Learner Dashboard

Author: Arunabh Ghosh

Why am I interested in working with Oppia?

A year back when I came across Oppia, its mission - to help anyone learn anything in an effective and enjoyable way resonated well with my own personal beliefs about democratizing knowledge by making it readily available and as simple and comprehensible as possible. As I began to explore Oppia I fell in love with the concept of explorations - the fact that the learner is embarking on a journey to explore the unknown is at the very heart of the learning process and an exploration perfectly catches this metaphor. Since then I have been an active member of the community creating explorations -- the 'Get Electrified' series, fixing issues -- about 26 in total, ranging from displaying top rated and recently published explorations in the library page to fixing bugs in the math expression input, adding docstrings to files like stats_jobs_continuous and stats_domain, and implementing new features like the subscription functionality and it has been a worthwhile and enriching experience. I can say with a good degree of confidence that going forward I will further increase my involvement and would like to contribute in many more ways such as taking up more challenging projects and creating new explorations. I believe participating in the coveted Google Summer of Code 2017 through Oppia would be the ideal way to strengthen my relationship with Oppia and boost my involvement.

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

I believe Learner Dashboard is one of the most significant projects of GSoC 2017. It will be central to the core user experience of a learner. A learner would finally feel a part of the Oppia community as the learner dashboard would be unique to him/her only. He/She would be able to track his/her progress, save explorations and collections for playing them later, and view the creators to whom he/she has subscribed. This, I believe is essential if we want to better engage the learners with Oppia. This year Oppia is focusing on creating a set of basic mathematics lessons for students in rural areas and the learner dashboard will be the cornerstone of its success by providing an enriching learning experience. This is why it is essential that we complete the learner dashboard by this summer itself. By completing this project I would get the satisfaction of completing a challenging project and contributing towards a significant goal of Oppia.

Prior Experience

I have been a contributor to Oppia for 9-10 months and have made several contributions - from fixing issues to ideating and implementing new features. Here is a complete list of my contributions to Oppia on the technical side. I'll describe two of my most significant contributions which I feel will also substantially help me in implementing the Learner Dashboard.

- Subscriptions Project - Learners are now able to subscribe to their favorite creators and learn from explorations created by them. This would help improve the overall quality of explorations on Oppia, as good creators now have an additional incentive to continue to create quality explorations. When creators publish a new exploration, an email is sent to his/her subscribers informing them and inviting them to try out the new explorations. Here is the link to some of the PRs for this project - 6th Milestone, 5th Milestone, 4th Milestone, 3rd Milestone, 2nd Milestone, 1st Milestone.
- Pages for top rated and recently published explorations - These pages fetch the 20 most top rated/recently published explorations and display them in a sorted order. Here is a link to the PR - #2507.

From my contributions to Oppia, I have gained a pretty fair idea about the backend and frontend aspects of Oppia. Being a user and a creator myself, I also have a good idea about what goes into making a delightful and cohesive user experience for both learners and creators. I recently wrote a blog on my 'Teaching Experience' which describes my journey of teaching with Oppia. It can be found over here.
Further, I am proficient in Python and AngularJS. Apart from my projects for Oppia here is a list of all my projects on Github:
Python - Python Projects.
Web - Rich Text to Markdown, Online Knowledge Sharing Platform.
Fun Fact: After a long time of developing with Oppia I just realized my level of familiarity with the demo explorations :P. For example in 'Welcome to Oppia!' I can get the location of Helsinki right in my first attempt, without zooming into the map.

Project Plan and Implementation Strategy

Learner Flow

Synopsis of the Design Philosophy

I believe the goal of learner dashboard should be – “It should welcome each learner as this is one page that will be personalized for each and every learner on Oppia. On reaching the dashboard - he/she should be proud of his/her progress and be able to pick up where he/she left off. It should welcome the learner and motivate him/her to explore further.”
I have given due consideration to the aesthetic and functional aspects of the dashboard. The design is based on -

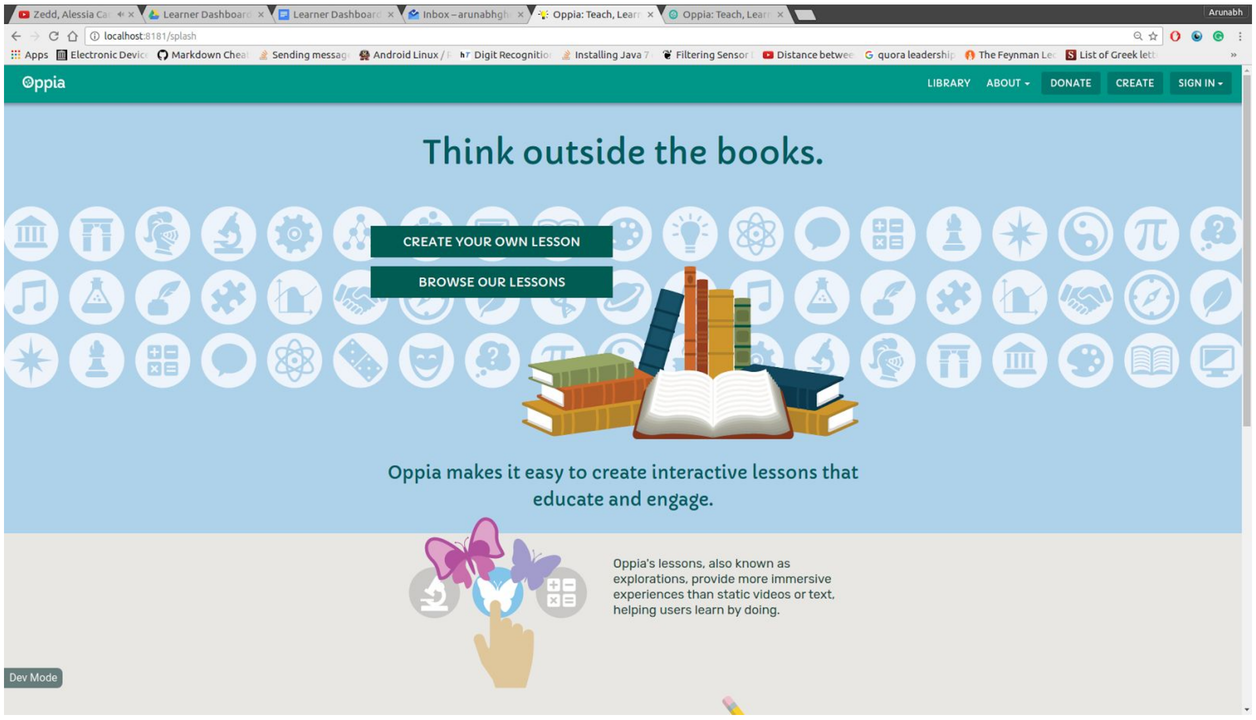
- ❖ The best usability practices followed in contemporary user dashboards – about which I did some self-study and also checked out some sites to validate the same.
- ❖ 11 User interviews, 17 proper peer reviews (don't know how many informal peer reviews) and feedback.
- ❖ Consistency with the overall design and user experience of Oppia so that navigating in and out of the dashboard is a seamless experience for the Oppia users.
- ❖ My own thought as to what information should be available on the dashboard – here my own experience as an Oppia user has come in handy as I created the first draft based on the features which I wish were there as a user.

Based on my research, two design layouts arose for our learner dashboard, both well suited to become our learner dashboard. However, only one could become our learner dashboard. Both

were put up against each other, analyzed, evaluated against various parameters and finally one emerged victorious over the other!
Note: For a detailed analysis of the design, please refer to this document.

We start by explaining how the learner gets to his/her dashboard. Effort has been taken make the process as intuitive as possible and seamlessly integrate the dashboard into the learning process. Next, the actual dashboard is presented -- packed with powerful features to help the learner learn faster. better.

Reaching Oppia, the beautiful splash page will greet the user. The behavior of the splash page will depend on whether the learner is signed in or not. If the user is signed in, we can safely assume that the user has been to the site (at least once :P) and is somewhat familiar with the concept of the site. If he/she is not signed in, there's a chance that the user has never been to the site and so the options presented to him/her must be easily understandable in layman terms. The current splash page serves this purpose well. Therefore if the user is not signed in, this is how the splash page will look like.

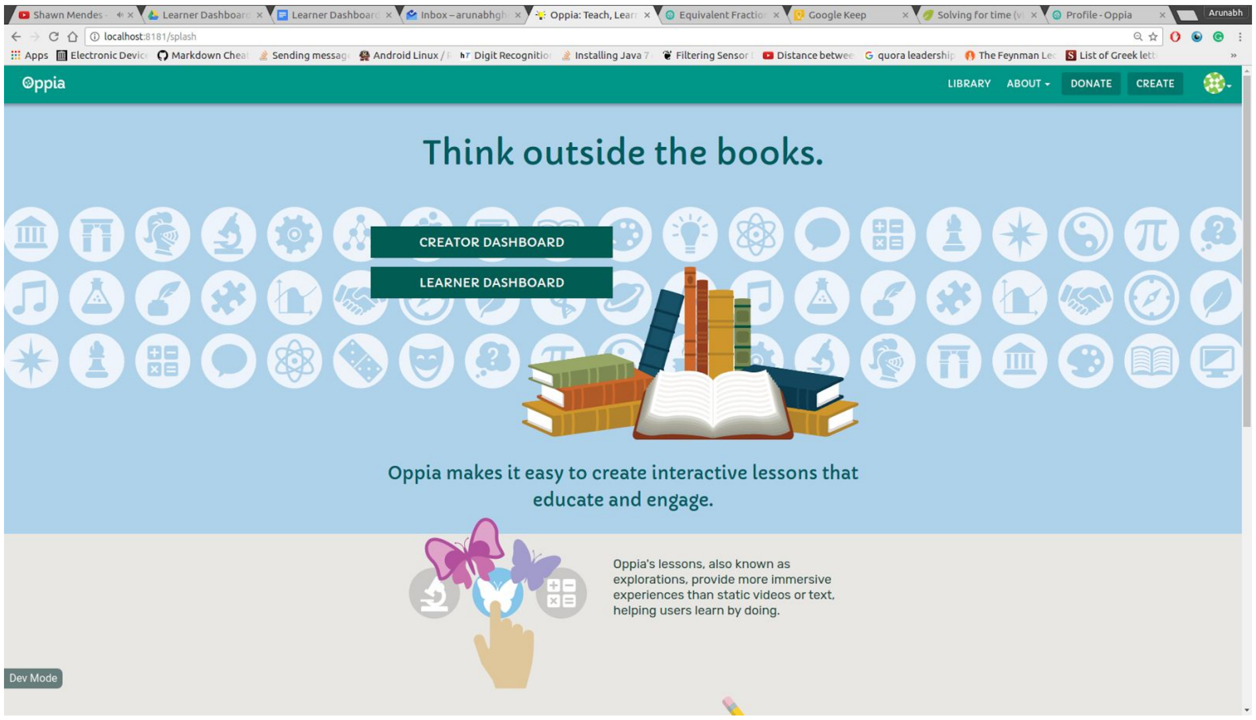


Think outside the books.

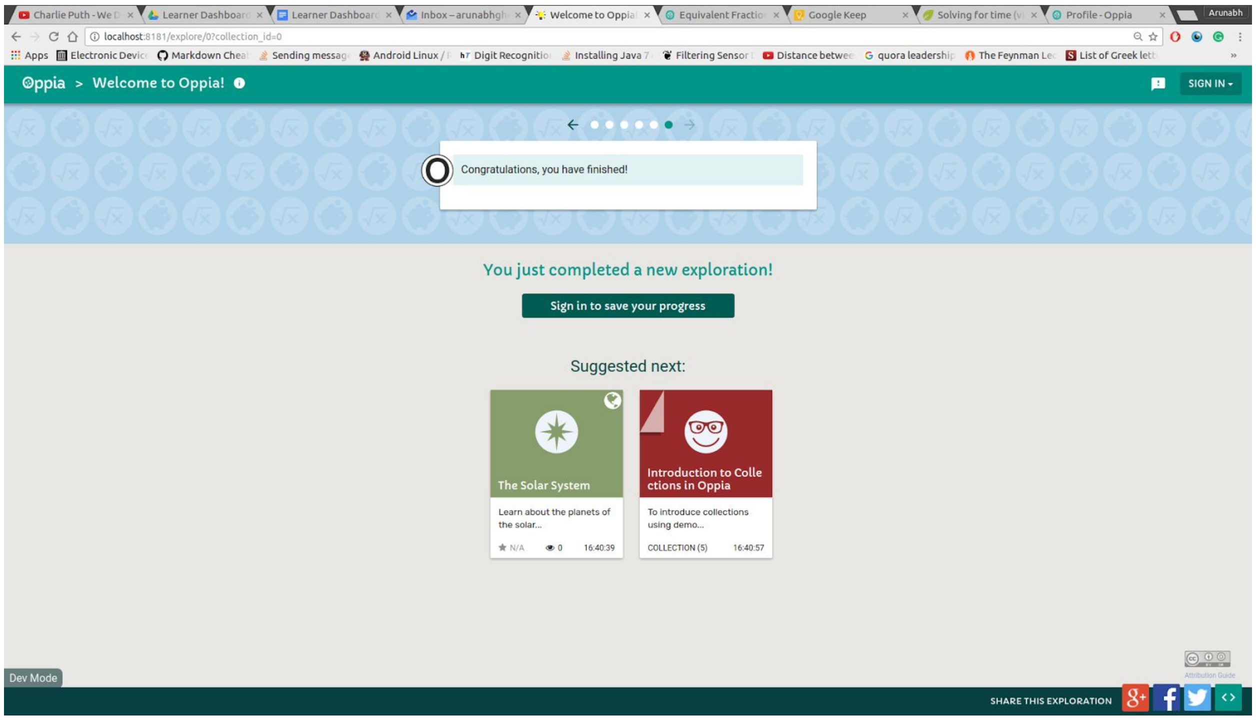
Oppia makes it easy to create interactive lessons that educate and engage.

Oppia's lessons, also known as explorations, provide more immersive experiences than static videos or text, helping users learn by doing.

If the user is signed in, the splash page must be designed to quickly take the user to his/her destination. A returning user usually has a certain goal in his/her mind - to learn or to teach (these are not my opinions, all this is user tested). Therefore there are two options presented on the splash page, both very relevant to the user -- learner dashboard and creator dashboard. If a user wants to learn, he/she will proceed towards his/her learner dashboard. If he/she wants to create, he/she will proceed towards the creator dashboard. However, if a learner wants to try his/her hands at teaching, or a creator wants to learn a new topic, there's no stopping them from clicking the other option. Here's how the slash page will look like for a signed in user.

