GSoC 2021 Proposal - Oppia

Improve Exploration Save Flow

Aryaman Gupta

Table of contents

About You

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

I got to know about Oppia from my seniors in my college when I asked them to suggest some open source organizations as I was new to opensource. So I simply filled the CLA and started contributing. Still, when I got to know more about Oppia, I was fascinated with its motive: to provide high-quality education to those who lack access to it. This motivated me a lot as I always thought of working with some non-profit organizations to help the person in need, so here I got a way of helping them that is best suited for me.

Secondly, I have always been inspired by Oppia's team, which works so hard to make Oppia a better platform, its friendly and helpful mentors, and it's great work culture that can not be found in any other organization.

Project Chosen: Improve Exploration save flow: Syncing edits in the background

I chose this project because I wanted to work on something which is new to the codebase and which can be really helpful for its users. We also often get stuck because of bad network connection, and if our progress or changes lose, it can be really frustrating. So this project will solve that problem where the changes will be saved in case of a bad network connection and will be merged once a reliable network source is found. Secondly, it will also enable real-time collaboration effectively, which is quite an interesting feature to implement in the codebase. Also, this feature will help many creators work simultaneously, which will save a lot of time, and then the changes will be easily done.

Prior experience

I have been doing web development since last year. I have been coding in javascript and python. I have experience with frontend frameworks like Angular, VueJs, React, Flutter and have also worked with backend frameworks like DRF, Django, etc. I have worked on a few college projects also and I have a prior experience with problem-solving and DSA. You can visit my **Github Profile**.

Apart from this, I have been contributing to Oppia for the past three months and have been working with the LaCE and Migration team simultaneously. In this short period of time, I have become quite familiar with the codebase.

Some of My PR's are:

- **Migrated Fatigue Detection Service**
- **Added a unit test for Graph Utils Service**
- **Audit Job to check for explorations with number of ratio terms greater than 10**
- **Cover Few Services with strict checks**

This is a complete list of all my PRs.
This is a complete list of all the issues I opened.

Apart from the issues I fixed in the PRs, I also worked on finding what goes in angular-html-bind for many directives and components using an instance of $compile, which gave me a lot of knowledge of the codebase and its different components and services.

Contact info and timezone(s)

- **Contact:**
  - Email: aryaman.gupta.met19@itbhu.ac.in
  - Phone No.: (+91) 7355929950

  I am fine with any form of communication.
- **TimeZone:** Indian Standard Time (GMT+5:30)
  But I will still be available at any time zone that you guys prefer.

