Two latest blog posts</li> </ul> </li> </ul>	<p>Yes</p> <p>Heading - 'Suggested For You'.</p>	<a href="#">Blog Post Page</a>	<p>Yes</p> <p>Share icon should have 'alt text' added, so that everything on the page is read by the screen reader.</p> <p>Page should be keyboard navigable. ( eg. Selecting blog post cards in 'Suggested for You' section. )</p>

		<p>with 1 or more same tags will be shown first. In case no similarity of tags is found then blog posts published by the same author will be shown. Still if 2 blog posts are not present then 2 latest blog posts will be shown.</p> <ul style="list-style-type: none"> <li>- Clicking on the back button in navbar will take the user to the previous page from which they navigated to the page ( that is it can be blog home page, author's profile page or incase if the user comes directly through link, it will take to blog home page )</li> </ul>			
3.	Author Profile Page	<p>The author profile page that contains bio of the author and the blog posts written by him/her in chronological order.</p> <ul style="list-style-type: none"> <li>- Clicking on the blog post card should take the user to blog post page.</li> </ul>	<p>Yes</p> <p>Word - 'Posts' below author name</p>	<a href="#">Author Page</a>	<p>Yes</p> <p>Page should be keyboard navigable. All the text should be read by the screen reader.</p> <p>(eg. Profile image should have alt text.)</p>
4.	Blog Dashboard Statistics Tab	<p>Tab in the blog dashboard page that will for each blog post show the number of views, reads and reading time.</p>	<p>Yes</p> <p>Headings</p>	<a href="#">Statistic tab</a>	<p>Yes</p>

## Data Handling and Privacy

No user sensitive data field is being introduced in the codebase.

---

## Section 2.2: HOW

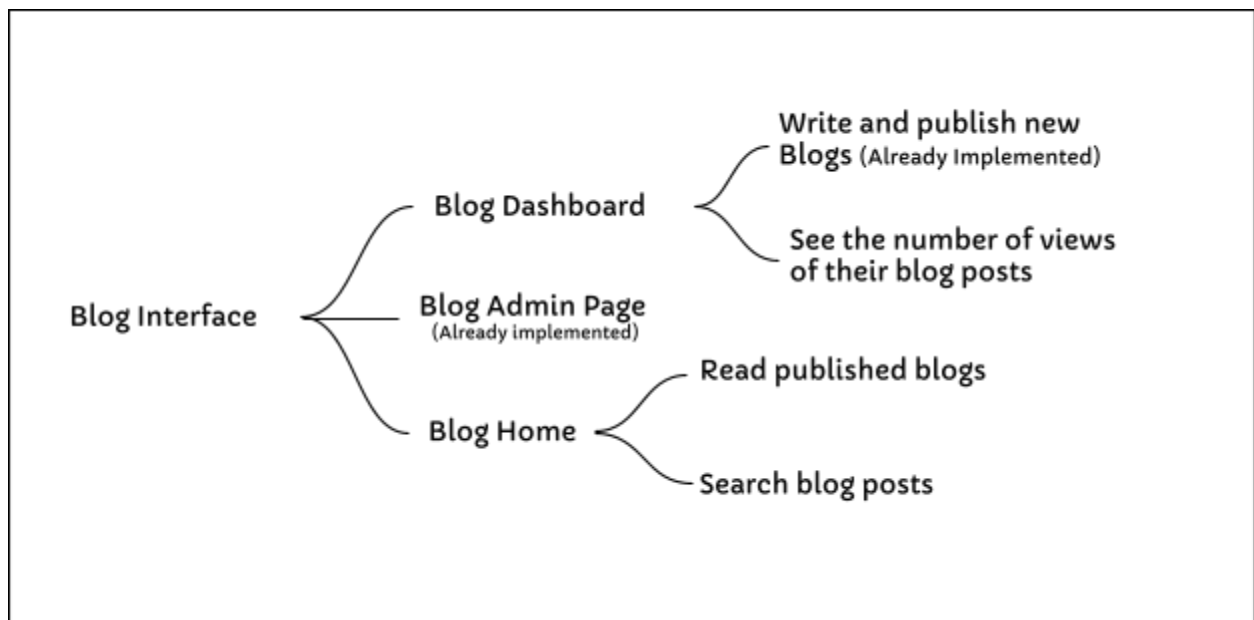
### Existing Status Quo

- Users can currently write blog posts using Blog Dashboard. They can classify their blog posts by choosing tags from a list of options. They can access all of their blog posts, both published and drafts. Blog editors have the ability to modify and delete their blog posts as well.
  - These blog postings, however, are not visible to other people as there is no interface to display them.

- Blog editors cannot see the number of views on their blog posts by any medium way.
- Most of the controllers for blog home page and related pages already exist in the codebase except for search handlers.
- A form to update the published date and author name of the blog posts is present in the Blog Admin Page which will be used to migrate blog posts from 'Medium' manually. [ Implemented in GSoC' 2021 project - [ Milestone 2.4 B : [PR link](#) ] ]
- Before we start using the blog homepage and blog dashboard functionality, issue number - [#13397](#) should be addressed to avoid draft blog posts to be reported in validation checks.

## Solution Overview

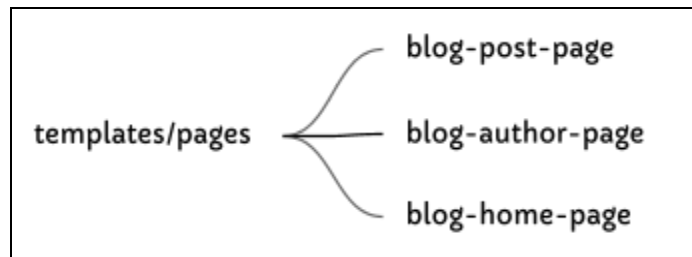
Overall blog interface will have the following structure:



To achieve this -

- An interface to showcase blog posts which is accessible to everyone and allows users to search for blog posts easily and go through them has to be implemented.
  - New handler to perform search for blog posts and filter blog posts will be added in **blog\_homepage.py**.
  - **'main.py'** will be added with new routes.
  - In core/domain, **'blog\_post\_search\_services.py'** will be added and its corresponding test file to allow for searching of blog posts.

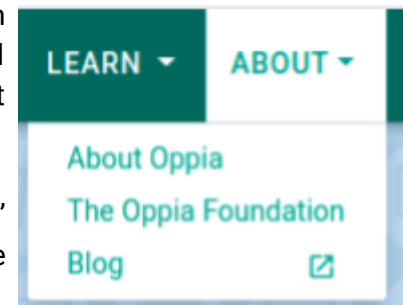
- In jobs/batch\_jobs, **'blog\_post\_search\_indexing\_jobs.py'** will be added along with its test file to index blog posts in elastic search.
- Frontend files for blog home page, blog post page and author profile page will be added.



- A comment will be added in the 'blog-card' component mentioning that it is also being used in 'blog homepage' module. Minor changes will be done in order to include tags on the 'blog-card'
- Inside 'templates/domain/blog', new files will be added :
  1. blog-home-page-backend-api.service.ts
  2. blog-home-page-backend-api.service.spec.ts

- In components/top-navigation-bar/top-navigation-bar.component.html:  
Currently the 'about' dropdown contains a 'blog' nav item which help our users to directly navigate to all the blog posts related to Oppia on Medium and has an icon to let users know that it takes to an external website.

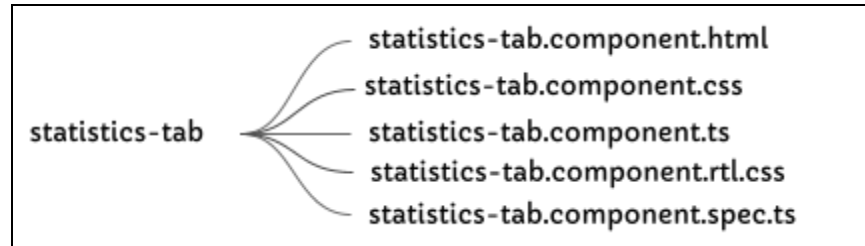
'top-navigation-bar.component.html' will be edited to link 'blog' nav item to the blog-homepage and the icon beside it will be removed.



Current 'About' drop-down

- A statistics tab in the blog dashboard to allow authors to know the number of views on their blog posts with time. Following changes will have to be done :
  - 3 new storage models will be added in user/gae.py to store total number of views, number of reads and number of user vs reading time on all blog posts written by the user.
  - Raw event log storage models will be added in blog-statistics/gae.py to store event logs for number of views, number of reads and number of users vs reading time on each blog post.
  - 3 new storage models will be added in blog/gae.py to store summary of number of views, number of reads and number of users vs reading time for each blog post.
  - Related domain objects and services will be accordingly changed.

- New handler (controller layer) will be added in `blog_dashboard.py` to populate data on the statistics tab.
- Regeneration jobs will be added to regenerate the related summary statistics models in case of data corruption from raw event log models.
- Frontend files for statistics tab will be added in `template/pages/blog-dashboard`



- For issue [#13397](#) :  
As writing custom jobs for validating each and every field of the model will be a complicated task, we can use the 'published\_on' data field of the models to know if the blog posts are private/drafts. We will need to update the 'published\_on' data field to 'none' every time the blog post is 'unpublished'. Currently, once a blog post is 'unpublished', we do not use 'published\_on' data field in any way. Thus updating it's value to 'None' will not break away any functionality. Therefore, if the value of 'published\_on' is **None** in 'BlogPostModel' and 'BlogPostSummaryModel', the blog post is private i.e it is a draft and non strict validation will be performed.