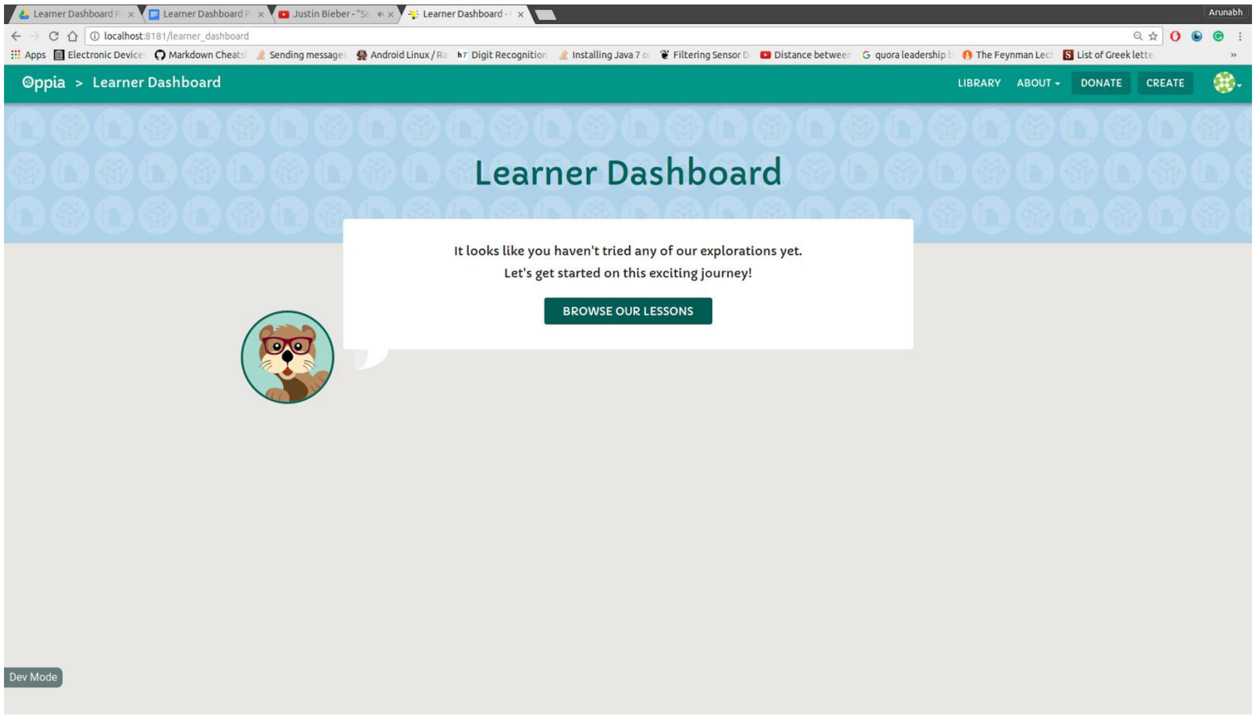

Oppia
LIBRARY
ABOUT
DONATE
CREATE
Think outside the books.
CREATOR DASHBOARD
LEARNER DASHBOARD
Oppia makes it easy to create interactive lessons that educate and engage.
Oppia's lessons, also known as explorations, provide more immersive experiences than static videos or text, helping users learn by doing.
Dev Mode

But there's one problem, right? The learner dashboard has kind of gone out of the focus for a non-signed in user. How will he/she come to know of the learner dashboard and all the powerful features it has to offer to help him/her learn faster. better. There's a solution. Here's what happens when a learner who has not signed in, completes an exploration.


Oppia > Welcome to Oppia!
SIGN IN
Congratulations, you have finished!
You just completed a new exploration!
Sign in to save your progress
Suggested next:
The Solar System
Learn about the planets of the solar...
Introduction to Collections in Oppia
To introduce collections using demo...
COLLECTION (5)
Dev Mode
SHARE THIS EXPLORATION

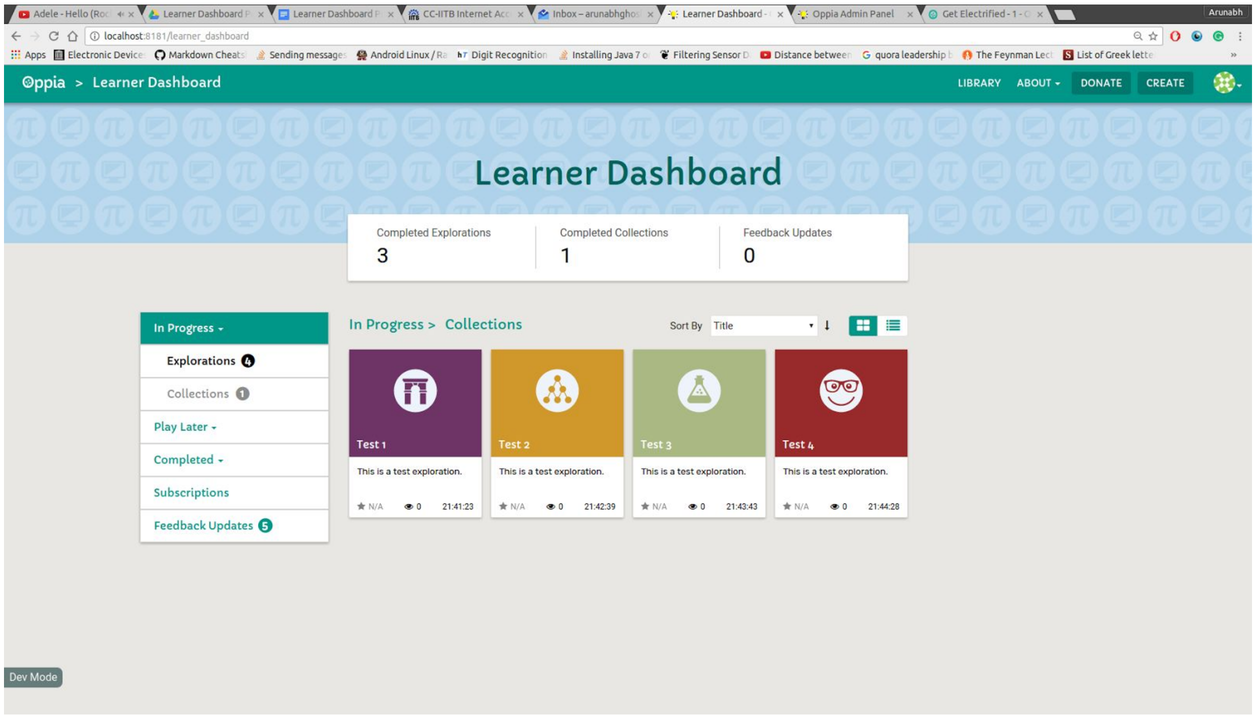
After the learner completes an exploration, he/she would be feeling motivated and would have a strong urge to continue exploring further. He/She would be motivated to save his/her progress and build a list of his/her accomplishments. The prompt to save his/her progress comes just at the right time -- and the learners would proceed towards the learner dashboard having this exploration in the completed list, thus boosting his/her morale.

Now we start with the **learner dashboard**. If a learner is just starting his/her journey with Oppia, he/she would be invited with a pleasant greeting card inviting him/her to explore the explorations at Oppia.



It looks like you haven't tried any of our explorations yet.
Let's get started on this exciting journey!

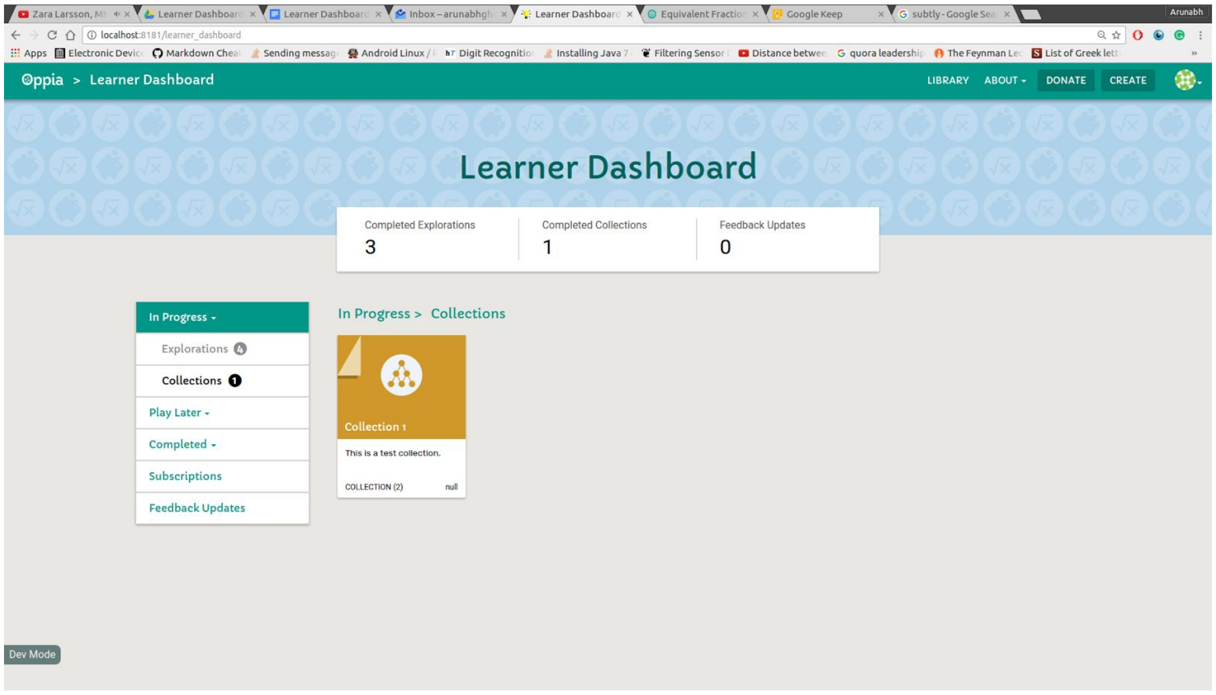
The first thing that the learner sees on reaching the dashboard is the explorations in progress. The reason that this is the first thing that learner sees is explained over here [5]. The sort by options are 'Title' and 'Categories' which learners found quite relevant (judging by my user interviews). Next to each the option in the menu, is a number indicating the quantity. Note that numbers have deliberately not been centered in the circles as one of the learners said that -- keeping the numbers a bit off, gives a nice playful touch to the learner dashboard.



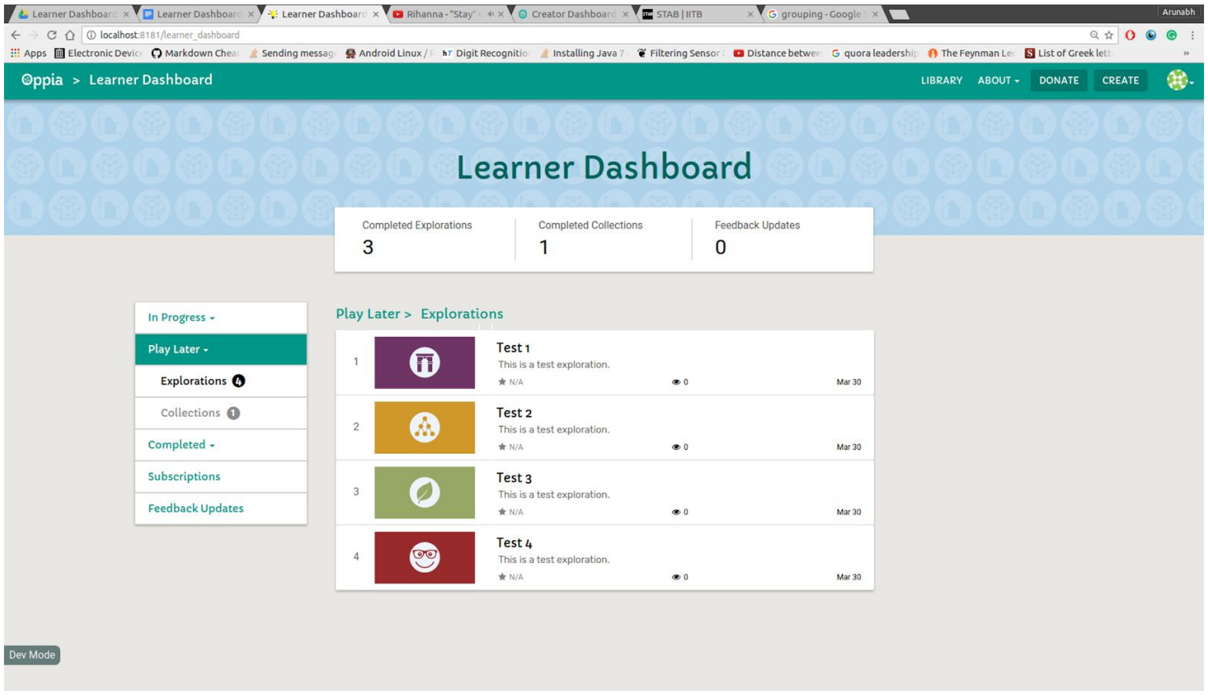
Completed Explorations	Completed Collections	Feedback Updates
3	1	0

Title	Description	Rating	Views	Time
Test 1	This is a test exploration.	N/A	0	21:41:23
Test 2	This is a test exploration.	N/A	0	21:42:39
Test 3	This is a test exploration.	N/A	0	21:43:43
Test 4	This is a test exploration.	N/A	0	21:44:28

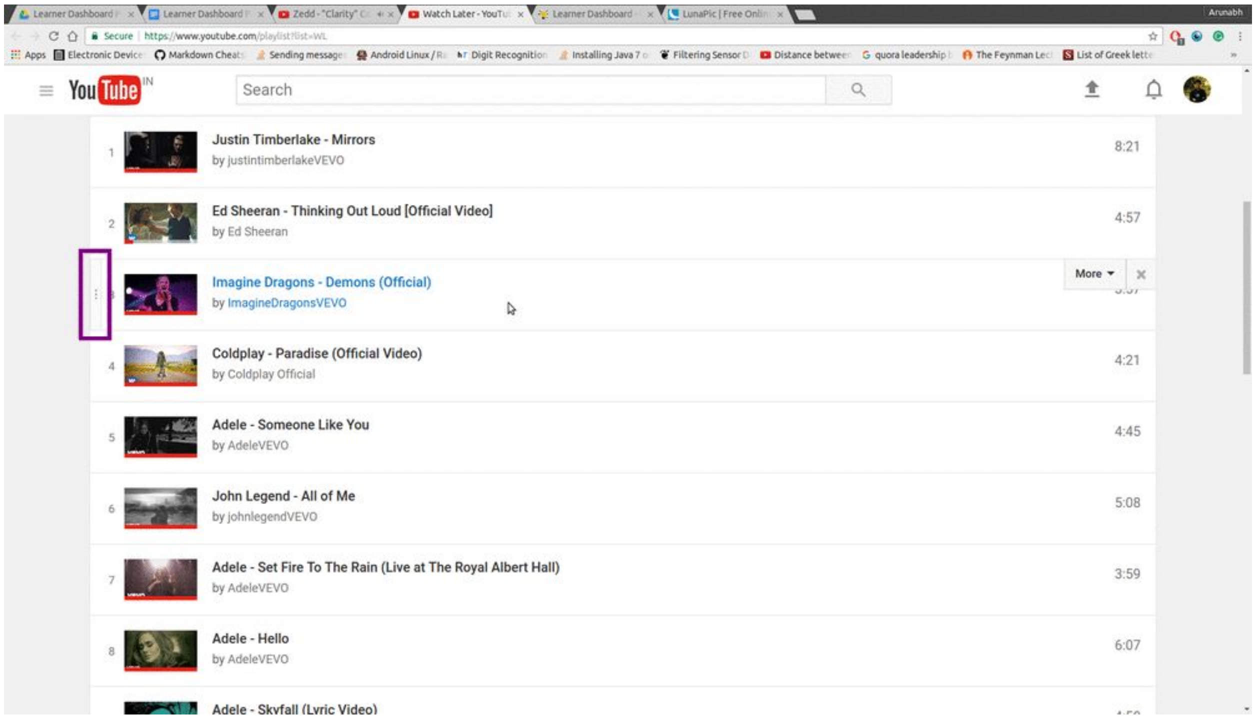
The collections, in progress. Note that learners will have the option of removing explorations and collections if they decide that they do not wish to complete them. A modal will appear asking them to reconfirm their actions.