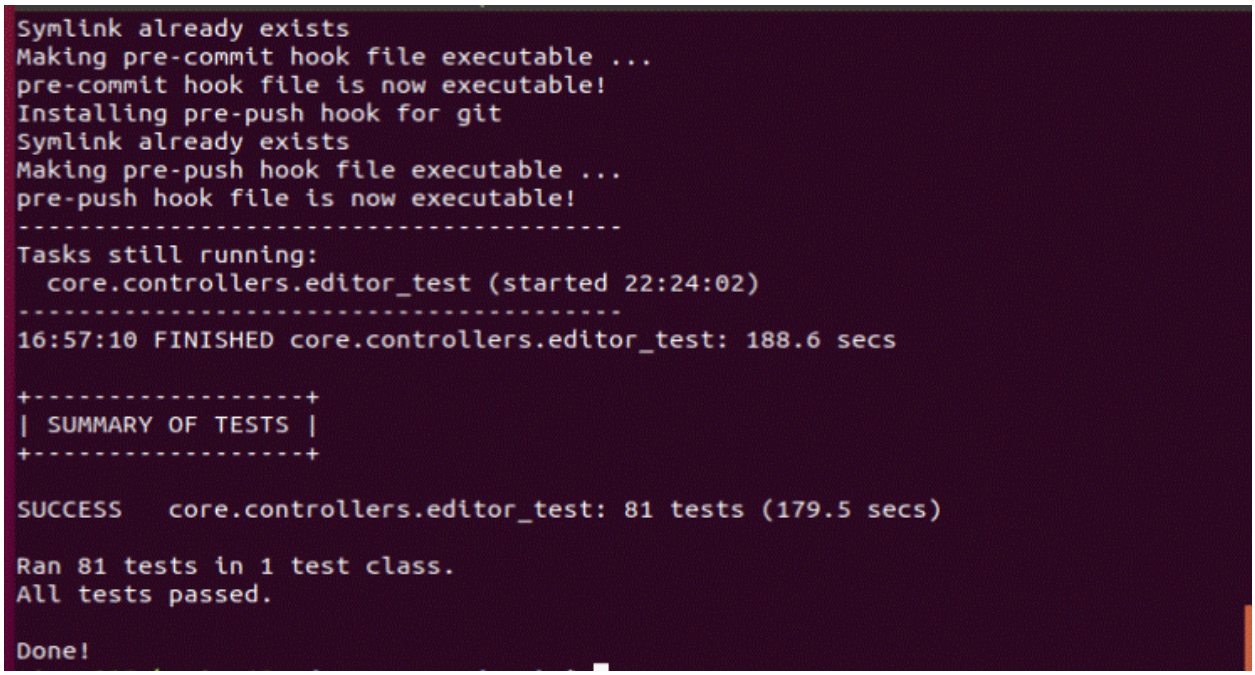
Time commitment

- I would be working throughout the GSoC Period time from 7th June to 16th August (10 week period).
- I will commit approx 4 hrs per day and approx 30 hrs per week during the GSoC period.
  Actually, my college is teaching in an online mode right now but it may happen that college reopens in august starting (most probably will be online during the GSoC period due to the increasing covid cases), so in that case, before going to college, I will increase my working hours to approx 5-6 hours per day and will be working 2-3 hours per day from college.

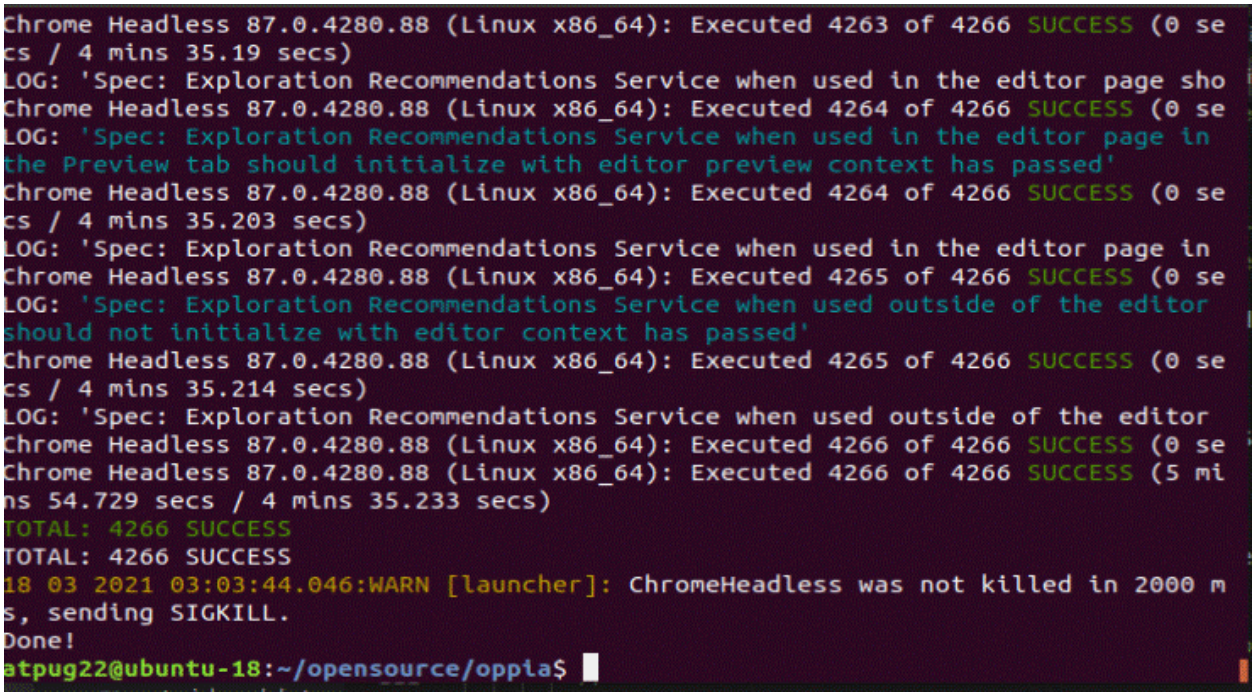
Essential Prerequisites

Answer the following questions:

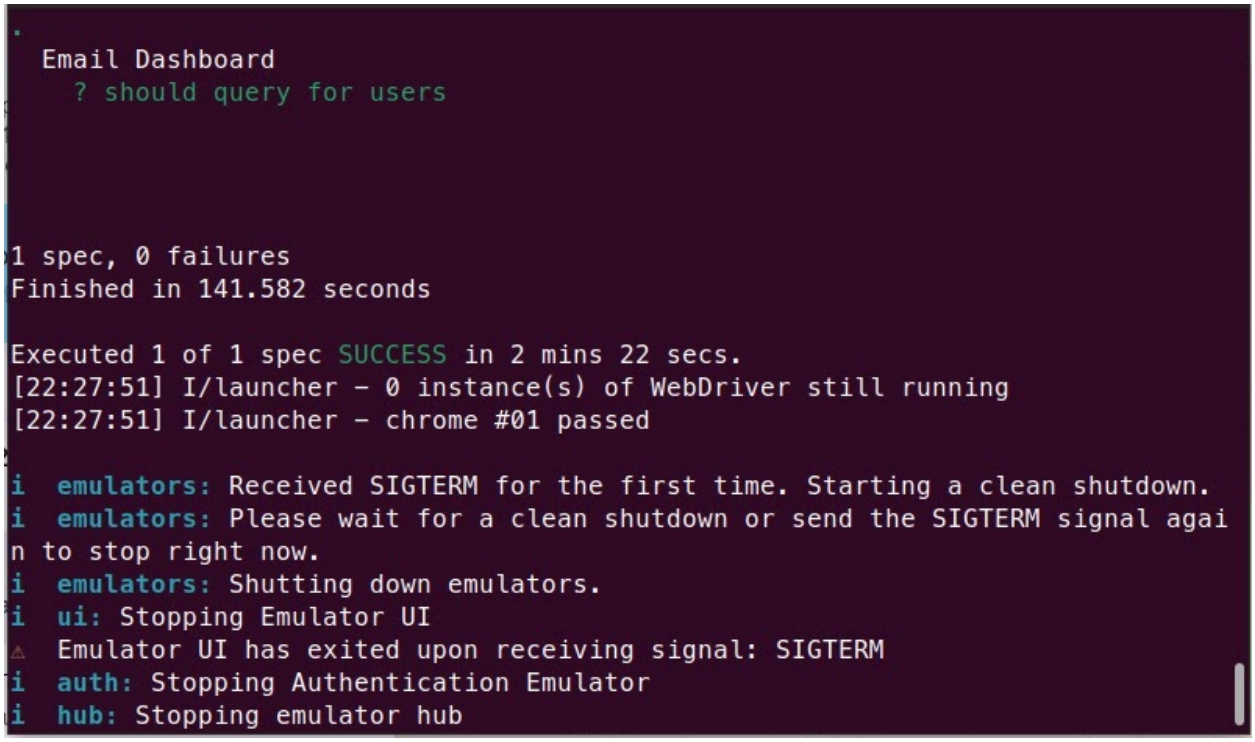
- *I am able to run a single backend test target on my machine. (Show a screenshot of a successful test.)*



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



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



Other summer obligations

I don't have any other commitments this summer except if college reopens, then it may take 1-2 days shifting to college, but I make sure that I'll cover up those days sometime else.

Communication channels

I am planning to communicate through hangouts and email. Also, I am comfortable with any other channel that the mentor prefers. And I will be able to communicate with the mentors almost daily and give my updates on the project.