## Third-Party Libraries

N.A

## "Service" Dependencies

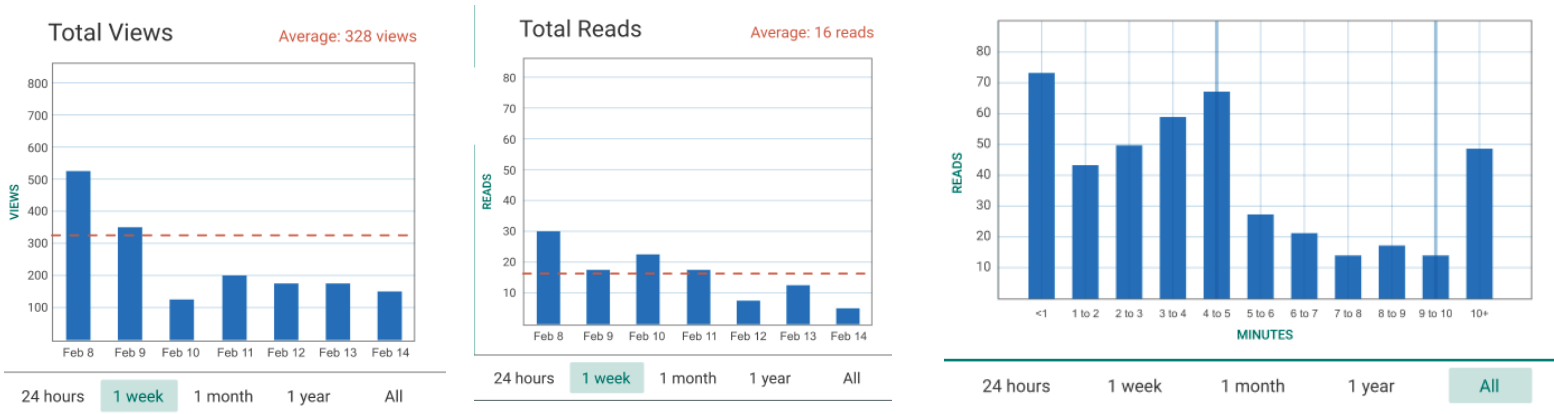
N.A

## Impact on Other Oppia Teams

It's a new interface and will not be affecting any existing team. The functionality of this interface will be mostly handled by the LaCE Team.

# Implementation Approach

For Statistics Tab Frontend Queries Requirement:



\*We will be storing all the date-time data in storage models in UTC Date-Time. We will convert Locale Time into UTC before storing data into models and convert UTC date time into Locale Time in frontend before generating graphs.

Frontend Queries requirements for generating graphs:

24 Hours: Number of views hour wise for the past 24 hours (inclusive of the views in the ongoing hour).

- ❖ Check for current local time
- ❖ Convert it to UTC date time
- ❖ Fetch hour wise views for past 24 hours (includes the views in the ongoing hours)

1 week: Number of views date wise for the past 6 days and the ongoing day.

- ❖ Check for current local date
- ❖ Convert it to UTC date
- ❖ Fetch views\_by\_date for past 6 days and the ongoing day

1 month: Number of views date wise for the past days for the ongoing month (inclusive of the ongoing date)

- ❖ Check for current local date
- ❖ Convert it to UTC date
- ❖ Fetch views\_by\_date for all the past days of the month (including the views of the ongoing day)

1 year: Number of views month wise for the ongoing year

- ❖ Check for current local date
- ❖ Convert it to UTC date
- ❖ Fetch views\_by\_month for the ongoing year (inclusive of the views of the ongoing day)

All: Number of views month wise for all the past year as well as ongoing year

- ❖ Check for current local date

- ❖ Convert it to UTC date
- ❖ Fetch views\_by\_month for all the years

\* Will be similar for number of reads and users vs reading time

We will be storing raw event logs for number of views, number of reads and number of users vs reading time for each blog post which will help us to regenerate summary models in case frontend requirements change or in case of data corruption in summary models due to race conditions.

## Storage Model Layer Changes

We will be having 2 types of models for storing statistics for the blog dashboard's statistics tab.

### 1) Event Raw log model:

- Will be used to generate summary models in case of data corruption in summary model
- Will be keyed to UTC date time
- There will be 3 types of raw log models based on event
  - i) **BlogPostViewedEventLogEntryModel**
  - ii) **BlogPostReadEventLogEntryModel**
  - iii) **BlogPostReadingTimeModel**

### 2) Summary models: Auto incrementing models that will be used to provide data for generating graphs in the frontend.

<u>Axes of Comparison</u>	<u>ALTERNATIVE 1</u>	<u>ALTERNATIVE 2</u>
	Having separate summary models according to the type of events i.e having separate models for number views, number of reads, average reading time	Having all summary data in one model i.e having all the fields of number of reads, number of views and average reading time in one model
Number of GET requests	Will need multiple GET request to load data	A single GET request will be sufficient to get all the data at once.
Lazy Loading	Will allow lazy loading on the statistics tab, i.e only the number of views data will be loaded initially and number of reads and reading time data will be loaded only if the user clicks on those tabs.  Also, we will not be making a new fetch every time when users will be switching between views, read and reading time charts for the same blog post. Once the data for them is loaded, it will not have to be loaded again until the user switches to a chart for different blog post.	Will not allow lazy loading of data
Faster Processing and	As the amount of data to be fetched in a	Due to large amount of data to be



Loading time	single request will be small, processing time to get data in the required format will be less and data will get loaded quickly in the frontend.	fetches in a single request, processing time to get data in the desired format will be more and the request will be slower which may also result in time out.
PUT request collisions	PUT request collisions for updating the same model will be less as the summary data for each event is in different model reducing the chances of data corruption	PUT request collisions for updating the same model will be more as the summary data for all the events is in same model increasing the chances of data corruption

### Therefore Alternate 1 is more preferable!

- There will be 6 summary models
  - i. **AuthorBlogPostViewsStatsModel**
  - ii. **AuthorBlogPostReadsStatsModel**
  - iii. **AuthorBlogPostReadingTimeModel**
  - iv. **BlogPostViewsAggregatedStatsModel**
  - v. **BlogPostReadsAggregatedStatsModel**
  - vi. **BlogPostReadingTimeModel**

#### 1. In `blog-statistics/gae.py` :

- Class **BlogPostViewedEventLogEntryModel** : It will register raw logs whenever a user will view a blog post. (It will be keyed to [timestamp]: [blog\_post\_id]: [random\_hash]; where timestamp is the UTC time in milliseconds of the occurrence of the event.)

Field Name	Property Type	To store	When event is fired
blog_post_id	String	blog_post_id	As soon as the user visits the blog post
author_id	string	user_id of the author	

This model will then be used to regenerate the number of views weekly, monthly, yearly and all views in the summary model - '**BlogPostViewsAggregatedStatsModel**' and '**AuthorBlogPostViewsAggregatedStatsModel**' using regeneration jobs written in **blog\_post\_statistics\_computation\_job.py** file if required.

For this data model following policies will be applied:

#### For export :

'get\_export\_policy' class method will be defined. Model does not contain user data and will not be exported.

**For deletion :**

'get\_deletion\_policy' class method will be defined. Model contains data to pseudonymize corresponding to a user: author\_id field.

**For model association to user :**

'get\_model\_association\_to\_user()' class method will be defined under which model will not be exported. (NOT\_CORRESPONDING\_TO\_USER')

To generate a unique id to key the model the following classmethod will be defined:

```

@classmethod
def get_new_event_entity_id(cls, blog_post_id: str) -> str:
    """Generates a unique id for the event model of the form
    '[timestamp]:[blog_post_id]:[hash]'.
    """
    timestamp = datetime.datetime.utcnow()

    # To avoid collision between events that occur at the same date time
    # for the same blog post, a random hash is appended to the event model id.
    for _ in range(base_models.MAX_RETRIES):
        random_hash = utils.convert_to_hash(
            str(utils.get_random_int(base_models.RAND_RANGE)),
            base_models.ID_LENGTH)
        model_id = cls.get_new_id('%s:%s:%s' % (
            utils.get_time_in_millisecs(timestamp),
            blog_post_id,
            random_hash))
        if not cls.get_by_id(blog_post_id):
            return model_id
    raise Exception(
        'New event model id generator is producing too many collisions.')

```

- Class **"BlogPostReadEventLogEntryModel"** : A new storage model will be introduced which will register raw logs whenever a user reads that blog post.(It will be keyed to [timestamp]: [blog\_post\_id]: [random\_hash]; where timestamp is the UTC time in milliseconds of the occurrence of the event.)

Field Name	Property Type	To store	When event is fired
blog_post_id	String Property	blog_post_id	If the user stays on the blog post longer than 50% of the time calculated using the number of words in the blog post ( 200 words per minute is a decent speed), the blog post will be marked as read and this event will be fired .
author_id	string	user_id of the author of the blog post	

This model will then be used to regenerate '**BlogPostReadsAggregatedStatsModel**' and '**AuthorBlogPostReadsStatsModel**' using regeneration jobs written in **blog\_post\_statistics\_computation\_job.py** file when required.

For this data model following policies will be applied:

**For export :**

'get\_export\_policy' class method will be defined. Model does not contain user data and will not be exported

**For deletion :**

'get\_deletion\_policy' class method will be defined. Model contains data to pseudonymize corresponding to a user: author\_id field.

**For model association to user :**

'get\_model\_association\_to\_user()' class method will be defined under which model will not be exported. (NOT\_CORRESPONDING\_TO\_USER')

A classmethod **get\_new\_event\_entity\_id(cls, blog\_post\_id)** will be defined to generate a unique id to key the model.

- Class "**BlogPostExitedLogEntryModel**": New storage model will be introduced which will register raw logs for the amount of time the user stayed on the blog post. (It will be keyed to [timestamp]: [blog\_post\_id]: [random\_hash]; where timestamp is the UTC time in milliseconds of the occurrence of the event.)

Field Name	Property Type	To store	When event is fired
time_user_stayed_on_blog_post	Float Property	Time users stayed on the blog post (in seconds).	As soon as the user leaves the blog post completely. (i.e either closes the tab or switches the page on the same tab.)  The event will also be fired if the user spends more than 45 minutes on the blog post or 5*estimated_reading_time (calculated using number of words on the blog post) whichever is greater.
blog_post_id	String Property	Id of the blog post	

author_id	String Property	user_id of the blog post	
-----------	-----------------	--------------------------	--

It will then be used to regenerate "BlogPostReadingTimeModel" if required using a regeneration job written in **blog\_post\_statistics\_computation\_job.py** file.