Next, we move on to 'Play Later' feature. Here's how I decided on the name -- [Play Later](#) [4]. The list of explorations to be played later by the learner. The user can rearrange the order of explorations by dragging the exploration tiles around. If the user starts an exploration and leaves it in between, the exploration is moved from the Play Later list to the in progress section. If he/she completes it, it is moved to the completed section. Can the learner move an exploration from the in progress section to the play later list? I have addressed this question over [here](#) [2].

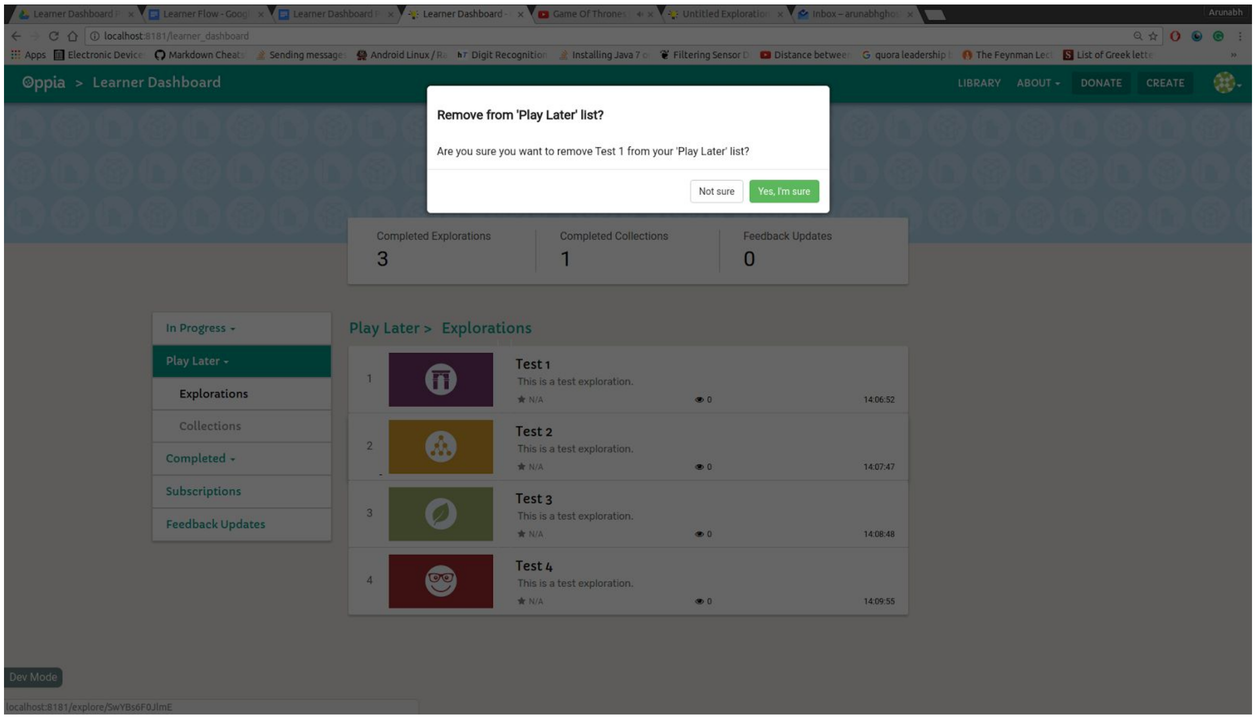

localhost:8181/learner_dashboard
Oppia > Learner Dashboard
LIBRARY
ABOUT
DONATE
CREATE
Learner Dashboard
Completed Explorations 3
Completed Collections 1
Feedback Updates 0
In Progress
Play Later
Explorations 4
Collections 1
Completed
Subscriptions
Feedback Updates
Play Later > Explorations
1 Test 1 This is a test exploration. N/A 0 Mar 30
2 Test 2 This is a test exploration. N/A 0 Mar 30
3 Test 3 This is a test exploration. N/A 0 Mar 30
4 Test 4 This is a test exploration. N/A 0 Mar 30
Dev Mode

The drag and drop functionality would be similar to Youtube's 'Watch Later' functionality. Given below is a screenshot for reference. Our tiles would also have a grip handle (shown in the purple rectangle), which users can drag to rearrange the explorations.



1	Justin Timberlake - Mirrors by justintimberlakeVEVO	8:21
2	Ed Sheeran - Thinking Out Loud [Official Video] by Ed Sheeran	4:57
3	Imagine Dragons - Demons (Official) by ImagineDragonsVEVO	
4	Coldplay - Paradise (Official Video) by Coldplay Official	4:21
5	Adele - Someone Like You by AdeleVEVO	4:45
6	John Legend - All of Me by johnlegendVEVO	5:08
7	Adele - Set Fire To The Rain (Live at The Royal Albert Hall) by AdeleVEVO	3:59
8	Adele - Hello by AdeleVEVO	6:07
	Adele - Skyfall (Lyric Video)	

Also on hovering over the exploration tiles, an option to remove the exploration from the 'Play Later' list would appear in the form of a cross on the top right corner of the tile similar to the screenshot above. Before removing the exploration, a modal would appear asking the learners to reconfirm their action.



Remove from 'Play Later' list?

Are you sure you want to remove Test 1 from your 'Play Later' list?

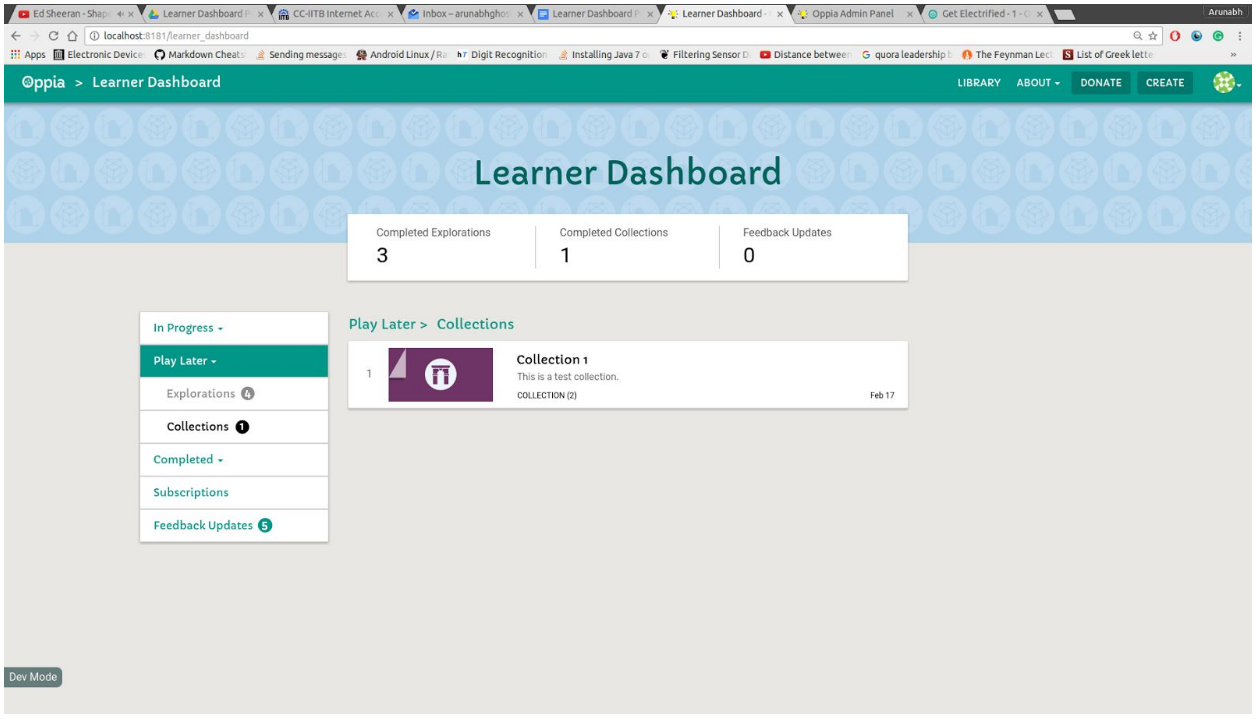
Not sure | Yes, I'm sure

Completed Explorations	Completed Collections	Feedback Updates
3	1	0

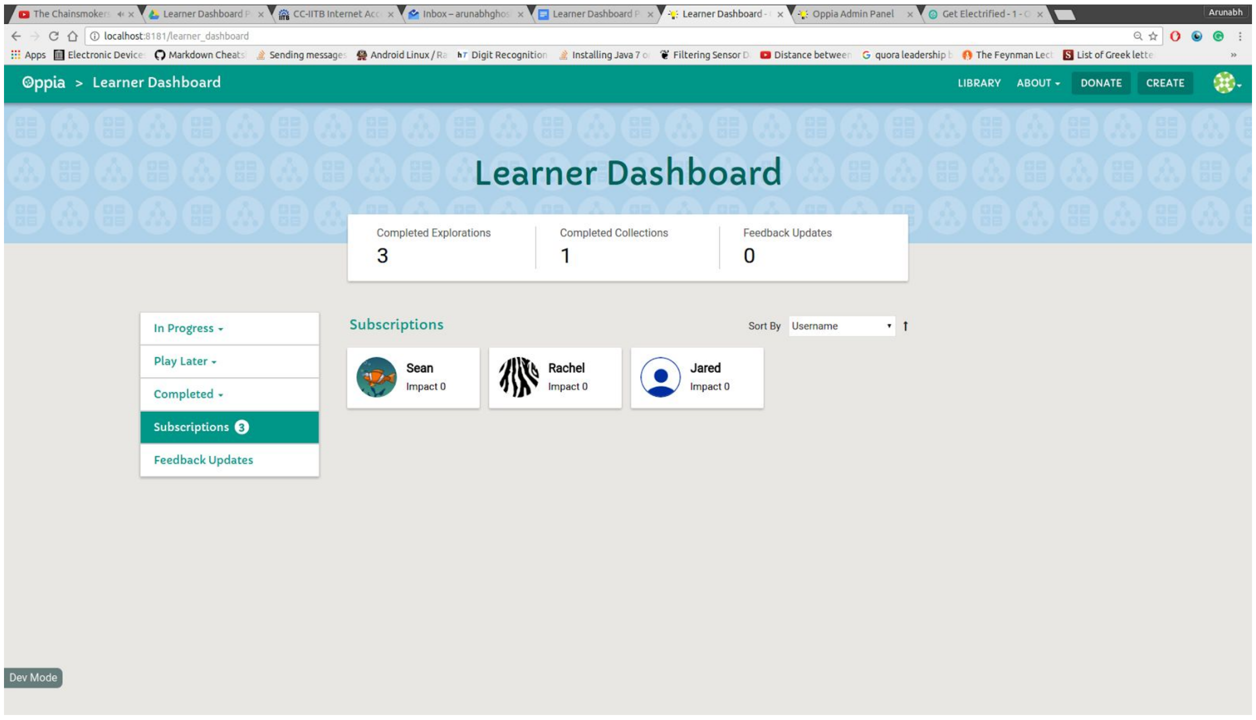
Play Later > Explorations

1	Test 1 This is a test exploration.	N/A	0	14:06:52
2	Test 2 This is a test exploration.	N/A	0	14:07:47
3	Test 3 This is a test exploration.	N/A	0	14:08:48
4	Test 4 This is a test exploration.	N/A	0	14:09:55

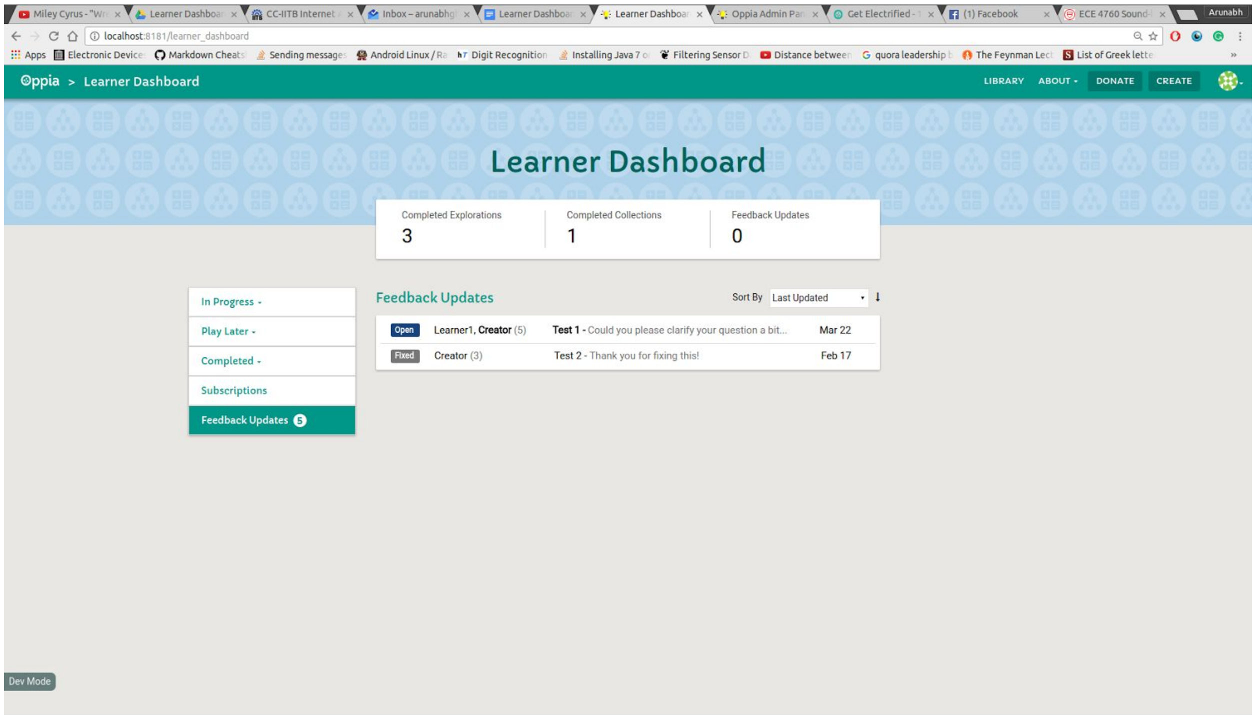
The learner can also add collections to his/her Play Later list. The behavior is same as that of explorations. I'm attaching a screenshot for reference. I would also like to address an important question - What's the rationale for separating explorations and collections into different submenus? I have answered it over here [1].