For this data model following policies will be applied:

**For export :**

'get\_export\_policy' class method will be defined. Model does not contain user data and will not be exported

**For deletion :**

'get\_deletion\_policy' class method will be defined. Model contains data to pseudonymize corresponding to a user: author\_id field.

**For model association to user :**

'get\_model\_association\_to\_user()' class method will be defined under which model will not be exported. (^NOT\_CORRESPONDING\_TO\_USER`)

A classmethod **get\_new\_event\_entity\_id(cls, blog\_post\_id)** will be defined to generate a unique id to key the model.

## 2. In blog/gae.py:

- Class **BlogPostViewsAggregatedStatsModel**:

Summary model to store number of views on a published blog post.

**[ Each instance of this model is keyed to the unique `blog_post_id` of the blog post. ]**

Field Name	Property Type	To Store	Required/Indexed/Repeated	Repacking of field
views_by_hour	Json Property	<p>A Dict. It will consist of a dict of dictionaries where key is the date (YYYY-MM-DD) in the UTC format) and value is dict with keys as hours (in the UTC format) and number of views as values.</p> <p>Dict Format :</p> <pre>{(date_1): {   "00": num_of_views,   "01": num_of_views,   ....   "23": num_of_views }}</pre> <p>E.g.</p> <pre>{   "2022-08-02": {     "00": 40,     "01": 50,     ....     "23": 60   }   "2022-08-03": {     "00": 45,     "01": 55,     ....     "23": 85,   } },</pre> <p>Date is in YYYY-MM-DD Hour: HH</p>	<p>Required = True Indexed = False Repeated = False</p>	<p>At max there will be hourly views keyed to 3 dates.</p> <p>We don't need views by hour for more than the past 24 hours ( including the current ongoing hour). To be on safer side, we will maintain hourly views for past 3 days ( including the ongoing day) and delete the rest whenever a PUT request is performed on the storage model</p>

views_by_date	Json Property	<p>A dict It will consist of key-value pairs where key is the month(YYYY-MM) and value is dict with keys as UTC date and values as number of views on the blog posts on that date.</p> <p>Dict format: (month_1): { date_1: num_of_views date_2 : num_of_views }</p> <p>E.g.</p> <pre>{   "2022-08": {     "01": 40,     "02": 50,     ....     "31": 60   }   "2022-09": {     "01": 45,     "02": 55,     ....     "30": 850,   } },</pre> <p>Month format: YYYY-MM Date format- DD</p>	<p>Required = True Indexed = False Repeated = False</p>	<p>At max there will be views by date keyed to 3 months. (Ongoing month and the past 2 months(all the days of the month)))</p> <p>We want past 30 days data (including the ongoing day) from the present date to generate monthly views in the chart.</p>
views_by_month	Json Property	<p>A Dict</p> <p>It will consist of a dict of dictionaries where key is the year ( in the UTC format) and value is dict with keys as month number (in the UTC format) and number of views in that month as value.</p> <p>Dict Format :</p> <pre>year: {   month: num_of_views }</pre>	<p>Required = True Indexed = False Repeated = False</p>	<p>Monthly views for all the years have to be stored to be displayed in 'All' option of the chart. Repacking of data is not required.</p>

		E.g. <pre>{   "2022": {     "01": 400,     "02": 500,     ....     "12": 600   }   "2023": {     "01": 450,     "02": 550,     ....     "12": 850,   } },</pre>		
--	--	--	--	--

**Repacking of data fields:** Whenever a user views the blog post, a PUT request will be performed on the model to update the number of views in the model. A classmethod 'RepackBlogPostViewsAggregatedStats()' will be called which will remove all the data in the 'views\_by\_hour' data field except that of the past 2 days ( along with the ongoing day data) and in 'views\_by\_date' data field except that of the past 1 month (along with the ongoing month). We only need to show views by hours for the past 24 hours (including the ongoing hour). Therefore maintaining hour wise views for all the dates is not required and will just increase the size of the data model. To be on a safer side, we maintain past 2 days data along with the ongoing date data. For views\_by\_date, for generating 'monthly views' in chart, we will need past 30 days data (including the ongoing month), therefore we will maintain the ongoing month data and the past 1 month data to ensure having 30 days data at all times.

**For export :**

'get\_export\_policy' class method will be defined. The model does not contain data corresponding to a user and it will not be exported.

**For deletion :**

It will be NOT\_APPLICABLE: The model is not related to user data at all

**For model association to user :**

Model does not have any association to user, thus, inside 'get\_model\_association\_to\_user()' - model will have value - 'NOT\_CORRESPONDING\_TO\_USER'.

- Class **BlogPostReadsAggregatedStatsModel:**  
Summary model to store number of reads on a published blog post.  
**[ Each instance of this model is keyed to the unique blog\_post\_id of the blog post. ]**

**It will be formatted similar to the views model but will just store the number of reads!**

- **Class BlogPostReadingTimeModel:**

New storage model which will store the total number of users staying for a particular time on the blog post. It will be updated as soon as the user leaves the blog post completely ( eg. Closes the tab, goes back to other pages)

**[ Each instance of this model is keyed to the unique blog\_post\_id of the blog post. ]**

Field Name	Property Type	To store	When event is fired
zero_to_one_min	Integer Property	Number of user taking less than a minute to read the blog post	As soon as the user leaves the blog post completely. (i.e either closes the tab or switches the page on the same tab.)  The event will also be fired if the user spends more than 45 minutes on the blog post or 5*estimated_reading_time (calculated using number of words on the blog post) which ever is greater.
one_to_two_min	Integer Property	Number of users taking one to two minutes to read the blog post.	
two_to_three_min	Integer Property	Number of users taking two to three minutes to read the blog post.	
three_to_four_min	Integer Property	.....	
.....	.....	.....	
more_than_ten_min	Integer Property	Number of users taking more than ten minutes to read the blog post.	

**For export :**

'get\_export\_policy' class method will be defined. The model does not contain data corresponding to a user and it will not be exported.

**For deletion :**

It will be NOT\_APPLICABLE: The model is not related to user data at all

**For model association to user :**

Model does not have any association to user, thus, inside 'get\_model\_association\_to\_user()' - model will have value - 'NOT\_CORRESPONDING\_TO\_USER'.

- **In Class BlogPostRightsModel -**

Class method - get\_published\_blog\_posts\_count() will be added. It will query based on the 'blog\_post\_is\_published' data field and then return the count.



### 3. In user/gae.py:

- Class **AuthorBlogPostViewsStatsModel:**

Summary model to store number of views on all published blog posts by the user.  
**[ Each instance of this model is keyed to the user\_id ].**

**It will be formatted similar to BlogPostViewsAggregatedStatsModel.**

This model will hence record total views on **all** the published blog posts by the user.

**For export :**

'get\_export\_policy' class method will be defined. The model contains data corresponding to a user and it will be exported.

**For deletion :**

'get\_deletion\_policy' class method will be defined. The model contains data corresponding to a user - and it will be deleted.

**For model association to user :**

'get\_model\_association\_to\_user()' class method will be defined under which model will be exported as one instance per user.

- Class **AuthorBlogPostReadsStatsModel:**

Summary model to store number of reads on all published blog posts by the user.  
**[ Each instance of this model is keyed to the user\_id ].**

**It will be formatted similar to BlogPostReadsAggregatedStatsModel.**

This model will hence record total reads on **all** the published blog posts by the user.

**For export :**

'get\_export\_policy' class method will be defined. The model contains data corresponding to a user and it will be exported.

**For deletion :**

'get\_deletion\_policy' class method will be defined. The model contains data corresponding to a user - and it will be deleted.

**For model association to user :**

'get\_model\_association\_to\_user()' class method will be defined under which model will be exported as one instance per user.

- Class **AuthorBlogPostReadingTimeModel**:  
Summary model to store the total number of users staying for a particular time on all the blog posts written by the user.

[ Each instance of this model is keyed to the user\_id ].

It will be formatted similar to **BlogPostReadingTimeModel**.

**For export :**

'get\_export\_policy' class method will be defined. The model contains data corresponding to a user and it will be exported.

**For deletion :**

'get\_deletion\_policy' class method will be defined. The model contains data corresponding to a user - and it will be deleted.

**For model association to user :**

'get\_model\_association\_to\_user()' class method will be defined under which model will be exported as one instance per user.

- In Class **UserSettingsModel** following field will be added:

Field Name	Property Type	To Store	Required/Indexed/ Repeated
publicly_viewable_name	String Type	A string. A name which will be shown as author name on blog posts and on blog author page.	Required and Indexed

Domain Layer:

**1. In blog\_domain.py:**

- class **BlogPostViewsAggregatedStats** : It would be added to handle all functions directly related to the BlogPostViewsAggregatedStatsModel.  
Some functions (class methods) inside it will be :

- i. `__init__()` : Initializes a `BlogPostViewsAggregatedStats` model domain object.
  - ii. `to_frontend_dict()`: It will return a 'dict' representation of the stats object to be used in frontend.
  - iii. `create_new_blog_post_views_stats()` : function would just call the constructor to initialize.
- class **BlogPostReadsAggregatedStats** : It would be added to handle all functions directly related to the `BlogPostReadsAggregatedStatsModel`.
  - Some functions (class methods) inside it will be :
    - i. `__init__()` : Initializes a `BlogPostReadsAggregatedStats` model domain object.
    - ii. `to_frontend_dict()`: It will return a 'dict' representation of the stats object to be used in frontend.
    - iii. `create_new_blog_post_reads_stats()` : function would just call the constructor to initialize.
- class **BlogPostReadingTime**: It would be added to handle all functions directly related to the `BlogPostReadingTimeModel`.
  - Some functions (class methods) inside it will be :
    - i. `__init__()` : Initializes a `BlogPostReadingTime` model domain object.
    - ii. `to_frontend_dict()`: It will return a 'dict' representation of the stats object.
    - iii. `create_new_blog_post_reading_time()` : function would just call the constructor to initialize.

## 2. In `blog_statistics_domain.py`:

- class **BlogPostViewedEventLogEntry** : It would be added to handle all functions directly related to the `BlogPostViewedEventLogEntryModel` .
  - Some functions (class methods) inside it will be :
    - 1. `__init__()` : Initializes a `BlogPostViewedEventLogEntry` model domain object.
    - 2. `create_new_blog_post_view_entry()` : function would just call the constructor to initialize.
- class **BlogPostReadEventLogEntry** : It would be added to handle all functions directly related to the `BlogPostReadEventLogEntryModel` .
  - Some functions (class methods) inside it will be :
    - 1. `__init__()` : Initializes a `BlogPostReadEventLogEntry` domain object.

2. `create_new_blog_post_read_entry()` : function would just call the constructor to initialize.
- class **BlogPostExitedLogEntry** : It would be added to handle all functions directly related to the `BlogPostExitedLogEntryModel` .  
Some functions (class methods) inside it will be :
    1. `__init__()` : Initializes a `BlogPostExitedLogEntry` model domain object.
    2. `create_new_blog_post_exited_entry()` : function would just call the constructor to initialize.

### 3. In `user_domain.py`:

- class **AuthorBlogPostViewsStats** : It would be added to handle all functions directly related to the `AuthorBlogPostViewsStatsModel`.  
Some functions (class methods) inside it will be :
  - i. `__init__()` : Initializes a `AuthorBlogPostViewsStats` model domain object.
  - ii. `to_frontend_dict()`: It will return a 'dict' representation of the stats object to be used in frontend.
  - iii. `create_new_views_stats()` : function would just call the constructor to initialize.
  
- class **AuthorBlogPostReadsStats** : It would be added to handle all functions directly related to the `AuthorBlogPostReadsStatsModel`.  
Some functions (class methods) inside it will be :
  - i. `__init__()` : Initializes a `AuthorBlogPostReadsStats` model domain object.
  - ii. `to_frontend_dict()`: It will return a 'dict' representation of the stats object to be used in frontend.
  - iii. `create_new_reads_stats()` : function would just call the constructor to initialize.
  
- class **AuthorBlogPostReadingTime**: It would be added to handle all functions directly related to the `AuthorBlogPostReadingTimeModel`.  
Some functions (class methods) inside it will be :
  - i. `__init__()` : Initializes a `AuthorBlogPostReadingTime` model domain object.
  - ii. `to_frontend_dict()`: It will return a 'dict' representation of the stats object.
  - iii. `create_new_reading_time_stats()` : function would just call the constructor to initialize.

#### 4. In jobs/transforms/validation/blog\_validation.py:

- To address issue [#13397](#),  
Instead of calling 'blogPostRightsModel' to know the status of the blog posts (published/drafts), we will check if the model's time of being published is not None as writing custom jobs for validating each and every field will be complicated.

```
def _get_domain_object_validation_type(self, unused_item):
    """Returns the type of domain object validation to be performed.

    Args:
        unused_item: datastore_services.Model. Entity to validate.

    Returns:
        str. The type of validation mode: strict or non strict.
    """
    if (model.published_on):
        return base_validation.ValidationModes.STRICT

    return base_validation.ValidationModes.NON_STRICT
```

- Validation to verify relationship of various Blog Post Statistics Models with BlogPostModel, BlogPostSummaryModel, BlogPostRightsModel will be added.

#### 5. In blog\_post\_search\_services.py:

Blog\_post\_search\_services file will be added to allow users to search through numerous blog posts, and the following functionalities will be added in it:

- **index\_blog\_post\_summaries():** To add blog posts to search index.

```
def index_blog_post_summaries(blog_post_summaries):
    """Adds the blog post summaries to the search index.

    Args:
        blog_post_summaries: list(BlogPostSummaryModel). List of Blog Post
        Summary domain objects to be indexed.
    """
    platform_search_services.add_documents_to_index([
        _blog_post_summary_to_search_dict(blog_post_summary)
        for blog_post_summary in blog_post_summaries
        if _should_index_blog_post(blog_post_summary)
    ], SEARCH_INDEX_BLOG_POSTS)
```

- **\_should\_blog\_post\_be\_indexed():** Checks whether the blog post should be indexed so as to apply search queries on it. It will check to see if the blog post has been published. It will take `blog_post_id` as an argument.
- **\_to\_search\_dict():** Converts blog post summary domain objects into a format so that search can be performed on them.

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

    Args:
        blog_post_summary: BlogPostSummaryModel. BlogPostSummary domain object.

    Returns:
        dict. The representation of the given blog post summary, in a form that can
        be used by the search index.
    """
    doc = {
        'id': blog_post_summary.id,
        'title': blog_post_summary.title,
        'tags': blog_post_summary.tags,
        'author': blog_post_summary.author_username,
    }
    return doc
```

- **search\_blog\_posts():**

```
def search_blog_posts(query, tags, size, offset=None):
    """Searches through the available published blog posts.

    Args:
        query: str or None. The query string to search for.
        tags: list(str). The list of tags to query for. If it is
            empty, no tag filter is applied to the results. If it is not
            empty, then a result is considered valid if it matches at least one
            of these tags.
        size: int. The maximum number of results to return.
        offset: int or None. A marker that is used to get the next page of
            results. If there are more documents that match the query than
            'size', this function will return an offset to get the next page.

    Returns:
        tuple. A 2-tuple consisting of:
        - list(str). A list of blog post ids that match the query.
        - int or None. An offset if there are more matching blog posts to
            fetch, None otherwise. If an offset is returned, it will be a
            web-safe string that can be used in URLs.
    """
    return platform_search_services.search(
        query, SEARCH_INDEX_BLOG_POSTS, tags, offset=offset, size=size, ids_only=True)
```

- `clear_blog_post_search_index()` : To clear blog post search index.

## 6. In `blog_post_search_services_test.py`:

It will have test functions for `blog_post_search_services.py` file. Test function will be :

1. `test_search_blog_posts()`: This function will contain a function - `mock_search()` which will mimic the behavior of `gae_search_engine`. A query will be provided to see that `search_blog_post` function works properly.
2. `test_clear_blog_post_search_index()`: It will check that all the blog post summaries indexed get cleared on calling `clear_blog_post_search_index()`.
3. `test_blog_post_summaries_are_added_to_search_index()` : To check indexing of blog posts is done properly.

## 6. In `blog_post_search_indexing_jobs.py`:

Jobs which will index blog posts to Elastic Search will be added.

- Class `IndexBlogPostsInSearchJob` will add all the published blog posts to elastic search.

```
class IndexBlogPostsInSearchJob(base_jobs.JobBase):
    """Job that indexes the blog posts in Elastic Search."""

    MAX_BATCH_SIZE = 1000

    def run(self) -> beam.PCollection[job_run_result.JobRunResult]:
        """Returns a PCollection of 'SUCCESS' or 'FAILURE' results from
        the Elastic Search.

        Returns:
            PCollection. A PCollection of 'SUCCESS' or 'FAILURE' results from
            the Elastic Search.
        """
        return (
            self.pipeline
            | 'Get all non-deleted models' >> (
                ndb_io.GetModels(
                    blog_post_models.BlogPostSummaryModel.get_all(include_deleted=False))
            | 'Split models into batches' >> beam.transforms.util.BatchElements(
                max_batch_size=self.MAX_BATCH_SIZE)
            | 'Index batches of models' >> beam.ParDo(
                IndexBlogPostSummaries())
            | 'Count the output' >> (
                job_result_transforms.ResultsToJobRunResults())
        )
```

- Class IndexBlogPostSummaries - DoFn to index blog post summaries.

```
class IndexBlogPostSummaries(beam.DoFn): # type: ignore[misc]
    """DoFn to index blog post summaries."""

    def process(
        self, blog_post_summary_models: List[datastore_services.Model]
    ) -> Iterable[result.Result[None, Exception]]:
        """Index blog post summaries and catch any errors.

        Args:
            blog_post_summary_models: list(Model). Models to index.

        Yields:
            JobRunResult. List containing one element, which is either SUCCESS,
            or FAILURE.
        """
        try:
            search_services.index_blog_post_summaries( # type: ignore[no-untyped-call]
                cast(List[blog_post_models.BlogPostSummaryModel], blog_post_summary_models))
            for _ in blog_post_summary_models:
                yield result.Ok()
        except platform_search_services.SearchException as e:
            yield result.Err(e)
```

## 7. In blog\_post\_search\_indexing\_jobs\_test.py:

Functions to ensure that indexing of blog post summary models is done properly are added in this file.

- test\_indexes\_not\_deleted\_blog\_post\_models: To check that blog post summary models which are not deleted are indexed.
- test\_reports\_failed\_when\_indexing\_fails: To check that the reports return failure status if indexing of blog post summary models fail.
- test\_skips\_deleted\_blog\_post\_model : To check that while indexing models, it does not add deleted model to the list.
- test\_skip\_draft\_blog\_post\_model : To ensure that the indexing jobs do not add draft blog post models to the elastic search.

## 8. In blog\_services.py :

- For issue [#13397](#): We do not use 'published\_on' date once any blog post is unpublished. Therefore we can make its value 'none' when a blog post is unpublished.



- Function 'unpublish\_blog\_post(blog\_post\_id)' will be modified to update BlogPostModel's and BlogPostSummaryModel's - 'published\_on' field to None
- get\_blog\_post\_views\_aggregated\_stats(blog\_post\_id): This function will retrieve the blogPostViewsAggregatedStats object. It will fetch the model from the storage layer, will call the 'repack\_data\_fields()' function and generate domain object from the storage model.
- get\_blog\_post\_reads\_aggregated\_stats(blog\_post\_id): This function will retrieve the blogPostReadsAggregatedStats object. It will fetch the model from the storage layer, will call the 'repack\_data\_fields()' function and generate domain object from the storage model.
- register\_blog\_post\_view( blog\_post\_id ) : This function will increment the number of views on a blog post by increasing the count of views in repacked blog\_post\_views\_aggregated\_stats. It will call the function 'get\_blog\_post\_views\_aggregated\_stats(blog\_post\_id)' to get a repacked blog\_post\_views\_aggregated\_stats domain object and will increment the number of views according to the data fields. We will use UTC datetime to store/ increment values.