Viewing the creators to which he/she has subscribed is as easy as clicking on the subscriptions tab.

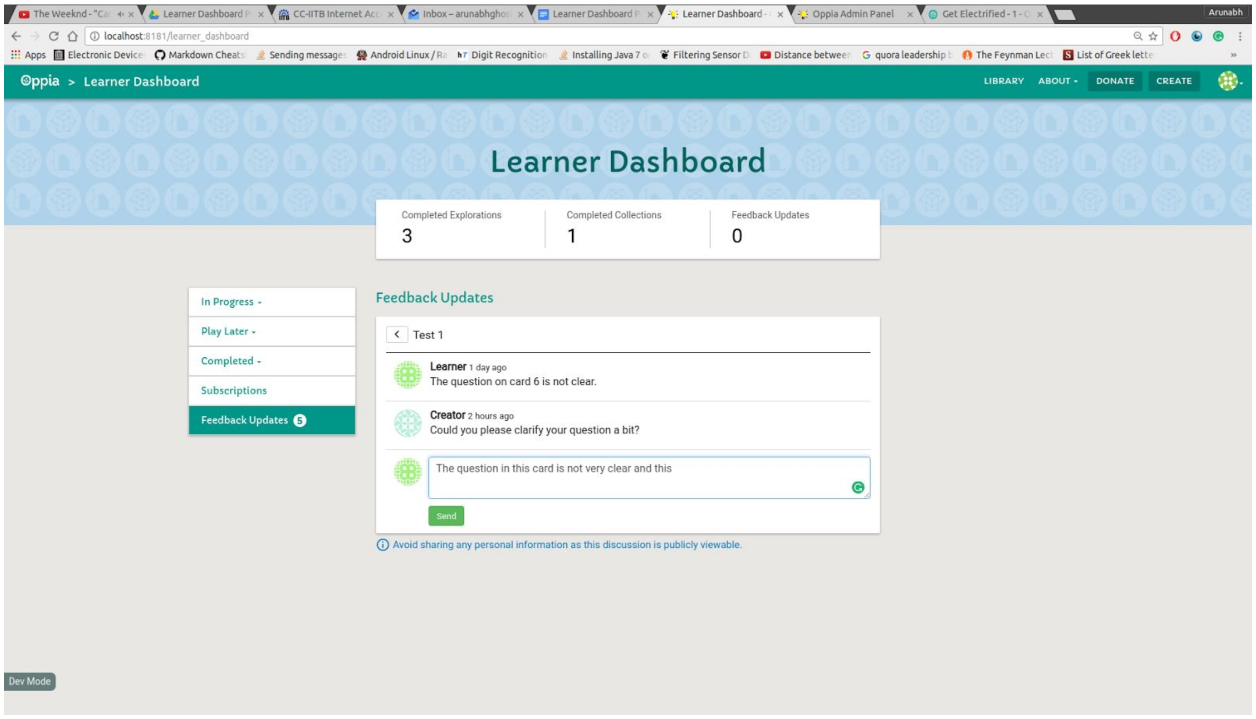


If a learner wants to view the feedback threads to which he/she is subscribed, he/she will simply head over to the feedback updates section. The feedback threads are sorted in order of last updated to allow the user to respond to threads that are most active. The other option to sort by is 'explorations' if the learner wants to find a feedback thread specific to an exploration. Why the name 'Feedback Updates'? The explanation is given here [3].



Open	Learner1, Creator (5)	Test 1 - Could you please clarify your question a bit...	Mar 22
Fixed	Creator (3)	Test 2 - Thank you for fixing this!	Feb 17

To view the messages of a feedback thread, the learner can simply click that feedback thread. He/She can view the messages and even reply directly from his/her dashboard.



Test 1

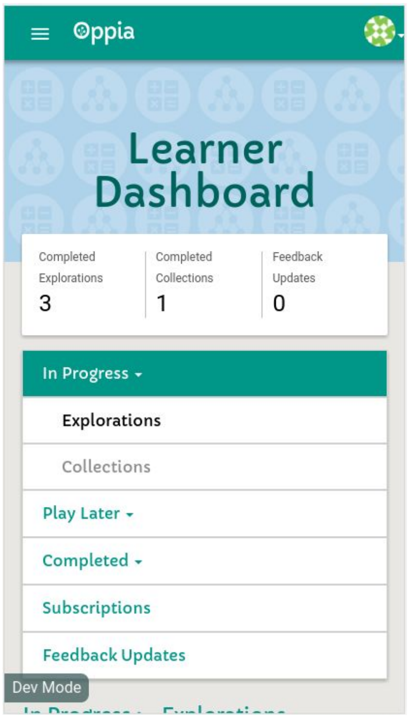
Learner 1 day ago
The question on card 6 is not clear.

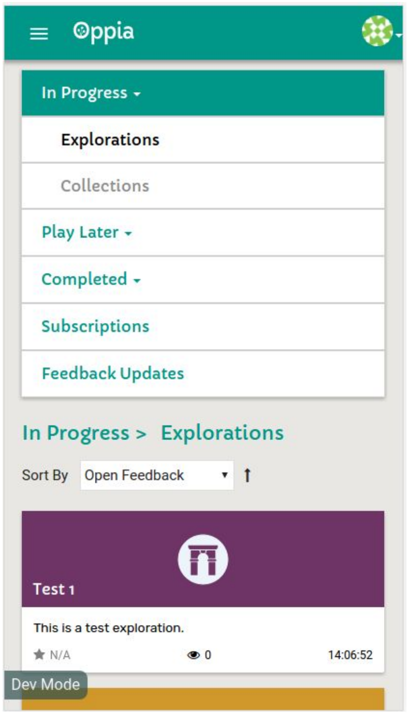
Creator 2 hours ago
Could you please clarify your question a bit?

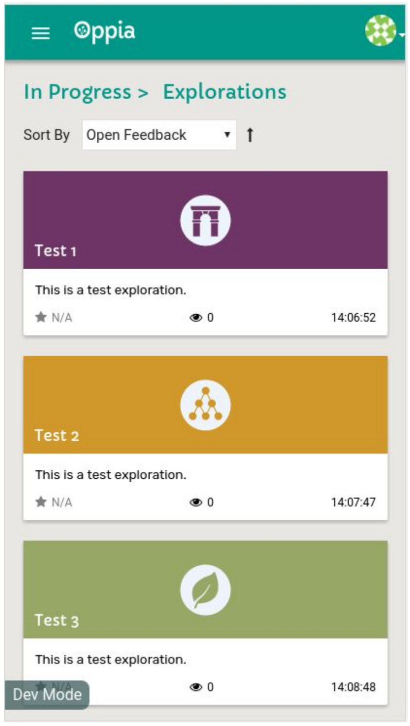
The question in this card is not very clear and this

Avoid sharing any personal information as this discussion is publicly viewable.

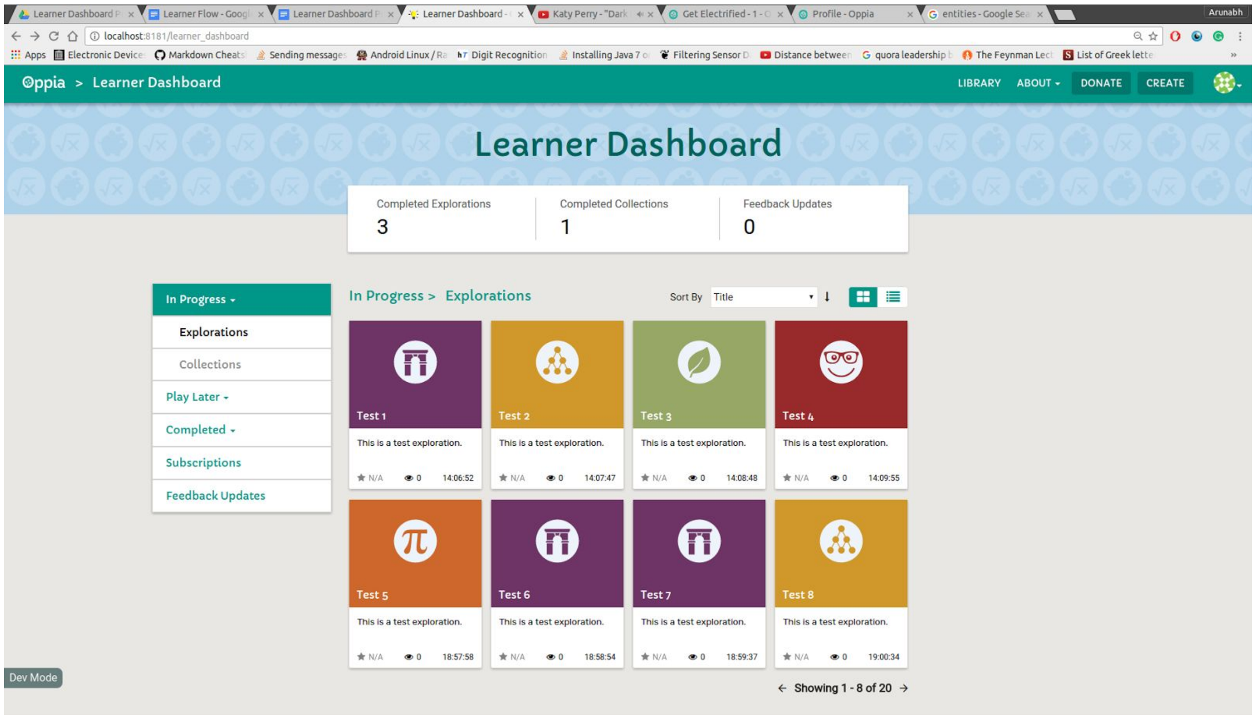
All the above design layouts can easily be extended to mobile screen width. In the screenshots below I've shown how I will extend the 'In Progress' section. All other sections will be extended in a similar manner. (The numbers on the side menu will be implemented for this as well.)



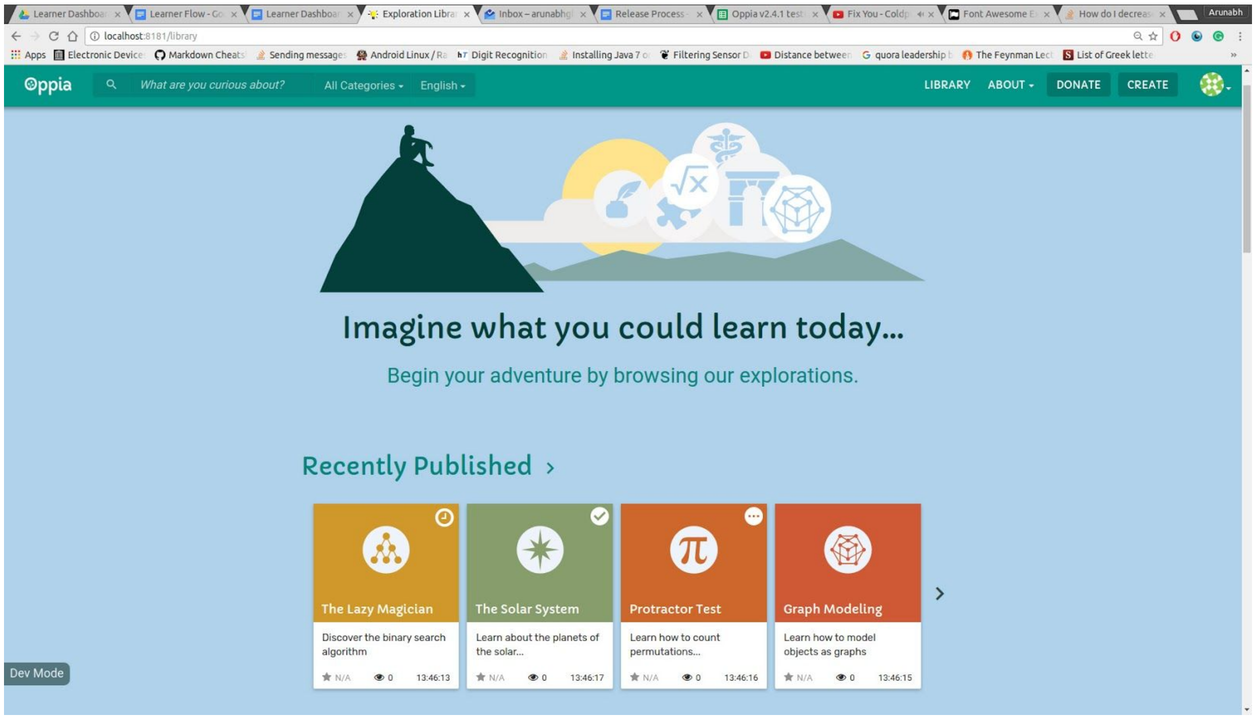



Oppia
In Progress > Explorations
Sort By Open Feedback
Test 1
This is a test exploration.
N/A 0 14:06:52
Test 2
This is a test exploration.
N/A 0 14:07:47
Test 3
This is a test exploration.
0 14:08:48
Dev Mode

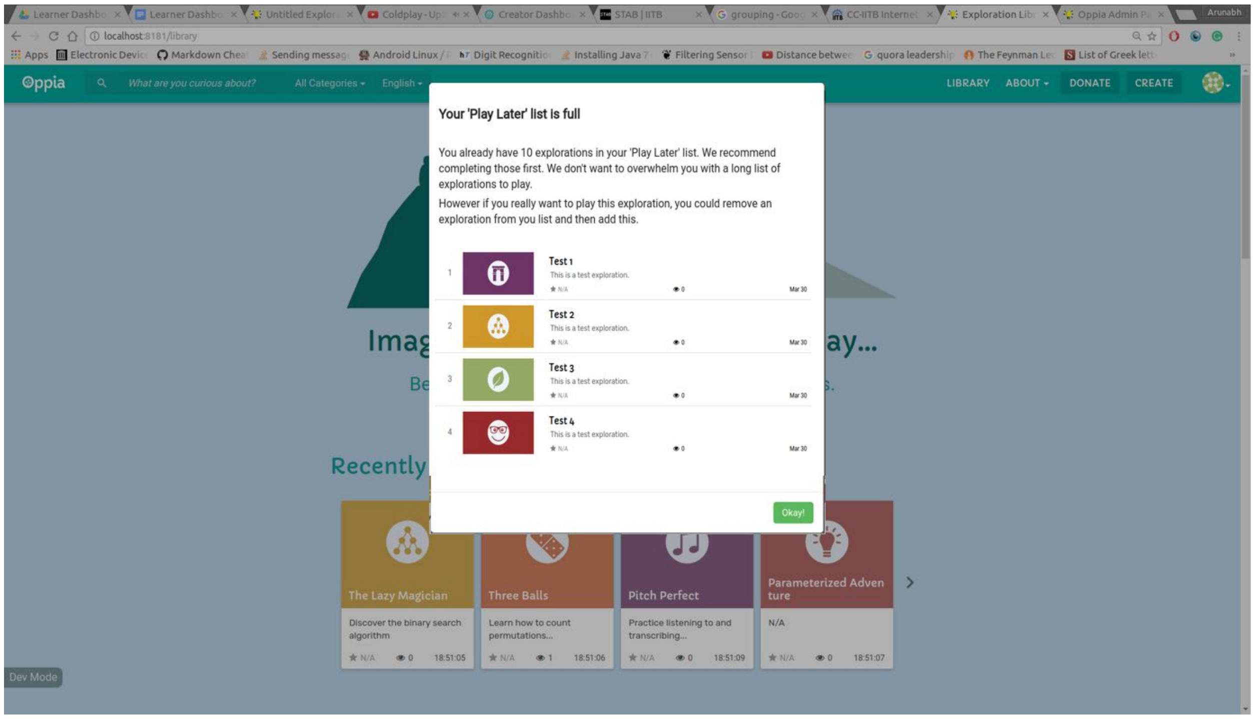
This dashboard is also easily scalable to many entities. In the layout below I've shown how we will extend the 'In Progress' explorations (same philosophy as profile page). All the other sections will be scaled in a similar manner.