```
date_time = datetime.datetime.utcnow()
year = date_time.year
month = date_time.month
day = date_time.day
hour = date_time.hour
```

- register\_blog\_post\_read(blog\_post\_id) : This function will increment the number of reads on a blog post by increasing the count of reads in repacked blog\_post\_reads\_aggregated\_stats. It will call the function 'get\_blog\_post\_reads\_aggregated\_stats(blog\_post\_id)' to get a repacked blog\_post\_reads\_aggregated\_stats domain object and will increment the number of reads according to the data fields. We will use UTC datetime to store/ increment values.
- register\_blog\_post\_exited(blog\_post\_id) : This function will increment the number of users in the respective time bucket according to the time spent on the blog post in BlogPostReadingTime model.
- get\_blog\_post\_stats(blog\_post\_id, chart\_type) : This function will return blog post stats domain object to be used in the blog dashboard statistics tab. It will take in blog\_post\_id and chart\_type (views, reads, reading time) as an argument.

- `get_total_number_of_published_blog_post()`: This function will return the total count of published blog posts. It will call the `blog_model.BlogPostRights.published_blog_post_count()` class method.
- `delete_blog_post(blog_post_id: str)`: This function will be modified to delete raw event log models as well as summary statistic models corresponding to the `blog_post_id`.
- `get_blog_post_ids_matching_query()`: A list of blog post ids matching given query or list of tags is returned by the function. This function will be called by the search handler in `blog_homepage.py`.

```
def get_blog_post_ids_matching_query(query_string, tags, offset=None):
    """Returns a list with all blog post 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 if there are at least that many, otherwise
    it returns all remaining results. (If this behaviour does not occur, an error will
    be logged.) The method also returns a search offset.

    Args:
        query_string: str. A search query string.
        tags: list(str). The list of tags to query for. If it is empty, no tags filter
            is applied to the results. If it is not empty, then a result is considered
            valid if it matches at least one of these tags.
        offset: int or None. Optional offset from which to start the search query. If no
            offset is supplied, the first N results matching the query are returned.

    Returns:
        2-tuple of (returned_blog_post_ids, search_offset). Where:
            returned_blog_post_ids : list(str). A list with all blog post ids matching
            the given search query string, as well as a search offset for future fetches.
            search_offset: int. Search offset for future fetches.
    """
    returned_blog_post_ids = []
    search_offset = offset
    for _ in range(MAX_ITERATIONS):
        remaining_to_fetch = feconf.SEARCH_RESULTS_PAGE_SIZE - len(returned_blog_post_ids)
        blog_post_ids, search_offset = blog_post_search_services.search_blog_posts(
            query_string, tags, remaining_to_fetch, offset=search_offset)

        invalid_blog_post_ids = []
        for ind, model in enumerate(
            blog_models.BlogPostSummaryModel.get_multi(blog_post_ids)):
            if model is not None:
                returned_blog_post_ids.append(blog_post_ids[ind])
            else:
                invalid_blog_post_ids.append(blog_post_ids[ind])

        if (len(returned_blog_post_ids) == feconf.SEARCH_RESULTS_PAGE_SIZE or search_offset is None):
            break
        logging.error('Search index contains stale blog post ids: %s' % ', '.join(invalid_blog_post_ids))
    if (len(returned_blog_post_ids) < feconf.SEARCH_RESULTS_PAGE_SIZE
        and search_offset is not None):
        logging.error('Could not fulfill search request for query string %s; at least '
            '%s retries were needed.' % (query_string, MAX_ITERATIONS))

    return [(returned_blog_post_ids, search_offset)]
```

## 9. In `user_services.py` :

- `get_author_blog_post_views_stats(user_id)`: This function will retrieve the `authorBlogPostViewsStats` object. It will fetch the model from the storage layer, will call the `'repack_data_fields()'` function and generate domain object from the storage model.
- `get_author_blog_post_reads_stats(user_id)`: This function will retrieve the `authorBlogPostReadsStats` object. It will fetch the model from the storage layer, will call the `'repack_data_fields()'` function and generate domain object from the storage model.
- `register_blog_post_view (user_id)` : This function will increment the number of views by increasing the count of views in repacked `author_blog_post_views_stats`. It will call the function `'get_author_blog_post_views_stats(user_id)'` to get a repacked `author_blog_post_views_stats` domain object and will increment the number of views according to the data fields. We will use UTC datetime to store/ increment values.
- `register_blog_post_read(user_id)` : This function will increment the number of reads by increasing the count of reads in `author_blog_post_reads_stats`. It will call the function `'get_author_blog_post_reads_stats(user_id)'` to get a repacked `author_blog_post_reads_stats` domain object and will increment the number of reads according to the data fields. We will use UTC datetime to store/ increment values.
- `register_blog_post_exited(user_id)` : This function will increment the number of users in the respective time bucket according to the time spent on the blog post in `AuthorBlogPostReadingTime` model.
- `get_author_blog_posts_stats(author_id, chart_type)`: This function will return user blog posts stats domain object to be used in the blog dashboard statistics tab to display statistics of all blog posts by an author according to the chart type( views, reads, reading\_time). It will take in `author_id` and `chart_type` as an argument.

## User Flows (Controllers and Services)

### 1. Inside `blog_homepage.py` :

- **In class `BlogPostStatisticsDataHandler`**: 'put' type request handler function will be added. It will update both summary models and raw event log model according to the type of event and `blog_post_id`. It will take `'blog_post_id'` and `'event_type'` as an argument.

- In class **BlogHomepageDataHandler** : 'get' type request will be edited to include 'total\_number\_of\_published\_blog\_posts'.
- **class BlogPostSearchHandler**: This handler will have a get() function that returns a list of blog post summary dicts that satisfy the user's query. It will call get\_matching\_blog\_post\_dicts(). It will format the query string into a form that can be used for searching, removing all punctuation and replacing it with spaces. Any other required formatting will also take place here .It will supply all the parameters provided in proper order to get\_matching\_blog\_post\_dicts() function. Snap of part of handler :

```

@acl_decorators.open_access
def get(self):
    """Handles GET requests."""
    query_string = utils.unescape_encoded_uri_component(
        self.normalized_request.get('q'))
    # Remove all punctuation from the query string, and replace it with
    # spaces. See http://stackoverflow.com/a/266162 and
    # http://stackoverflow.com/a/11693937
    remove_punctuation_map = dict(
        (ord(char), None) for char in string.punctuation)
    query_string = query_string.translate(remove_punctuation_map)

    # If there is a tag parameter, it should be in the following form:
    # tag=("GSoC" OR "Oppia")
    tags_string = self.normalized_request.get('tags')

    # The 2 and -2 account for the '(' and ')' characters at the
    # beginning and end.
    tags = [
        You, 1 minute ago • Uncommitted changes
        tags_string[2:-2].split(' OR ') if tags_string else []
    ]

    search_offset = self.normalized_request.get('offset')

```

- **Function `get_matching_blog_post_dicts()`:** Given the details of a query i.e query string and tags, and a search offset, it will return a list of blog post summary dicts that satisfy the query.

```
def get_matching_blog_post_dicts(query_string, tags, search_offset):
    """Given the details of a query and a search offset, returns a list of
    blog post 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. If it is not
            empty, then a result is considered valid if it matches at least one
            of these tags.
        search_offset: int or None. Offset indicating where, in the list of
            blog post search results, to start the search from. If None,
            blog post search results are returned from beginning.

    Returns:
        tuple. A tuple consisting of two elements:
        - list(dict). Each element in this list is a blog post dict,
            representing a search result.
        - int. The blog post summary index offset from which to start the
            next search.
    """
    blog_post_ids, new_search_offset = (
        blog_services.get_blog_post_ids_matching_query(
            query_string, tags, offset=search_offset))
    blog_post_list = (
        blog_services.get_blog_post_summary_dicts_by_ids(
            blog_post_ids))

    if len(blog_post_list) == feconf.DEFAULT_QUERY_LIMIT:
        logging.exception(
            '%s blog posts were fetched to load the page. '
            'You may be running up against the default query limits.'
            % feconf.DEFAULT_QUERY_LIMIT)
    return blog_post_list, new_search_offset
```

## 2. Inside `blog_homepage_test.py` :

New test functions to test 'BlogPostSearchHandler' will be introduced.

- `test_search_blog_post_from_query()`: Multiple query strings will be created and checked for returning correct blog posts from a bunch of created blog posts. In all cases no value for 'tags' will be provided.
- `test_filter_blog_post_from_tags()`: Multiple tags will be used to call the handler and check if blog post associated to the tags are only returned. No query string will be provided in this case.
- `test_search_and_tag_filter()`: Both query string and tags will be used to hit the handler and check if the correct results are returned.
- `test_blog_post_stats_data_update()`: It will check that statistics in raw event log models and summary models get incremented correctly.

## 3. Inside `blog_dashboard.py` :

- **class BlogDashboardStatisticsHandler**: This handler will be responsible for loading data on statistics tab of the blog dashboard. It will respond to 'GET' type requests. It will return data from summary model based on chart\_type and blog\_post\_id / author\_id.

## 4. Inside `blog_dashboard_test.py`:

Test functions to test 'BlogDashBoardStatisticsHandler' will be added. We will check if correct data is provided according to chart type and author\_username/ blog\_post\_id.

For Regenerating Statistics:

## 1. In `jobs/batch_jobs/blog_post_statistics_computation_job.py` file -

- `regenerateViewsStatistics()`: This function will take in **BlogPostViewedEventLogEntry** Models to regenerate **BlogPostViewsAggregatedStatsModel** and **AuthorBlogPostViewsStatsModel**.

- regenerateReadsStatistics(): This function will take in **BlogPostReadEventLogEntry** Models to regenerate **BlogPostReadAggregatedStatsModel** and **AuthorBlogPostReadStatsModel**.
- regenerateReadingTimeStatistics(): This function will take in **BlogPostExitedEventLogEntry** Models to regenerate **BlogPostReadingTimeStatsModel** and **AuthorBlogPostReadingTimeStatsModel**.
- **Class RegenerateBlogPostViewsStats** : DoFn to regenerate all blog post views stats. It will call regenerateViewsStatistics() function.
- **Class RegenerateBlogPostReadStats** : DoFn to regenerate all blog post reads stats. It will call regenerateReadStatistics() function.
- **Class RegenerateBlogPostReadingTimeStats** : DoFn to regenerate blog post reading time stats. It will call regenerateReadingTimeStatistics() function.

## Web frontend changes

### 1. FOR BLOG CARD:

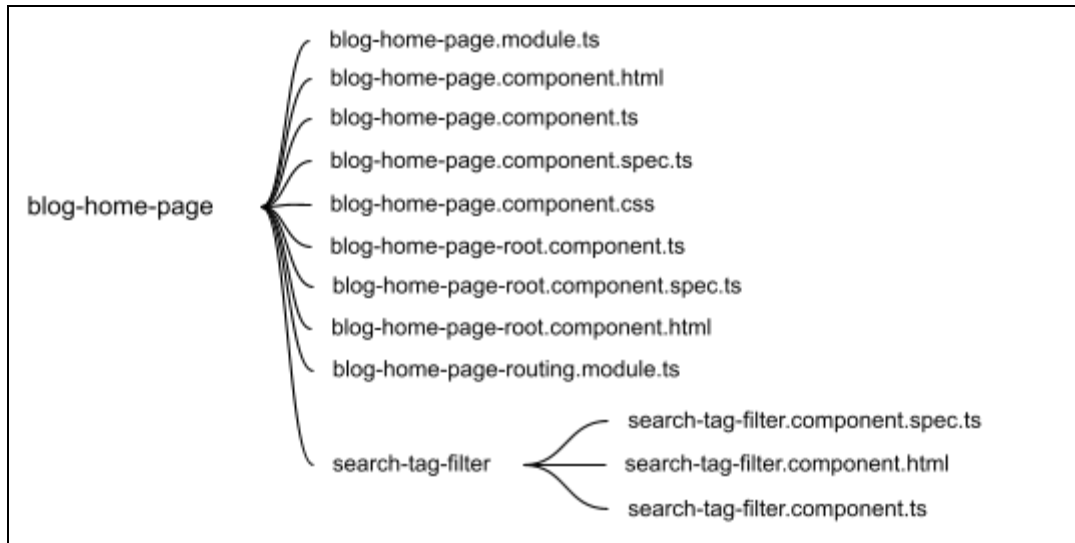
- The 'blog-card' component will be moved from 'templates/pages/blog-dashboard' to 'templates/components' and will be added to 'shared-component.module.ts' .
- Inside blog-card-component.html -
  - '\*ngFor' will be used to display - tag buttons in the bottom of the blog card. Clicking on any of the tag buttons will call 'filterBy(tag)' function in blog-card.component.ts which will lead to [search/filter by](#) page where blog posts associated with tag will be shown.
- Inside blog-card-component.ts -
  - Function - 'filterBy(tag)' will be added. It will call the 'executeSearchQuery(query, tags)' function in blog-post-search-service.ts. URL of the page will be updated with query accordingly.
- Inside blog-card-component.spec.ts -
  - Frontend test for the function filterBy(tag) will be added. Different tags will be supplied to the function to check that the URL of the webpage gets updated and executeSearchQuery(query, tags) get called in blog-post-search-service.ts .

Inside templates/pages 3 new folder will be created:

- blog-home-page
- blog-authors-page
- blog-post-page

## 2. FOR BLOG HOME PAGE:

Inside blog-home-page -



In **blog-home-page.component.html** :

1. We will use 'ng-if' to display search page heading. If `is_showing_search_result()` is true, 'I18N\_BLOG\_HOMEPAGE\_SEARCH\_RESULTS\_HEADING' will be shown.
2. Each page will show only 10 published blog posts at max. [ Feconf constant - 'MAX\_NUM\_CARDS\_TO\_DISPLAY\_ON\_BLOG\_HOMEPAGE' is already added in data base ]. '\*ngFor' will be used to display blog-cards using the list of `blogPostSummaries` loaded. Clicking on any of the blog card will call `loadBlogPostPage()` function.



- At a single time blog post summaries for 10 blog post will only be loaded from backend. When user user pagination to go to next page, only then next 10 blog posts summaries will be loaded.
- To allow blog-home-page to be accessible in all languages following I18N Keys will be introduced and will be used in the HTML template:

I18N KEY	VALUE
I18N_BLOG_HOMEPAGE_LATEST_POST_HEADING	Latest post
I18N_BLOG_HOMEPAGE_WELCOME_TEXT	Welcome to the oppia blog!
I18N_BLOG_HOMEPAGE_OPPIA_MOTTO	Building a community to provide quality education for those who lack access to it.
I18N_BLOG_HOMEPAGE_FILTERBY_HEADING	Filter by
I18N_BLOG_HOMEPAGE_TAGS_HEADING	Tags
I18N_BLOG_HOMEPAGE_SEARCH_FIELD_TEXT	Search
I18N_BLOG_HOMEPAGE_TOTAL_BLOG_POST_HEADING	Displaying
I18N_BLOG_HOMEPAGE_CLEAR_BUTTON_TEXT	Clear
I18N_BLOG_HOMEPAGE_SEARCH_RESULTS_HEADING	Search results for

#### In **blog-home-page.component.ts** :

`publishedBlogPosts` : It will be a list containing blog post summary dictionaries as its items. The values will be provided by the `blog-homepage.service.ts`

`totalBlogPostCount` : Total number of published blog posts.

`loadBlogPostPage()`: It will load the blog post page for the blog with the given blog post id. It will take `blogPostId` as a parameter

#### Inside **blog-homepage-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 put and get requests will be placed in order to retrieve data from the datastore and update stats in the datastore.

The class name will be "BlogHomePageBackendApiService".

It will import existing class 'BlogPostSummary' from 'templates/domain/blog/blog-post-summary.model.ts', class 'BlogPostData' from 'core/templates/domain/blog/blog-post.model.ts' The domain objects related to these classes have the following structure :

BlogPostData	<p>It will contain data from BlogPostBackendDict. It will have the following fields:</p> <table border="1" data-bbox="516 485 1495 1220"> <tr> <td>id</td> <td>Id of the blog post</td> <td>string</td> </tr> <tr> <td>authorName</td> <td>User name of the author</td> <td>string</td> </tr> <tr> <td>title</td> <td>Title of the blog post</td> <td>string</td> </tr> <tr> <td>content</td> <td>Content of the blog post</td> <td>string</td> </tr> <tr> <td>tags</td> <td>List of tags associated with the blog post</td> <td>string[ ]</td> </tr> <tr> <td>publishedOn</td> <td>Date-Time on which blog post was published</td> <td>string</td> </tr> <tr> <td>thumbnailFilename</td> <td>The name of the thumbnail</td> <td>string</td> </tr> <tr> <td>lastUpdated</td> <td>The date-time on which the blog post was last updated.</td> <td>string</td> </tr> <tr> <td>urlFragment</td> <td>The url fragment associated with the blog post</td> <td>string</td> </tr> </table>	id	Id of the blog post	string	authorName	User name of the author	string	title	Title of the blog post	string	content	Content of the blog post	string	tags	List of tags associated with the blog post	string[ ]	publishedOn	Date-Time on which blog post was published	string	thumbnailFilename	The name of the thumbnail	string	lastUpdated	The date-time on which the blog post was last updated.	string	urlFragment	The url fragment associated with the blog post	string
id	Id of the blog post	string																										
authorName	User name of the author	string																										
title	Title of the blog post	string																										
content	Content of the blog post	string																										
tags	List of tags associated with the blog post	string[ ]																										
publishedOn	Date-Time on which blog post was published	string																										
thumbnailFilename	The name of the thumbnail	string																										
lastUpdated	The date-time on which the blog post was last updated.	string																										
urlFragment	The url fragment associated with the blog post	string																										
BlogPostSummary	<p>It will contain data from BlogPostSummaryBackendDict. It will have the following fields:</p> <table border="1" data-bbox="516 1392 1495 1858"> <tr> <td>id</td> <td>Id of the blog post</td> <td>string</td> </tr> <tr> <td>authorName</td> <td>User name of the author</td> <td>string</td> </tr> <tr> <td>title</td> <td>Title of the blog post</td> <td>string</td> </tr> <tr> <td>tags</td> <td>List of tags associated with the blog post</td> <td>string[ ]</td> </tr> <tr> <td>publishedOn</td> <td>Date-Time on which blog post was published</td> <td>string</td> </tr> <tr> <td>thumbnailFilename</td> <td>The name of the thumbnail</td> <td>string</td> </tr> </table>	id	Id of the blog post	string	authorName	User name of the author	string	title	Title of the blog post	string	tags	List of tags associated with the blog post	string[ ]	publishedOn	Date-Time on which blog post was published	string	thumbnailFilename	The name of the thumbnail	string									
id	Id of the blog post	string																										
authorName	User name of the author	string																										
title	Title of the blog post	string																										
tags	List of tags associated with the blog post	string[ ]																										
publishedOn	Date-Time on which blog post was published	string																										
thumbnailFilename	The name of the thumbnail	string																										

	lastUpdated	The date-time on which the blog post was last updated.	string
	urlFragment	The url fragment associated with the blog post	string

It will have the following functions:

- `_fetchBlogHomepageDataAsync()` : It will return a promise containing data to be displayed on the homepage. When the blog homepage is loaded by the user it will place a request on `'/blogdatahandler/data'` handler.
- `registerBlogPostViewAsync()` : Taking `blogPostId` as parameter it will place a put request to the **"BlogPostDataHandler"** in the controller layer. It will increment the number of views on the blog post.
- `registerBlogPostReadAsync()` : Taking `blogPostId` as parameter it will place a put request to the **"BlogPostDataHandler"** in the controller layer. It will increment the number of reads on the blog post.
- `registerBlogPostExitedAsync()` : Taking `blogPostId` as parameter it will place a put request to the **"BlogPostDataHandler"** in the controller layer. It will increment the number of user for the respective time bucket depending on the amount of time user stayed on the blog post.
- `_fetchBlogPostPageDataAsync()` : It will take `blogPostId` as parameter and will place a get request to **"BlogPostDataHandler"**. It will load all the data on the blog post page.
- `fetchBlogPostSearchResultAsync()`: It will take `searchQuery` as parameter and will place a call to `BlogPostSearchHandler` and will load results on blog post search/filter By page.
- `isSearchInProgress()`: Will return true if the search query is being executed.

#### Inside `search-tag-filter.component.ts` -

- `isSearchInProgress()` : Will return true if the search query is being executed.
- `updateSearchandFilterByFieldsBasedOnUrlQuery()` : This function will update the search input field and the value in tag filter dropdown based on query parameters in the url.

- tagFilterDropDown() :It will be responsible for loading values in the tag filter dropdown menu.
- onSearchQueryChangeExec(): It will update the url of the webpage according to the search/ filter applied.
- searchToBeExec(): Function will register input event and will result in execution of the search.
- onMenuKeyPress() : Function to allow tag filter drop down to be keyboard navigable.