The design of the exploration card indicates the user's past interaction. On the library page, small icons on the top right corner of the exploration indicate whether the exploration has been completed, is in progress or is in the 'Play Later' list of the learner. On hovering over an exploration card a small clock symbol would appear. By clicking this the learner can add the exploration to his/her 'Play Later' list (needless to say, also applicable for collections).


localhost:8181/library
Oppia
What are you curious about?
All Categories
English
LIBRARY
ABOUT
DONATE
CREATE
Imagine what you could learn today...
Begin your adventure by browsing our explorations.
Recently Published
The Lazy Magician
Discover the binary search algorithm
The Solar System
Learn about the planets of the solar...
Protractor Test
Learn how to count permutations...
Graph Modeling
Learn how to model objects as graphs
Dev Mode

The user can add a maximum of 10 explorations to his/her 'Play Later' list so that learners don't end up accumulating a large to-do list and then get kind of turned off because it feels so overwhelming later on. If the learner tries to add an additional exploration, we show a modal recommending them to first complete the explorations, already in their play later list.



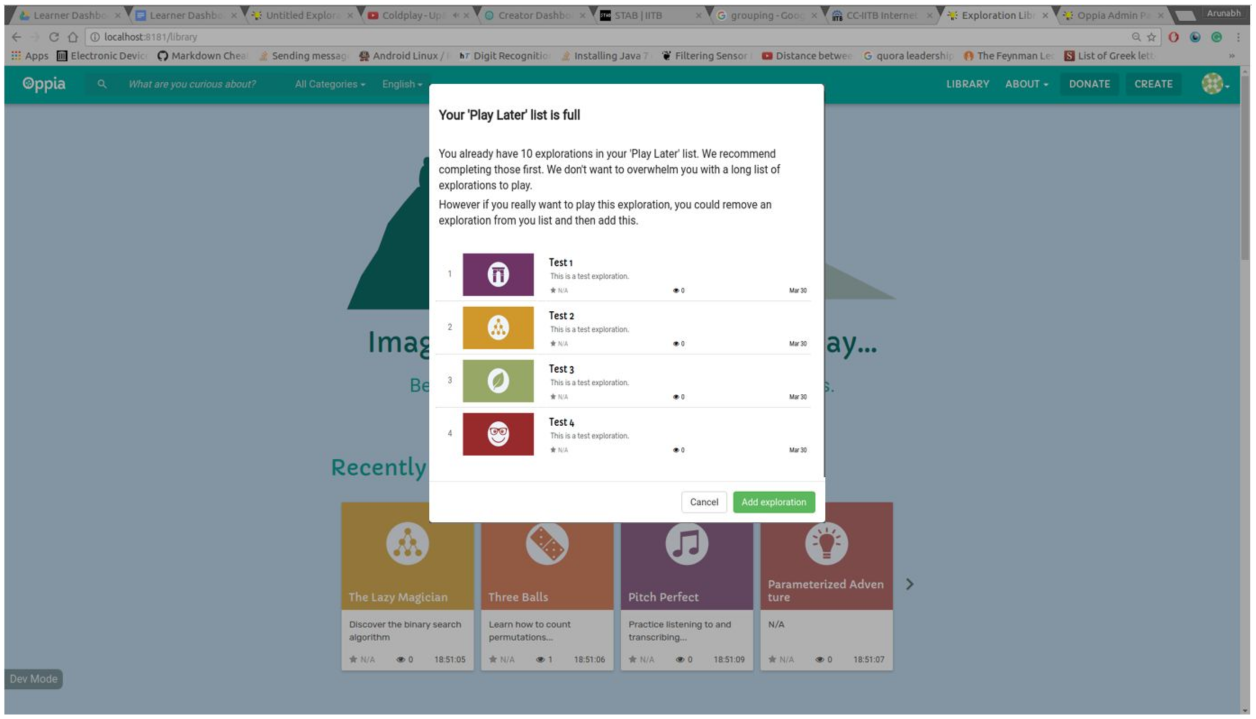
Your 'Play Later' list is full

You already have 10 explorations in your 'Play Later' list. We recommend completing those first. We don't want to overwhelm you with a long list of explorations to play.

However if you really want to play this exploration, you could remove an exploration from you list and then add this.

1	Test 1	This is a test exploration.	N/A · 0	Mar 30
2	Test 2	This is a test exploration.	N/A · 0	Mar 30
3	Test 3	This is a test exploration.	N/A · 0	Mar 30
4	Test 4	This is a test exploration.	N/A · 0	Mar 30

However, there may be cases when the learner really likes the exploration and wants to add the exploration to his/her play later list. In those cases, the learner may decide to remove a few (or one) explorations from his/her list to add this new exploration.



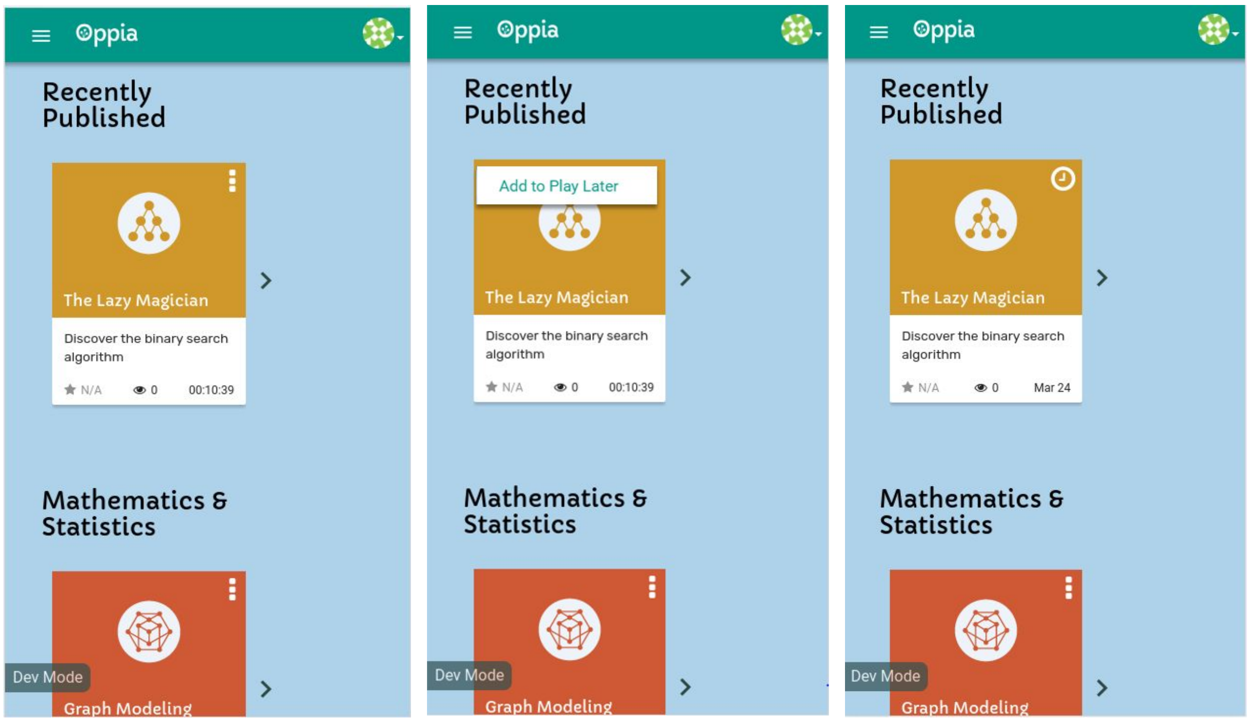
Your 'Play Later' list is full

You already have 10 explorations in your 'Play Later' list. We recommend completing those first. We don't want to overwhelm you with a long list of explorations to play.

However if you really want to play this exploration, you could remove an exploration from you list and then add this.

1	Test 1 This is a test exploration.	N/A	0	Mar 30
2	Test 2 This is a test exploration.	N/A	0	Mar 30
3	Test 3 This is a test exploration.	N/A	0	Mar 30
4	Test 4 This is a test exploration.	N/A	0	Mar 30

For mobile screen widths, the user can simply add the exploration to his/her 'Play Later' list by clicking the three dots on the top right corner and then selecting - 'Add to Play Later'.



We have now come to the end of the presentation of the design layout for learner dashboard. Given below is a brief explanation, of all the major decisions taken above. I hope this would give more concreteness to my proposal and make it clearer.

Explanations

1) What's the rationale for separating explorations and collections into different submenus (in Play Later list)?

I had conducted user interviews for this and here's what I found out. Most learners were okay with both decisions (together or not together), there were some learners (4 college students) who preferred the segregation of collections and explorations. Here's a brief summary of the reasons they mentioned -

- Learners think of collections and explorations as different entities. Learners said that they thought of collections as a long term commitment. So if a learner doesn't have much time, he/she wouldn't choose to start a collection because once he/she starts a collection, he/she wouldn't start any other exploration in between. So if the student is short of time he/she would just go through the explorations he/she had saved to play later. Later if the learner has time on his/her hands, he/she would go through the list of collections.
- When I showcased learners, collections together with explorations and asked them to compare with just collections and explorations, they said they liked the latter better. Putting collections together with explorations kind of spoilt the aesthetic appeal of the page, in the sense that the uniformity in the rows was broken.

This is why I decided it is best to show explorations separate from collections.

2) Can the learner move an exploration from the in progress section to the play later section?

I think we shouldn't give learners the option to move an exploration from the in progress section to the play later section. One of the reasons is that learners (3 students belonging to 8th and 9th grade) started complaining about the increasing complexity of the learner dashboard.
IMO, we should encourage learners to first finish the explorations in progress and so that the number of completed explorations goes up and learners feel like that they have accomplished a lot. So we shouldn't give learners the option to move explorations in progress to play later section. (which could encourage procrastination :P).
This is why there is no provision to move explorations from the in progress section to the play later section.

3) Why is the section titled 'Feedback Updates'?

In my user interviews, I found that 'Feedback Updates' gave the most accurate impression of the feature. Other options 'Messages' or simply 'Updates' gave a slightly misguided impression to the learner. The feedback threads are not exactly a conversation between the creator and the learner. They are more like an "issue/feature request" where a learner would like to receive regular updates on the action being taken by the creator. This is why 'Feedback Updates' would be a suitable title.

4) How did I decide on the 'Play Later' feature?

Here's what I want to accomplish - Learners should be able to save explorations and collections which they want to play, but don't have time at the moment. I had initially considered the name 'Bookmark' for this feature. Learners initially liked the name, as the explorations and collections gave the feel of a book and bookmark fitted right in. However, on digging deeper, I found that bookmark gave a misguided impression -- learners thought of this as favorites, which is not what I wanted to accomplish.
So we began thinking of suitable names for this feature. Someone suggested the name 'Play Later' inspired by 'Watch Later' feature of YouTube and we loved it. Because this name perfectly explained the feature to the learner - they could save explorations they would like to play later but don't have the time at the moment.

Hence the name -- Play Later.

5) On the reaching the dashboard, the first thing that learner sees is the explorations and collections in progress. Why is this?

We want the learner to start learning as quickly as possible from where he/she left off. The moment the learner reaches the dashboard, he/she knows what to do next. I think if we can achieve this, our learner dashboard would be successful. This why, on reaching the learner dashboard, we show the explorations and collections in progress. This is also why the next option on the side menu is -- Play later, as this is next on the learner's priority list.

We now move onto describing the technical implementation - as this is *what will make our dreams come true.*

Technical Implementation

Completed and partially completed explorations

Database

Two new models will be created **ActivitiesCompletedByLearnerModel** and **ExplorationsPartiallyCompletedByLearnerModel** in user/gae_models.py. The ActivitiesCompletedByLearnerModel will contain a field - completed_exploration_ids to store the ids of the exploration completed by the learner. ExplorationsPartiallyCompletedByLearnerModel will contain a field -- partially_completed_explorations. This field will be a list of dictionary objects having keys -- exploration_id, timestamp, state_name and version. timestamp -- time at which the learner left the exploration, state_name -- the name of the state at which learner left the exploration, version -- version of the exploration.

New Files

learner_progress_services.py - This file in /core/domain/, will contain services related to the progress of the learner such as adding an exploration to the list of completed explorations, getting all explorations partially completed by the learner etc.