```
onMenuKeyPress(
  evt: KeyboardEvent,
  eventsTobeHandled: EventToCodes): void {
  this.navigationService.onMenuKeyPress(evt, 'tagMenu', eventsTobeHandled);
  this.activeMenuName = this.navigationService.activeMenuName;
}
```

#### Inside search-tag-filter.component.html -

- Text input type search field in which input is not changed will call searchToBeExec() function. It will bind input to searchQuery variable using ng form module.

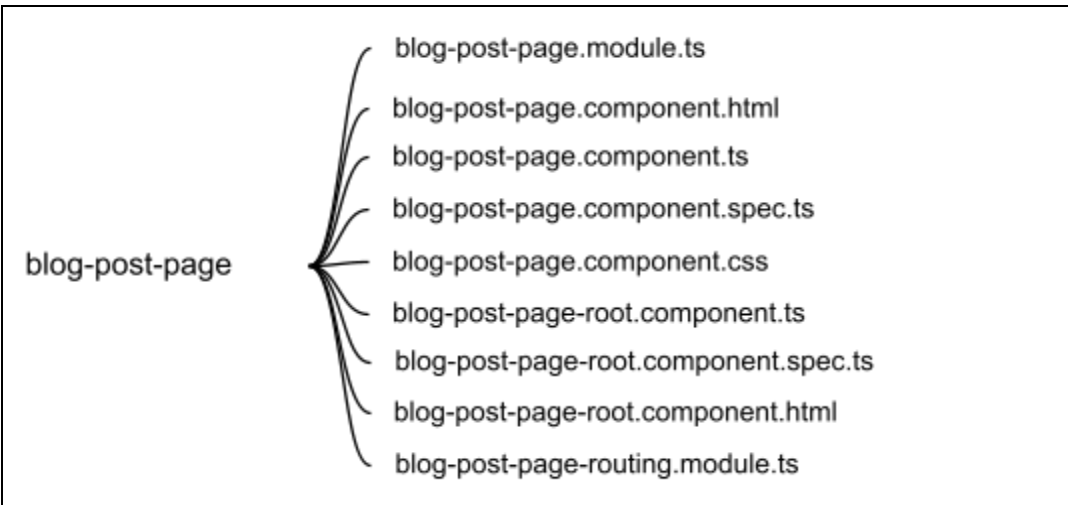
```
<input type="text"
  class="form-control oppia-search-bar-input oppia-search-bar-text-input"
  [placeholder]="searchBarPlaceholder"
  [(ngModel)]="searchQuery"
  (ngModelChange)="searchButtonIsActive ? null : onSearchQueryChangeExec()"
  [ngModelOptions]="{ updateOn: 'blur', standalone:'true' }"
  aria-label="Search bar"
  (keydown.enter)="$event.target.blur()"
  (input)="searchToBeExec($event)">
```

- Tag Filter Dropdown in which all the options will be the available tags for categorization of blog posts. We will use NgbDropDownToggle.

```
<button NgbDropDownToggle
  (click)="openSubmenu($event, 'tags')"
  (keydown)="onMenuKeyPress($event, 'tags', {shiftTab: ACTION_CLOSE, enter: ACTION_OPEN})"
  type="button"
  class="btn">
```

'\*ngFor' will be used on tagsList and will be tracked by index. Clicking on any of the tag options will call 'searchToBeExec(\$event)'.

### 3. For Blog Post Page -



#### Inside blog-post-page.component.html -

- It will have a share icon clicking on which a pop up with different methods to share the blog post will open. We will use the already existing 'sharing-link.component' in this pop up. We will use '\*ngIf' to remove the option to share on 'google-classroom' and option to 'embed code' which is not required in case of blog posts. We will provide an option to copy the webpage url which can then be shared across any platform.
- At the bottom of the blog posts, Tag buttons associated with the blog post will be shown. Clicking on any of them will lead to 'search/ filterBy' page where all the blog posts associated with these tags will be shown. Clicking on the tag button will call filterBlogPosts(tag) in the component.ts file.
- At the bottom of the page - In 'suggested for you', blog-card component will be used to display the 2 blog cards. Clicking on them will load the blog post page for the respective blog card.
- To allow blog-post-page to be accessible in all languages following I18N Keys will be introduced and will be used in the HTML template:

I18N KEY	VALUE
I18N_BLOG_POST_PAGE_SUGGESTED_FOR_YOU_HEADING	Suggested For You
I18N_BLOG_POST_PAGE_COPY_LINK	Copy
I18N_BLOG_POST_PAGE_TAGS_HEADING	Tags

### Inside blog-post-page.component.ts -

- suggestedBlogPostSummaries: list of blog post summaries suggested based on the current blog post.
- filterBlogPost(tag): Function will call execSearchQuery() function in blog-home-page.services.ts leading to execution of a search for the given tag filter.
- To **record reading time** on a blog post:  
Inside ngOnInit() we will be getting the date and time as soon as the user comes to the blog post page using function - 'new Date()'. We will then get the date and time when the visibilityState of the page changes (We will be utilising ngx-page-visibility library - [link](#)). Subtracting the 2 will give us the reading time of the blog post which will then be stored in a variable - 'readingTime'. When the user returns to the webpage we will again record the reading time until he leaves the webpage again and add it to the variable. Finally when user navigates to other webpage from the blog post or closes the tab we will update the reading time in the backend by using ngOnDestroy hook to register blog post exited event and increase number of users in reading time stats accordingly. We will also register blog post exited event in case the reading time exceeds 45 minutes or 5\*estimated\_reading\_time which ever is greater.
- To have **number of reads** on a blog post:  
Inside the function 'isBlogPostRead()' we will check if the reading time has exceeded 50% of the time calculated using number of words in the blog post and as soon as the function value returns true we will update the number of reads in the backend.

### Inside blog-post-page-root.component -

- Tags associated with the blog posts will be used as metaTags to optimize search engines. page-head.service.ts will be used to provide blog post tags as a list of strings to MetaTagData content for the webpage.

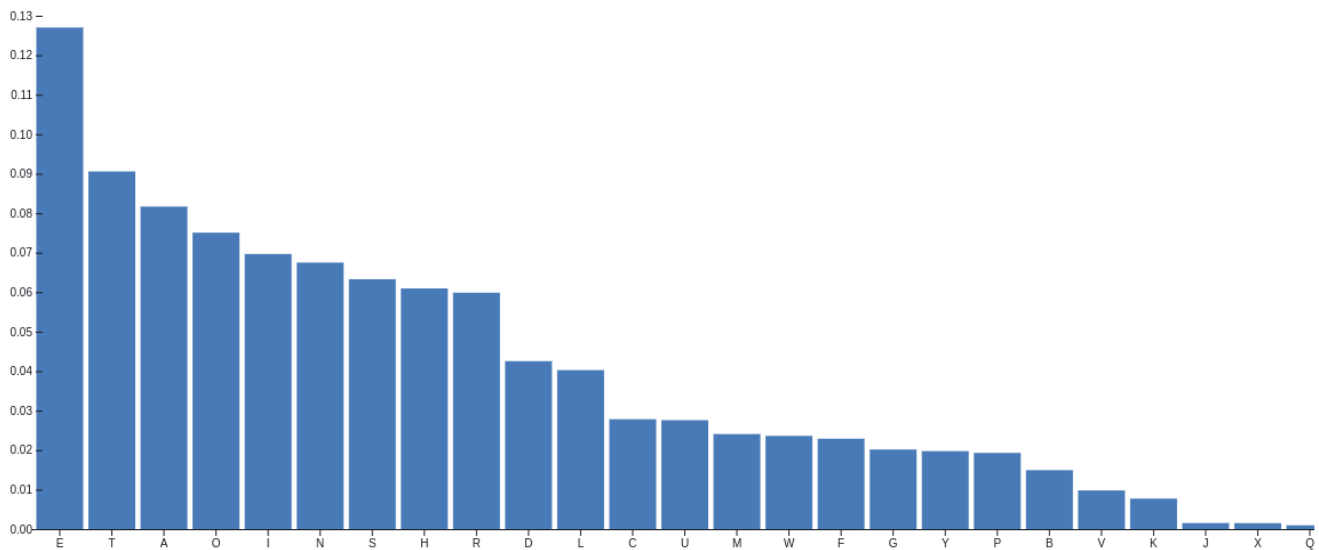
## 4. For Blog Dashboard Statistics Tab -

### 1. Inside blog-dashboards.component.html:

- New 'mat-tab' will be added in 'mat-tab-group' - 'Statistics'.  
Clicking on statistics will load the stats tab with the views on all the blog posts by the author displayed all together on a histogram.

## 2. Inside statistics-tab.component.html:

- To generate histograms from the stats data we will use - D3 library. A rough histogram generated using the library. It will be further refined to match our case -



- On the left side of the histogram list of published blog post by the author will be displayed. Clicking on the title of the blog post will load the blog post stats in the histogram. Clicking on the title will call `loadBlogPostStats()` in the component.ts file.

## 3. Inside statistics-tab.component.ts -

- `loadBlogPostStats(blog_post_id, chart_type)`: Will load data for blog post statistics histogram. It will call `loadBlogPostStatsAsync(blog_post_id, chart_type)` function in `blog-dashboard-backend-api.service.ts` which will hit the `BlogPostStatsHandler` to get the required data.
- `loadAllBlogPostsStats(author_username, chart_type)` : Will load data for blog post statistics histogram. It will call `loadAllBlogPostsStatsAsync(author_username)` function in `blog-dashboard.service.ts` which will call `fetchBlogDashboardStatisticsDataAsync()` in `blog-dashboard-backend-api.service.ts`.

## 4. Inside templates/domain/blog/blog-dashboard-backend-api.service.ts -

- `fetchBlogDashboardStatisticsDataAsync()` - It will load data for the statistics tab in the blog dashboard. It will hit 'BlogPostStatsHandler' with GET request.

- It will import class 'BlogPostViewsStats' from 'core/templates/domain/blog/blog-post-stats.model.ts' which will be created. Domain Object created from the backend dictionary will have the following structure:

BlogPostViewsStats	It will contain data from BlogPostStatBackendDict. It will have the following fields:		
id	Id of the blog post	String   Null  Null incase stats are for all the blog posts all together.	
hourlyViewsList	Dict of key value pairs where key is the hour and value is the number of views in that hours	Stats{}	
weeklyViewsList	Dict of key value pairs where key is the date and value is the number of views in that day for past 7 days (including the ongoing day)	Stats{}	
monthlyViewsList	Dict of key value pairs where key is the date and value is the number of views in that day for the past 30 days (including the ongoing date)	Stats{}	
yearlyViewsList	Dict of key value pairs where key is the month and value is the number of views in that month for	Stats{}	



		all th a psast months of the ongoing year (including the ongoing month)	
	allViewsList	Dict of key value pairs where key is the year and value is dict with key as month and value as number of views in that month for all the past years and the ongoing year	Stats{}

**Similary BlogPostReadStats and BlogPostReadingTimeStats will be formatted.**

## Documentation changes

In Oppia Github Wiki, '[Editor Pages](#)' will be edited to add guidelines for adding new blog post. Steps to assign blog post admin role and then blog post editor role will be included.

## Testing Plan

### E2e testing plan

#	Test name	Initial setup step	Step	Expectation
1.	Search and read blog posts  It should increment blog post stats	Populate blog homepage by dummy blog posts	Create and publish dummy blog posts associated with different tags using the blog dashboard.	Blog Homepage should have published blog cards
			Navigate to blog homepage at /blog	
			Input search query in search bar.	Url of the webpage should get updated with the search query. Blog Homepage should get loaded with correct search results
			Click on any blog card	Blog post page with the correct blog post should get loaded. It should have appropriate blog cards shown in

				'suggested for you' section
			Click on any of the tag button	Blog Home page with filtered blog post for the given tag should get loaded. Url of the webpage should have the appropriate query string.
			Click on any blog card	Blog post page with the correct blog post should get loaded.
			Click on any blog card in 'suggested for you' section	Blog post page with the correct blog post should get loaded.
			Navigate to /blog-dashboard statistics tab	It should show correct statistics of the current hour.

### 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.

### Backend tests:

All the backend files will be accompanied by their test files.

### Lighthouse tests:

All the new pages url will be added to the lighthouse.js, and lighthouse-accessibility.js to perform accessibility tests on the webpage.

### Feature testing

Does this feature include non-trivial user-facing changes? **YES**

## Migration Of Blog Posts From Medium to Oppia

I'll make a manual transfer because, while Medium allows you to export your blog post, it just presents the material as an html file. As a result, a manual transfer will be significantly easier, whereas attempting an automated method will necessitate additional codebase features. The fact that some free plugins exist that allow rendering of medium blog posts in wordpress editor format after being provided the html files in zip format (which is downloaded from medium itself) will contain a few unnecessary fields such as claps and feedback threads that aren't handled by the oppia's blog dashboard interface. Furthermore, the formatting that is obtained will not be the same as that which was planned. Because we don't support the features of the medium's blog editor in the same manner, our editor may not support the html tags as intended. Automated transmission may appear to be simple to implement and will undoubtedly save time, but it can have significant unintended

consequences. Furthermore, manual transmission ensures that all blog posts are transferred without formatting issues, which is difficult to guarantee with an automated transmission.

**The Blog Admin Page has a form to edit the published date and author name of blog entries, which will be used to manually migrate blog posts from 'Medium.'**

[ Implemented in GSoC' 2021 project - reference docs - [Exporting Blog Posts From Medium](#), [Form Implementation In Blog Admin Page](#) ] [ Milestone 2.4 B : [PR link](#) ]

### **STEPS TO BE FOLLOWED TO MIGRATE BLOG POSTS:**

Users should be logged in as Blog Post Admin.

Step 1 : Copy Paste all the blog post content in the blog post editor. Add the title of the blog post in the title field. Select appropriate blog post tags and publish the blog post. From the url of the webpage note down the blog post id of the blog post. (It is a 12 character string at the end of the url)

Step 2 : Go to admin-page - misc tab

Step 3 : In form to update blog post data - Enter blog post id, author username of the original author and the original published on date. Click on Update.

**On medium Oppia's Blog Exist from year-2014. In total from 2014 to 2022 there are 44 blog posts. Migrating about 15 blog posts each day will ensure that all the blog posts get migrated within a span of 3 days.**

## Implementation Plan

The implementation plan consists of two milestones.

I will be working from July 20 on my GSoC project.

All the frontend views i.e mobile, tablet and laptop views will be done together.

### Milestone 1 Table

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
-----	----------------------------	-------------------	-----------------------------	---------------------------------

1.1	Issue number <a href="#">#13397</a> will be addressed. <ul style="list-style-type: none"> <li>- Edits in blog_servies.py file will be made.</li> <li>- In jobs/transforms/validation/blog_validation.py edits will be made accordingly.</li> </ul>	-	July 24	July 28
1.2	Modifications in Storage, Domain and Controller layer will be done to create blog post search and filter functionality. <ul style="list-style-type: none"> <li>- blog_post_search_services.py file will be added</li> <li>- Blog_post_search_indexing_jobs file will be added.</li> <li>- Related edits in blog_services.py file and blog_homepage.py will be done</li> </ul>	-	4th August	10th August
1.3	Blog Home Page Frontend will be done. <ul style="list-style-type: none"> <li>- blog-home-page.component will be done.</li> <li>- blog-card.component will be accordingly edited.</li> </ul> <p>Ensure that blog posts are properly pseudonymized when the blog post author deletes his/her account. Attach video proof in the PR link.</p>	1.2	14th August	20th August
1.4	Blog Post Page Frontend will be done. <ul style="list-style-type: none"> <li>-blog-post-page.component will be done</li> </ul>	1.3	23rd August	30th August

## Milestone 2 Table

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
2.1	Migration of Blog Posts from Medium To Oppia. ( Functionality to migrate blog posts is already deployed onto the main site.)	1.1	-	To be completed by 5th Sept
2.2	Changes in Storage, Domain and	-	8th Sept	14th Sept

	Controller layer for Statistics functionality will be done.			
2.3	Blog Author Page Frontend will be done.	1.2, 1.3	14th Sept	18th Sept
2.4	The front end of the Statistics tab will be done.	2.3	23rd Sept	30th Sept
2.5	E2E tests for blog homepage functionality will be done	Milestone 1, 2.3, 2.4	October 5th	October 10th
2.6	A week to fix reported bugs and issues			October 17th

Final deadline for GSoC students with extended work period is : Nov 21, 2022 .

### Launch Plan:

- As soon as 1st Milestone PRs are deployed on the server, Blog Editor Team can be allowed to explore the functionality of blog dashboard and blog home page. This will make them comfortable with the functionality before we completely make a switch from medium to our site. In the meantime they can also queue up posts which need to be published once everything is in place! Therefore, I believe by the end of September we can have a blog team make use of the existing functionality for writing blog posts and have them on the blog homepage.
- Feature review for both blog homepage and blog dashboard functionality will also have to be done to ensure no major bug remains in the features and fixes for the issues found will have to be done before switching from Medium. This will be completed in November after all the PRs for fixing bugs are deployed on the server.
- Therefore, the Ads team can work after the November release cut is made and all the PRs made in the month of October are deployed.

Things To Be Done	Prerequisite	Expected Dates
Feature Review of Blog Dashboard ( by Diana and Blog Editors Team)	-	Aug 15, 2022
Addressing all the issues found during blog dashboard feature review	Blog Dashboard Feature review is done	Late August
Migrating Blog Posts from Medium to Blog Home Page	M1 is complete, issues found in blog dashboard feature review are fixed	Mid September

Blog Dashboard is available for blog Editors Team to explore and queuing up posts to publish	M1 is complete and Migration of Blog Post from Medium to Oppia is done and issues found during migration are fixed (if any).	Late September
Feature Review of Blog Homepage (by Diana and Blog Editors Team)	M1 is complete	Late September
Addressing Issues from feature review of Blog Homepage	Feature review for Blog Homepage is done	Early October
Blog Homepage and Blog Dashboard are available for Blog Editor Team for use and complete transition from Medium to Oppia	All the PRs addressing issues from feature review are merged and released on the website	Late November

## Future Work

- In future, we can have a functionality for blog post viewers to 'clap' for the blog post as similar to that in medium.
- We can also allow users to comment their views on the blog post.
- We can add a functionality to sort and search for blog posts in statistics tab as well