Services

add_exploration_id_to_completed_list - This service, defined in learner_progress_services, will receive two fields, the user id, and exploration id. If the user is one of the contributors to the exploration, the exploration is not added to the list of completed explorations as a creator wouldn't like to see an exploration created by him/her in the learner dashboard. If the user is not a contributor, the exploration is added to the list of completed explorations. At the same time, the exploration is removed from the list of partially_completed_explorations and/or from the list of explorations to be played later (if present).
add_exploration_id_to_partially_completed_list - This service, defined in learner_progress_services, will receive five fields, the user id, exploration id, timestamp at which exploration was left, the state name and the version number. If the user is one of the contributors to the exploration or the exploration is already present in the list of completed explorations, the exploration is not added to the list of partially completed explorations. Otherwise, the exploration is added to the list of partially completed explorations and removed from the list of explorations to be played later (if present). If the exploration is already present only the timestamp, state_name and version number fields are updated.
remove_exploration_id_from_partially_completed_list - This service, defined in learner_progress_services, will receive two arguments -- user id and exploration id. It removes the entry corresponding to that exploration from the list of partially completed explorations.
get_all_completed_exploration_ids - This service, defined in learner_progress_services, receives a user id and returns a list of the exploration ids completed by the user.
get_all_partially_completed_exploration_ids - This service, defined in learner_progress_services, receives a user id and returns a list of the exploration ids partially completed by the user.

Controllers

Modify **ExplorationCompleteEventHandler** in reader.py to call add_exploration_id_to_completed_list to record that the user (if signed in) has successfully completed the exploration.
Similarly, we would modify **ExplorationMaybeLeaveHandler** in reader.py to call add_exploration_id_to_partially_completed_list to record that the user (if signed in) has partially completed the exploration.
RemoveExplorationFromPartiallyCompletedListHandler - This handler, in reader.py, would receive the id of the exploration as payload and then call the services of remove_exploration_id_from_partially_completed_list.

Completed and partially completed collections

Database

One more field will be added to **ActivitiesCompletedByLearnerModel** -- completed_collection_ids. A new model will be created - **CollectionsPartiallyCompletedByLearner**. It will have one field -- partially_completed_collection_ids to store the ids of the collections partially completed by the learner.

Services

add_collection_id_to_completed_list - This service, defined in learner_progress_services, is similar to add_exploration_id_to_completed_list. It takes in the collection id and user id. If the user is one of the contributors to the collection, the collection is not added to the list of completed collections. Otherwise, the collection is added to the list of collections completed by the user and removed from the list of incomplete and/or collections to be played later (if present).
add_collection_id_to_partially_completed_list - This service, defined in learner_progress_services, is similar to add_exploration_id_to_partially_completed_list. It takes in the collection id and the user id. If the user is one of the contributors to the collection or the collection is already present in the list of completed collections the collection is not added. Otherwise, the collection is added to the list of partially completed collections and removed from the list of collections to be played later (if present).
remove_collection_id_from_partially_completed_list - This service, defined in learner_progress_services, will receive two arguments -- user id and a collection id. It removes the entry corresponding to that collection from the list of partially completed collections.
get_all_completed_collection_ids - This service, defined in learner_progress_services, will receive a user id and return all the collections completed by the user.
get_all_partially_completed_collection_ids - This service defined in learner_progress_services, will receive a user id and return all the collections partially completed by the user.

Controllers

For this case, we will modify **ExplorationCompleteEventHandler** in reader.py. If the exploration that the learner has completed belongs to a collection we would first check using get_next_exploration_ids_to_complete_by_user if the learner has any additional explorations to complete in that collection. If there are no explorations left to complete, we will add the collection to the list of completed collections. Otherwise, the collection will be added to the list of partially completed collections.
Why have I not modified the ExplorationMaybeLeaveHandler handler?
This is because we consider a collection as being in progress only if the learner has completed at least one exploration. Because collections are long term commitments, adding it to the in progress section just after the user opens an exploration to play is not a very good idea. Once

the learner completes an exploration, the collection will remain in the in progress section until the learner completes it.
RemoveCollectionFromPartiallyCompletedListHandler - This handler, in reader.py, will receive the id of the collection as payload and then call the services of remove_collection_id_from_partially_completed_list.

Before we move on, I would like to discuss an important situation. What happens when an exploration is deleted after a learner completes it or has partially completed it. I have identified three different approaches to solve this problem. Below I'll analyze all the approaches, with respect to deleting an exploration after the user completes it. Similar arguments can be extended to other cases.
Red: Seriously bad Orange: Bad Yellow: Okay Green: Good

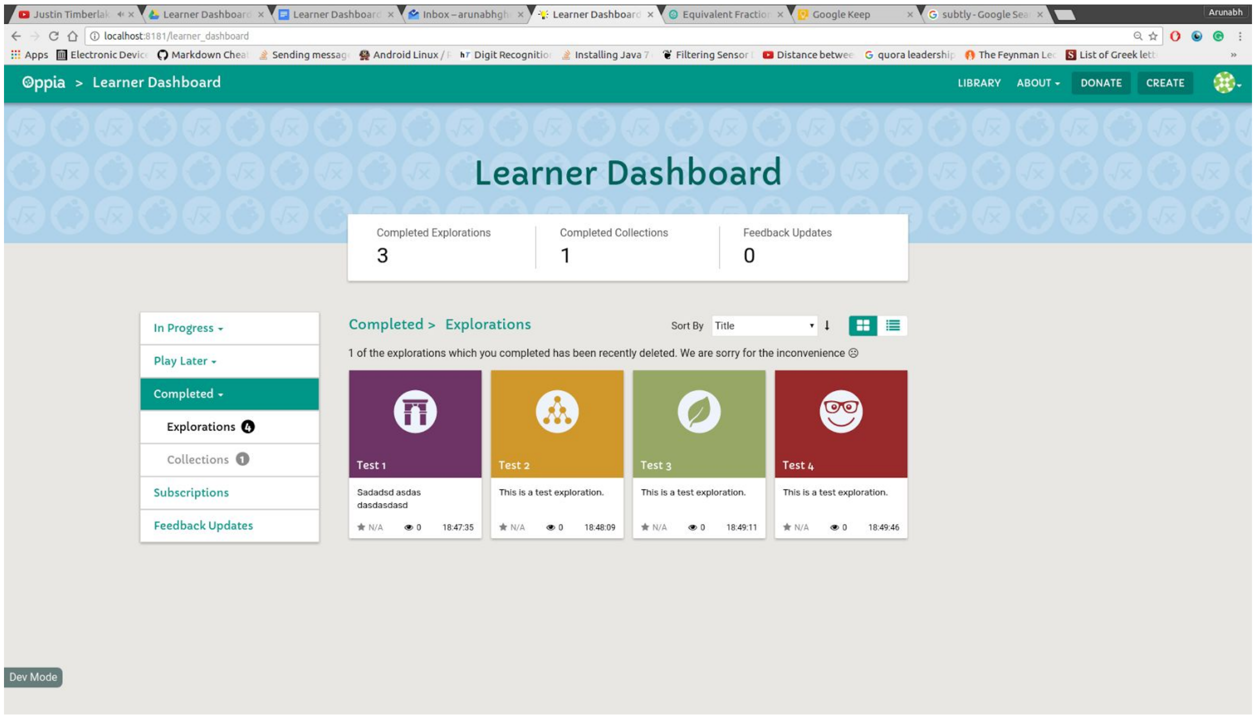
Approach	Add new fields to the exploration model which would store the ids of the users who have completed the exploration or are currently completing it. When an exploration is deleted, we iterate through all the users and delete the exploration if present.	On calling get_all_completed_exploration_ids or get_all_partially_completed_exploration_ids we check if the exploration exists by using the get_multi command. If the value returned is none, we make sure to remove the exploration from the respective list.	In the learner dashboard handler, when we fetch the exploration summaries of the explorations completed by using the get_exploration_summaries_matching_ids service in exp_services (which btw, also uses get_multi) if the value returned is none, we make sure to remove the exploration from the respective list.
Scalability	🟥 This approach suffers from serious scalability problems. The list of users completing an exploration keeps increasing with time. Some explorations are completed by more than 30K users. Iterating through this would take a long time. So this may break, horribly in production.	🟩 This approach is more scalable than the first one. Completing 30K explorations or even 1K explorations would take a long time :P. The get_multi function makes this operation faster.	🟩 This approach is also scalable. The number of explorations completed by the user will increase slowly, and we would anyway have to fetch all these explorations to display on the learner dashboard.
Speed of deleting an exploration	🟧 This approach would take a long time to delete an exploration. We would have to iterate through all the users and delete the exploration from their lists if present.	🟩 This approach won't increase the deletion time of an exploration. It won't have any effect. Clearly, a favorable result.	🟩 This approach won't increase the deletion time of an exploration. It won't have any effect. Clearly, a favorable result.
Speed of fetching entities from the lists.	🟩 This approach would have no effect on the time of fetching explorations completed by the learner.	🟨 This approach increases the time of fetching explorations as it will check if the exploration exists or not. Not a favorable result.	🟩 This approach won't increase the time required to fetch the explorations completed.

Time taken for the LearnerDashboard to load	This would add no extra cost.	As we would be calling the above function, the time for loading this page will increase as well.	This also would add no extra cost. We would anyway have to fetch the exploration summaries to display the explorations on the learner dashboard. An additional cost would be borne only if an exploration is deleted -- but this a rare event. For displaying the learner dashboard I'm very rarely expecting for more than one or two explorations to be deleted at a time.

From the above analysis, we can conclude that the best way to deal with deletion of entities is to follow the *third approach (the one on the extreme right)*. It performs better or as good as all the approaches in all cases.

Now how would this look from the user's perspective? Would they like to be notified? If yes, for what situations would they like to be notified?

From my user interviews, I found that users would like to be notified when an exploration/collection has been deleted. Should we then send all the users an email? Most users indicated that sending an email for each deleted exploration/collection would be a little too much. We can do better. Learners need this information when they reach the dashboard. If we do not show that an exploration has been deleted and they start looking for it -- it will lead to learner frustration. We send an email -- they might forget about it -- again leading to frustration. So it would be best to notify them when they reach the dashboard. Here's a mock which notifies the learner. The notification is neither too distracting, but it cannot be missed by learners.



1 of the explorations which you completed has been recently deleted. We are sorry for the inconvenience ☹

Test 1	Test 2	Test 3	Test 4
Sadadsd asdas dasdasdasd	This is a test exploration.	This is a test exploration.	This is a test exploration.
★ N/A 0 18:47:35	★ N/A 0 18:48:09	★ N/A 0 18:49:11	★ N/A 0 18:49:46

Another important situation - What happens when the creator adds a new exploration to a collection after the learner has completed it. There can be three approaches to deal with this situation, similar to the approaches above.

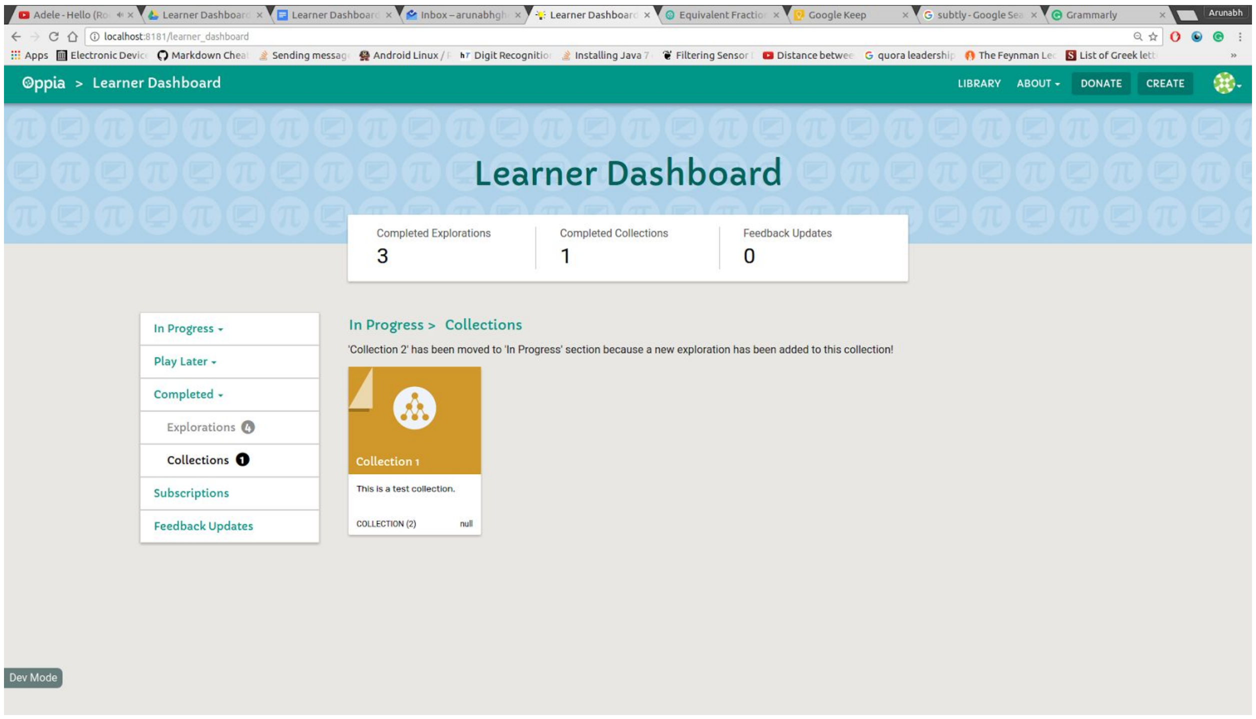
Red: Seriously bad   Orange: Bad   Yellow: Okay   Green: Good

Approach	Add new fields to the collection model. Whenever the creator adds a new exploration, we would iterate through all learners shifting the collection from completed to partially completed.	On calling get_all_completed_collection_ids we would check if a new exploration has been added by calling get_next_exploration_ids_to_complete_by_user. If yes, we will shift the collection from the completed section to the partially completed section.	In the learner dashboard handler, when we fetch the collections completed by the learner, we check if a new exploration has been added by get_next_exploration_ids_to_complete_by_user. If yes, we will shift the collection from the completed section to the partially completed section.
Scalability	This approach suffers from serious scalability problems. The list of users completing a collection keeps increasing with time. So this may break, horribly in production.	This approach is scalable. Completing a collection takes much longer than completing a collection and completing even 10 collections would take up a long time.	This approach is scalable. Completing a collection takes much longer than completing a collection and completing even 10 collections would take up a long time.
Speed of adding a new collection	This approach would take a long time to add a new collection. We would have to iterate through all the users and shift the collection from the completed section to incomplete section.	This approach won't increase the time of adding a new collection.	This approach won't increase the time of adding a new collection.
Speed of fetching completed collections	This approach would have no effect.	This would increase the time of fetching all the completed collections. This is because we will check if any new explorations have been added. If yes, the collection is shifted from the completed to incomplete section.	No Effect.
Time taken for the LearnerDashboard to	This would add no extra cost.	As we will be calling the above function, the time for loading the learner dashboard page will	This would increase the time for learner dashboard to load. This is because we will check if any new

load	🟩	increase as well.	explorations have been added. If yes, the collection is shifted from the completed to incomplete section.

From the above analysis, we see the third approach performs the best. *Therefore the third approach is chosen.*

Learners would be notified in a similar manner as described above. An appropriate explanation is provided which leaves the learner feeling cared for -- 'this shows that they (Oppia) care about me actually completing the collection. They could have just ignored the new exploration and I would feel happy that I had completed the collection. This is nice.' is what a user said during a user interview. Here's a mock -



In Progress > Collections

'Collection 2' has been moved to 'In Progress' section because a new exploration has been added to this collection!

Feedback

Database Design Analysis

Three different approaches for storing the messages read/unread by the user are considered below. These three approaches are analyzed in detail and compared with each other. The one which performs the best under all circumstances is chosen for this task

Red: Seriously bad   Orange: Bad   Yellow: Okay   Green: Good

Add a new field to FeedbackMessageModel which will store the user ids of the users who have read the	Add a new field to user model which will store the ids of the messages read by the user.	Create a new model FeedbackThreadUserModel which would be indexed by feedback thread id and user id.

message.		It would store the ids of the messages in that thread read by the user.
Action: Mark the messages of a feedback thread as being read.	When would we need this action? When a user clicks on a feedback thread we would like to mark all the messages in that thread as being read by the user.	
In this approach, we will start by fetching the model for the latest message and adding the user id to the list of users who have read the message (if not already present). We would do this for every message afterward until we hit a message as being already read by the user. We can see this approach is inefficient as we would have to fetch the model for each message and append the user to the read_by list.	In this approach, we will have to fetch the user model and append the ids of the messages which are currently not read by the user. We could start from the latest message, and stop at the message already read by the user. This method is also inefficient because, for every message, we would have to check if the message id is present in the list of messages read by the user -- which could be very very long! A user could easily read up to 10 to 15 messages in a single day. This is why this approach is not feasible.	In this approach, we would start by fetching or creating a FeedbackThreadUserModel. We will add the ids of the messages currently not read by the user to the list of messages read by the user. We will start with the latest message and stop at the message already read by the user. This is a very efficient way as it addresses both the shortcomings which the other two approaches had. Firstly, we would have to fetch only one model. Secondly, the length of the list would be as long as all the messages in the specific thread and most feedback threads are resolved within 5-10 messages. So checking if a message is already read by the learner would take no time at all.
Action: Find the messages unread by the user.	When would we need this action? When a learner reaches the learner dashboard, we need to display the feedback threads which contain unread messages. Although in the current proposal, we just need to find whether the last two messages are read by the learner, we consider the operation of finding all unread messages. This is because in the future we might want to know all the messages unread by the learner if want to implement new features like grouping of read messages (similar to Gmail).	
In this approach, we will iterate through all the feedback threads to which the learner is subscribed, and fetch the model for each message within the feedback thread. We will check if the learner id is within the list of users who have read the message and mark the message as unread, if not present. This method is very inefficient.	In this approach, we will first fetch the learner model. We will check if the message id is within the list of messages read by the user and if it is, we would mark the message as read. This method is also inefficient. We would have to check if the message is within the list of messages read by the user which can be a very very long list. Hence this is not feasible.	In this approach, we will first fetch the FeedbackThreadUserModel for each feedback thread to which the learner is subscribed. For a single feedback thread, we check if a message id lies within the list of messages read by the learner and mark the messages as read or unread. This is way more efficient than the other two approaches. The

We would have to fetch a model for each message which could be a long list of calls.		number of messages read by the user can be as high as the total number of messages in a feedback thread and most feedback is resolved within 5-10 messages. Therefore checking if a message is present or not is lighting fast.

Based on the analysis above, we conclude that the third approach is the way to go. *The third approach is efficient and better than the other two approaches in all cases.*

Database

A new model will be created - **FeedbackThreadUserModel** which will contain entries indexed by thread id and user id. It will have a field -- message_ids_read_by_user which would store the ids of the messages of that feedback thread read by the learner.

Services

update_messages_read_by_user - This service, defined in feedback_services.py, receives an exploration id, thread id, user id and a list of message ids. It adds the message ids to message_ids_read_by_user if not already present, thereby updating the messages read by the user.
get_all_threads_subscribed_to - This service, defined in subscription_services.py, receives the id of the user and returns all the thread ids to which the user is subscribed.
get_all_message_ids - This classmethod defined in FeedbackMessageModel, will receive an exploration id and thread id and return a list of all the message ids in that thread. One could raise the argument that this will go through the each of the models and hence would be inefficient. But we will use what is called a projection query to just fetch the message id property. Here's what the docs quote -

> Projection can be useful; if you only need two small properties each from several large entities, the fetch is more efficient since it gets and deserializes less data.

Later, we could find a way for the thread itself to keep track of the message ids thus making this function, even more, faster (see future plans).
get_thread_summaries - This service, defined in feedback_services.py, will receive a list of thread ids (without the exploration id part, just thread id) and a list of exploration ids to which the thread ids correspond to. It returns a dictionary corresponding to each entry in the list. The dictionary will contain the status, original author id, last message text, last updated, last_message_read (boolean), second_last_message_read (boolean), authors of the last two messages and the total number of messages. We can get the total number of messages using get_message_count function in FeedbackMessageModel. The information about the last two messages will be fetched using get_most_recent_messages. Whether or not the last two messages has been read by the user is determined by fetching the appropriate FeedbackThreadUserModel instance.
get_most_recent_message in FeedbackMessageModel will be renamed to **get_most_recent_messages**. It will have a new argument 'number_of_messages' to fetch

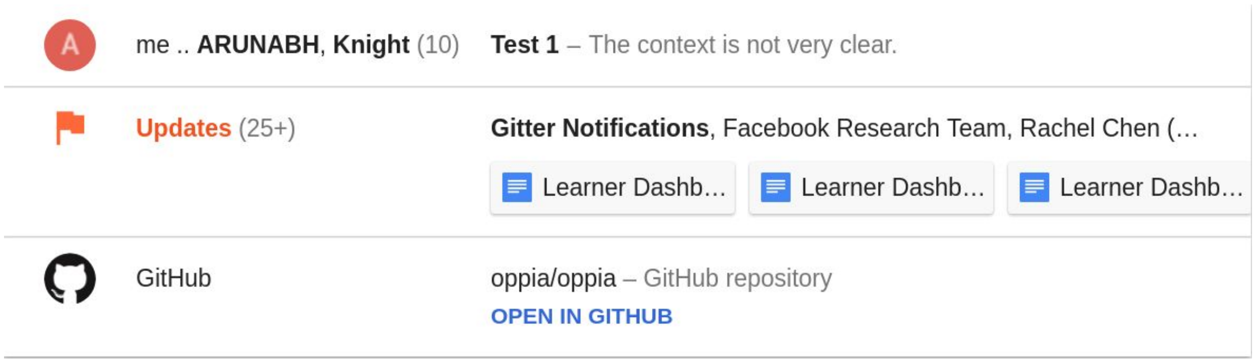
which will have a default value of 1. If we specify a value of number_of_messages it will return that many latest message models.
get_exploration_titles_by_ids - This service in exp_services.py will receive a list of exploration ids and return a list containing the titles of explorations provided. It will be used to fetch the exploration titles parameter of each of the feedback threads which will be added to the summary dict.

Controllers

LearnerDashboardFeedbackThreadHandler - This handler, defined in dashboard.py, receives two parameters - the exploration id and the thread id and returns all the messages in that thread. This will be called when the learner selects a particular feedback thread to view the messages. The update_messages_read_by_user is called to update the messages read by the user.
Modify **ThreadHandler** in feedback.py to update message_ids_read_by_user.

Behavior

When the user enters the feedback page for an exploration (the one on the editor page) or a specific feedback thread in 'Feedback Updates' section of the learner dashboard, we call the update_messages_read_by_user to update the messages read by the user. Would it be better to just show open feedback to the learner by default? I have answered this over here [3].
On the learner dashboard page, if the last two messages or the last message is not read by the learner, the text is bolded as shown in the mocks, which indicates to the learner that there are new messages in that thread. Is it sufficient to just check the status of the last two messages? Can't there be an "unread" block over a "read" block? I have answered it over here [1].
More specifically, in the feedback threads, we show the original author followed by the authors of the last two messages. The last two names or last name is bolded dependent on whether the learner has read the message. This behavior is similar to Gmail as shown below. -- the original author in the beginning followed by the authors of the last two messages (they are bolded as I've not read them).


me .. **ARUNABH**, **Knight** (10) **Test 1** – The context is not very clear.

Updates (25+) **Gitter Notifications**, Facebook Research Team, Rachel Chen (…
Learner Dashb… Learner Dashb… Learner Dashb…

GitHub oppia/oppia – GitHub repository
OPEN IN GITHUB

Play Later list for explorations and collections

Database

UserPlaylaterModel - This model will store the explorations and collections the learner wants to play but doesn't have time at the moment. It contains two fields - play_later_exploration_ids and play_later_collection_ids for explorations and collections respectively.

New Files

play_later_services.py - This file in /core/domain/, will contain services such as adding an exploration to the play later list, getting all the explorations the learner wants to play later etc.
play_later.py - This file in /core/controllers/, will contain handlers to handle tasks like adding a collection to the play later list, removing an exploration from the list of explorations to play later.

Services

add_exploration_id_to_play_later - This service, defined in play_later_services. receives a user id, an exploration id and an index value (whose default value is none). If the value of index is none, the exploration id is added to the end of the list. If it is not none, the exploration id is first removed from the list and then inserted at the desired position. However, if the exploration id is already present in completed or in progress list it is not added. If the length of the list exceeds the maximum limit, the exploration is not added.
remove_exploration_id_from_play_later - This service, defined in play_later_services, receives a user id and an exploration id. The exploration id is removed from the list (if present).
add_collection_id_to_play_later - This service, defined in play_later_services. receives a user id, a collection id and an index value (whose default value is none). If the value of index if none, the collection id is added to the end of the list. If it is not none, the collection id is first removed from the list and then inserted at the desired position. However, if the collection id is already present in completed or incomplete list it is not added. If the length of the list exceeds the maximum limit, the collection is not added.
remove_collection_id_from_play_later - This service, defined in play_later_services, receives a user id and a collection id. The collection id is removed from the list (if present).
get_all_explorations_to_play_later - This service, defined in play_later_services, receives a user id and returns all the exploration ids which the learner wants to play later.
get_all_collections_to_play_later - This service, defined in play_later_services, receives a user id and returns all the collection ids which the learner wants to play later.

Controllers

AddExplorationToPlayLaterHandler - This handler, defined in play_later, receives an exploration id and an index. It then calls the services of add_exploration_id_to_play_later, to add the exploration to the play later list.
RemoveExplorationFromPlayLaterHandler - This handler, defined in play_later, receives an exploration id and then calls the services of remove_exploration_id_from_play_later to remove the exploration from the play later list.

AddCollectionToPlayLaterHandler - This handler, defined in play_later, receives a collection id and an index. It then calls the services of add_collection_id_to_play_later, to add the collection to the play later list.
RemoveCollectionFromPlayLaterHandler - This handler, defined in play_later, receives a collection id and then calls the services of remove_collection_id_from_play_later to remove the collection from the play later list.

Behavior

When the learner makes a new change in the Play Later list by dragging an exploration/collection around or deleting an exploration an appropriate request is sent which calls the appropriate handler. For example - when the learner shifts an exploration to a new position a request is sent to AddExplorationToPlayLaterHandler which receives the exploration id and the new position as payload and then calls the services of add_exploration_id_to_play_later to update the position of the exploration.

Learner Dashboard Page

New Files

learner_dashboard.html - This will be our learner dashboard page. The layout of this page will be the same as the mocks presented above.
LearnerDashboardBackendApiService.js - This service will retrieve all the information for the learner dashboard.
LearnerDashboard.js - This will be the controller for the Learner Dashboard. This will complement learner_dashboard.html and make our learner dashboard fully functional.

Controllers

LearnerDashboardPage - This handler, defined in dashboard.py, redirects /learner_dashboard to the standard learner dashboard page if the user is signed in. If an unsigned user clicks the 'sign in to save your progress' button we also pass the id of exploration he/she just completed as payload. This handler adds the exploration to the list of completed explorations or adds the collection to the in progress section (if the exploration belongs to a collection).
LearnerDashboardHandler - This handler, defined in dashboard.py, is responsible for providing data to the learner dashboard. Here is the summary of the data and how it will gather this data -

- The list of explorations and collections completed by the learner will be fetched using get_all_completed_exploration_ids and get_all_completed_collection_ids services respectively. Similarly, we would fetch the explorations and collections partially completed by the user.
- The explorations and collections saved by the learner will be fetched using get_all_explorations_to_play_later and get_all_collections_to_play_later services.
- The list of feedback threads to which the learner is subscribed is fetched using get_all_threads_subscribed_to service. A basic summary of the feedback threads to be displayed on the learner dashboard is obtained using get_thread_summaries service.

- The creators to which the learner is subscribed is obtained using get_all_creators_subscribed_to service in subscription_services.py.
- While fetching the summaries, if any of the values returned is none, we will know if the exploration/collection has been deleted. Similarly, we will know if a new exploration has been added to a completed collection using get_next_exploration_ids_to_complete_by_user. We will update all the lists, and display the changes as notifications on the learner dashboard as seen in the above mocks.

Other general site changes

- Add Learner Dashboard to the navigation bar dropdown by modifying top_navigation_bar_directive.html.
- The splash page will be made the home page for all Oppia users. From there, they can proceed to the creator dashboard or learner dashboard. This decision is made considering the fact that learners can also be creators. The complete list of modifications to the splash page is specified above in the mocks.

Explanations

1) Is it possible for an "unread" block to have a "read" block following it? If so, simply checking the status of the last two messages would not be enough?

Nope, there cannot be an "unread" block before a "read" block. This is because we mark all messages as read by the user once he/she opens up a feedback thread. This is generally followed by standard platforms like WhatsApp and Gmail threads. If I open a thread, all messages are marked as read by me.
However, one notable exception is Gitter - we can have "unread" blocks over "read" blocks. But I would like to mention that this has not been found convenient by a lot of users as it kinds of forces them to scroll through each message even though they may not want to read them. If they don't do this, the unread messages keep accumulating over time.
Therefore I feel, it's best to keep this behavior simple and mark all messages as read once a user opens a thread.

2) If a learner is halfway through an exploration inside a collection, would that exploration appear in the Explorations tab?

Nope. We deal with collections and explorations separately so that there is no duplication of information. In the exploration tab, we just deal with stand alone explorations which are not linked to any collection. In the collection tab, we show collections which are currently being completed by the learner. If the learner is halfway through an exploration in a collection, we display the collection as being in progress. When the learner clicks the collection he/she is able to view his/her progress in the form of footsteps (which is already implemented) and then decide which exploration to play next.

3) Would it be better to show only open feedback threads to the learner by default?

If we are looking from the point of a creator, this would be the right way to go. The creator should have his/her focus on open feedback threads.

From the point of view of a learner, I found that a learner likes to see the feedback thread that has been fixed, or not actionable or ignored by the creator. If the learner sees that the feedback thread has been marked as not actionable he/she would click it to see the reason. In short, we are making it easier for the learner to navigate and ensuring that all feedback is properly addressed.

Milestones

Milestone 1.1 (2 weeks)

- Summary

  In this milestone, database models are created for tracking the progress of the learner. Explorations and collections partially completed or completed by the learner are stored in the database and services are created for fetching and interacting with the data. For example -- we have a service which fetches all the explorations completed by the learner which we can use to display explorations completed by the learner on the dashboard, a service for adding an exploration to the list of explorations completed by the learner, which we can call whenever the learner completes a new exploration.

- Code
  - Add three new models -- ActivitiesCompletedByLearnerModel, ExplorationsPartiallyCompletedByLearnerModel and CollectionsPartiallyCompletedByLearnerModel in user/gae_models.py.
  - Add two fields to ActivitiesCompletedByLearnerModel -- completed_exploration_ids and completed_collection_ids.
  - Add one field to ExplorationsPartiallyCompletedByLearnerModel -- partially_completed_explorations which is a dictionary containing the keys -- exploration_id, timestamp, state_name and version.
  - Add one field to CollectionsPartiallyCompletedByLearner -- partially_completed_collection_ids.
  - Add service add_exploration_id_to_completed_list to learner_progress_services.py.
  - Add service add_exploration_id_to_partially_completed_list to learner_progress_services.py.
  - Add service get_all_completed_exploration_ids to learner_progress_services.py.
  - Add service get_all_partially_completed_exploration_ids to learner_progress_services.py.
  - Add service add_collection_id_to_completed_list to learner_progress_services.py.
  - Add service add_collection_id_to_partially_completed_list to learner_progress_services.py.
  - Add service get_all_completed_collection_ids to learner_progress_services.py.
  - Add service get_all_partially_completed_collection_ids to learner_progress_services.py.

    - Update ExplorationCompleteEventHandler and ExplorationMaybeLeaveHandler in reader.py to update completed and partially completed explorations and collections.
    - Add RemoveCollectionFromPartiallyCompletedListHandler and RemoveExplorationFromPartiallyCompletedListHandler to reader.py.
- Tests
    - Test that add_exploration_id_to_completed_list works as expected - it adds the exploration id to the completed_exploration_ids field of the user. At the same time, if the exploration is present in partially_completed_explorations, it is removed from there.
    - Test that add_exploration_id_to_partially_completed_list works as expected - it adds the exploration to the partially_completed_explorations field of the user.
    - Test that get_all_completed_exploration_ids returns the correct list of completed explorations.
    - Test that get_all_partially_completed_exploration_ids returns the correct list of incomplete explorations.
    - Test that add_collection_id_to_completed_list correctly adds the collection to the list of completed collections. If the collection is also present in partially completed collections list, it is removed.
    - Test that add_collection_id_to_partially_completed_list adds the collection to the list of collections partially completed by the learner.
    - Test that get_all_completed_collection_ids correctly returns the list of collections completed by the user.
    - Test that get_all_partially_completed_collection_ids correctly returns the list of collections partially completed by the user.
    - Test that RemoveCollectionFromPartiallyCompletedListHandler removes the collection from the partially completed list.
    - Test that RemoveExplorationFromPartiallyCompletedListHandler removes the exploration from the partially completed list.

Milestone 1.2 (2 weeks)

- Summary

  In this milestone, our first ever learner dashboard will be released to all users. The learner dashboard will display the explorations/collection completed by the learner, explorations/collections in progress and the creators to which the learner has subscribed.

- Code
    - Backend
        - Add LearnerDashboardBackendApiService.js to fetch Learner Dashboard data.
        - Add LearnerDashboardPage handler to dashboard.py.

        - Add a controller LearnerDashboardHandler to dashboard.py. This will fetch all the data required for the learner dashboard page.
        - Remove the code in PreferencesHandler in profile.py for displaying the subscriptions page. Since we are showing the subscriptions on the learner dashboard, we should remove it from the preferences page.
    - Frontend
        - Create the first time ever Learner Dashboard page! This will be accomplished by creating LearnerDashboard.js and learner_dashboard.html.
        - Remove the subscriptions field from preferences page by removing the relevant code from preferences.html.
        - Add Learner Dashboard to the navigation bar dropdown by modifying top_navigation_bar_directive.html.
        - Modify the splash page to include learner dashboard.

- Tests
        - Create a new file - learner_dashboard_test.py. This will contain a class called LearnerDashboardHandlerTest in which we would test the validity of the data returned by LearnerDashboardHandler.
        - An e2e test is created to test this brand new Learner Dashboard. A learner is created, a few explorations are created, the learner plays some completely, some halfway through. He/she subscribes to a few creators. On navigating through the learner dashboard, the learner is able to see the explorations he/she completed, the explorations which he/she left halfway through and creators to which he/she has subscribed.

Milestone 2.1 (1 week)

- Summary

  In this milestone, database model is created for tracking the messages read/unread by the learner. We now know, to which feedback threads the learner is subscribed and if there are any new messages in those threads which the learner has not read. Services are created for fetching and interacting with this data. For example -- a service is created for fetching all the threads to which the learner is subscribed, which we can use to display the feedback threads on the learner dashboard.

- Code
    - Add a new model to feedback/gae_models.py -- FeedbackThreadUserModel
    - Add new field to FeedbackThreadUserModel -- message_ids_read_by_user.
    - Add service get_all_threads_subscribed_to to subscriptions_services.py.
    - Add service get_thread_summaries to feedback_services.py.
    - Add service get_exploration_titles_by_ids to exp_services.py.
    - Add get_all_message_ids to FeedbackMessageModel.
    - Add service update_messages_read_by_user to feedback_services.py.

    - Add handler LearnerDashboardFeedbackThreadHandler to dashboard.py.

- Tests
    - Test that update_messages_read_by_user correctly updates the messages read by the user.
    - Test that get_all_threads_subscribed_to correctly returns all the threads to which the learner is subscribed.
    - Test that get_thread_summaries correctly returns the summaries of the threads ids submitted to it.
    - Test that get_exploration_titles_by_ids correctly returns the titles of the explorations.
    - Test that get_all_message_ids returns the correct list of message ids.
    - Test that LearnerDashboardFeedbackThreadHandler correctly returns the messages of the given feedback thread.

Milestone 2.2 (a week and a half)

- Summary

  In this milestone, a new feature is released to all our users. Learners are now able to view all the feedback threads in which he/she is involved. Learners can even respond to any of the feedback threads from their dashboard itself.

- Code
    - Backend
        - Modify LearnerDashboardHandler to return the summaries of the feedback threads to which the learner is subscribed.
    - Frontend
        - Modify LearnerDashboard.js and learner_dashboard.html to include feedback threads. The learner will be able to view the feedback threads to which he/she is subscribed and reply to those feedback threads using addNewMessage service in threadDataService.js

- Tests
        - Test that LearnerDashboardHandler correctly returns the summaries of the feedback threads to which the learner is subscribed.
        - Modify the e2e tests to tests the feedback feature. Tests that the learner is able to view the feedback threads to which he/she is subscribed and on clicking a specific feedback thread he/she is able to view the messages in it.

User Testing period (half week)

The new dashboard is thoroughly user tested. User feedback is noted and analyzed. Appropriate action is then taken to incorporate this into our new learner dashboard to make it even more amazing and loved by learners!

Milestone 2.3 (a week and a half)

- Summary

  In this milestone, database models are created for storing all the explorations and collections which the learner wants to play later. Services are created for fetching and interacting this data. For example -- we now have a service which we can use to fetch all the explorations which the learner wants to play later, which we can use to display on the learner dashboard.

- Code
  - Add a new model UserPlaylaterModel to store explorations and collections to be played later by the learner.
  - Add two fields to UserPlaylaterModel - play_later_exploration_ids and play_later_collection_ids.
  - Add service add_exploration_ids_to_play_later to play_later_services.py.
  - Add service remove_exploration_ids_from_play_later to play_later_services.py.
  - Add service add_collection_ids_to_play_later to play_later_services.py.
  - Add service remove_collection_ids_from_play_later to play_later_services.py.
  - Add service get_all_explorations_to_play_later to play_later_services.py.
  - Add service get_all_collections_to_play_later to play_later_services.py.

- Tests
  - Test that add_exploration_ids_to_play_later correctly adds the explorations to the Play Later list.
  - Test that remove_exploration_ids_from_play_later removes the explorations from the Play Later list.
  - Test that add_collection_ids_to_play_later correctly adds the collections to the Play Later list.
  - Test that remove_collection_ids_from_play_later removes the collections from the Play Later list.
  - Test that get_all_explorations_to_play_later returns all the explorations to be played later by the learner.
  - Test that get_all_collections_to_play_later returns all the collections to be played later by the learner.

Milestone 3.1 (1 week)

- Summary

  In this milestone, functions are created to help us execute user actions. We now have functions for each of the user actions such as - adding an exploration, removing an exploration, adding a collection which we can call to execute user actions. For example, if the learner wants to add an exploration to his/her 'Play Later' list we can just call a function - AddExplorationToPlayLaterHandler to add the exploration.

- Code
  - Modify LearnerDashboardHandler to return the list of explorations and collections to be played later by the learner.
  - Add AddExplorationToPlayLaterHandler to play_later.py.
  - Add RemoveExplorationFromPlayLaterHandler to play_later.py.
  - Add AddCollectionToPlayLaterHandler to play_later.py.
  - Add RemoveCollectionFromPlayLaterHandler to play_later.py.
  - Modify previous services to account for Play later list. For example, add_exploration_id_to_completed_list will remove the exploration from the play later list, if present. For more details refer to the technical implementation above.

- Tests
  - Test that LearnerDashboardHandler correctly returns the list of explorations and collections the learner wants to play later.
  - Test that AddExplorationToPlayLaterHandler adds the exploration to the Play Later list.
  - Test that RemoveExplorationFromPlayLaterHandler removes the exploration from the Play Later list.
  - Test that AddCollectionToPlayLaterHandler adds the collection to the Play Later list.
  - Test that RemoveCollectionFromPlayLaterHandler correctly removes the collections from the Play Later list.
  - Modify the tests of the services modified to account for Play Later feature.

Milestone 3.2 (a week and a half)

- Summary

  In this milestone, a new feature will be rolled out to all our users. A learner will now be able to add explorations/collections to his/her play later list and even rearrange them in the order he/she wants to play them.

- Code
  - Frontend

        - Modify LearnerDashboard.js and learner_dashboard.html to implement Play Later functionality. The learner is now able to view the explorations and collections he/she wants to play but doesn't have time at the moment.
        - Modify exploration_summary_tile_directive.html and collection_summary_tile_directive.html to enable Play Later functionality. Implement the icons on the top right corner as seen in the mocks.

- Tests
    - Modify the e2e tests for learner dashboard. The learner will save a few explorations and collections to play later and then head over to the Learner dashboard. The learner is able to view all the explorations and collections saved by him/her and can even rearrange the explorations and collections according to their preference.

Timeline

The deadlines outlined here is the worst case timeline - by these dates, the given goals will definitely be completed. Although the formal date to start coding is May 30th, I will be able to start by end of April itself as I already have a very good bonding with the community. So during May, along with helping other students get along with Oppia, I will get a head start on this project and ensure that the goals are completed well before time.
Another advantage I have is that -- *It wouldn't take me much time to implement the mocks shown above as they are created using HTML and CSS -- I wouldn't have to convert designs to code, they are already converted. This would ensure faster completion of goals. We would have enough time to take in more user feedback and modify the dashboard to make it even more amazing.*

- 1st two milestones will be completed by the end of June. Which means that a basic learner dashboard site will be launched by June end.
- The next three milestones will be completed by July end. Which means that by the end of July, a learner will be able to see the feedback threads they are subscribed to and reply to them directly from their dashboard.
- During August, the last two milestones will be completed. Learners will now be able to add explorations and collections to their Play Later list if they don't have the time to play the lesson at the moment.

Summer Plans

Which timezone(s) will you primarily be in during the summer?

My timezone will be Indian Standard Time (IST) with a time offset of UTC+05:30.

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

Since the project schedule coincides with my summer vacation, I will be free from any other commitments (especially academic commitments).

Until July I have no commitments and will be able to work for 11-15 hours each day. My typical working hours will be from 10 am to 1 am (IST). I will definitely take small breaks in between those 11-15 hours if I can, as I know balance is important.
After July, my college will commence but initially, things would be light. I will be able to give 8 to 10 hours each day. Typical working hours will be from 5 pm to 2 am (IST).

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

Until July I have no commitments other than Oppia. In the month of August, my college will reopen but being the semester beginning, the workload would be light and I will be able to manage this project along with my academics. Apart from this, I have no other commitments.

Communication

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

I am active on Oppia's gitter channel and regularly participate in the discussions. Communicating over email is also perfectly fine.
Github Handle - Arunabh98.
Email Address - arunabhghosh98@gmail.com

How often do you plan on communicating with your mentor?

I plan to send weekly updates to my mentor informing them how much I have completed and how much I plan to complete in the following week. In addition to this, a document will be maintained which will contain how much I have accomplished each day. Also, I will be connected via gitter and email to discuss anything if needed.

Future Plans

There are a lot of ways we can make our learner dashboard even more amazing. more powerful. Because of the constraint of completing the project in 12 weeks, there was a limit on the number of features I could implement. But nothing stops us from taking the dashboard forward after summers. Here are a few ways in which I think we can improve the learner dashboard and help the learner learn more effectively.

Start explorations from where learners left off

In the current implementation, even though we save the explorations and collections in progress learners still have to begin those explorations from the start. While this is good in certain cases -- learners returning to an exploration after a long time, would not remember most of the content, so it would be better to start from the beginning. However in certain cases -- there is a need to start the exploration from in between. Say the learner starts an exploration during the day and returns to complete it by evening. To allow for a seamless learner integration, he/she should be able to start from where he/she left off.

The foundations for implementing this feature has been proposed in this proposal. We store information like the time at which the learner left the exploration, the name of the state at which he/she left the exploration based on which we can decide whether or not to start the exploration from where he/she left off.

Allow non-signed in users to try out the learner dashboard

We allow the non-signed in users to use the learner dashboard which would store data only for that session. This would give them a taste of the powerful features the learner dashboard has to offer and give them more incentive to sign in and start a long term learning relationship with Oppia.

Show learners which explorations have been deleted

Currently, if a learner completes an exploration and it gets deleted by the creator or moderator we just tell the learner that an exploration that you completed has been deleted and we are sorry for the inconvenience caused. It would be better if we could show the name of the exploration deleted in case the learner starts to wonder.

(Technical) Thread should keep track of the ids of the messages

Currently, we fetch the ids of the messages by querying and queries are expensive. In fact, currently, this approach is used for every task -- like fetching message models, getting message counts. A bit of research needs to be done, on why we are doing it this way and if we can change it. If we can change it, it would definitely be a big improvement for the dashboard and make it much faster.