Application to multiple orgs

No, I am not applying to any other org.

Project Details

Product Design

This feature's users will be mainly the exploration creators and later can be used for topics, skills, and lesson creators.
This project is divided into two parts, so I'll try to explain the two parts separately.

First Part:

Explorations are interactive activities that try to recreate one-on-one tutoring experience. These explorations may have multiple managers and collaborators.
So a situation may arise when two creators (managers or collaborators) at the same time try to change some content in various states. So now, let's see what happens in such situations.
So let us take two users, A and B.
Presently, if A changes the content and saves a draft and publishes it, and then if B changes some content and saves a draft and tries to publish it, it doesn't publish. When he reloads the screen, it shows some dialog box telling him to discard all the changes as the version is changed because the changes are made by some other user (shown in the image below).

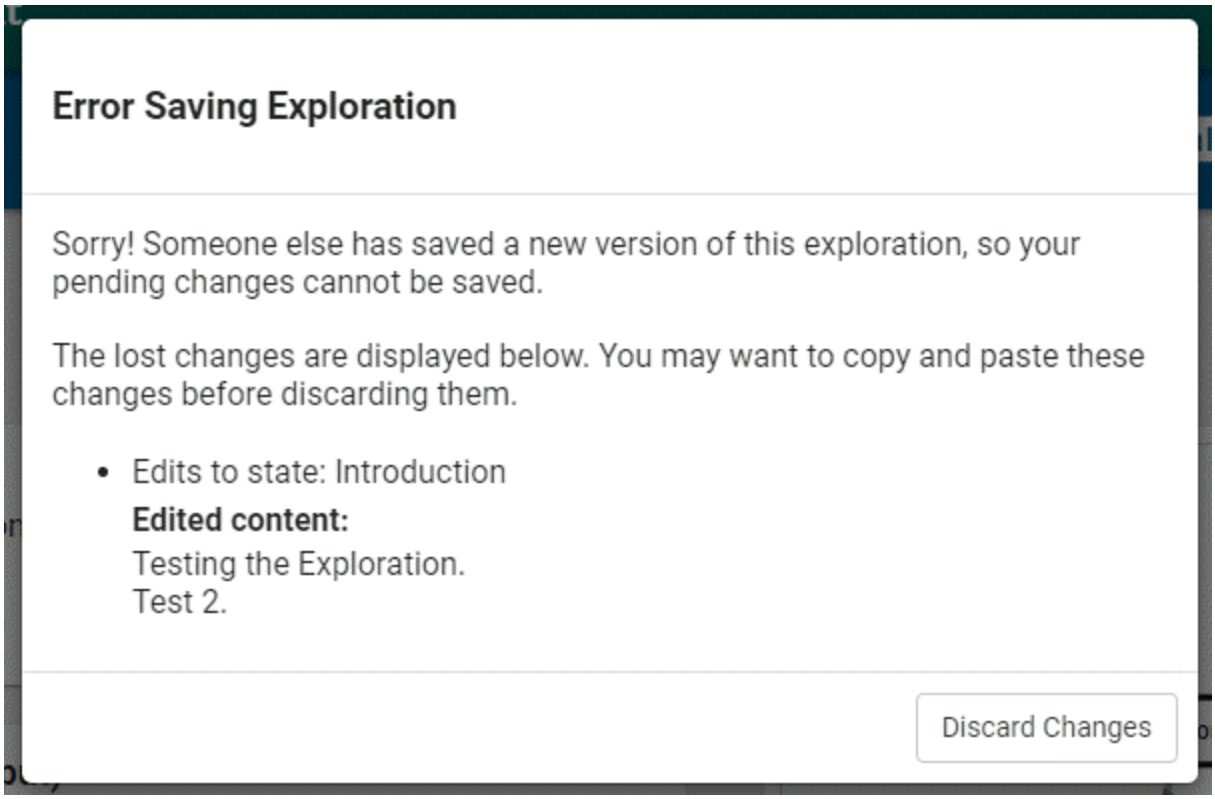


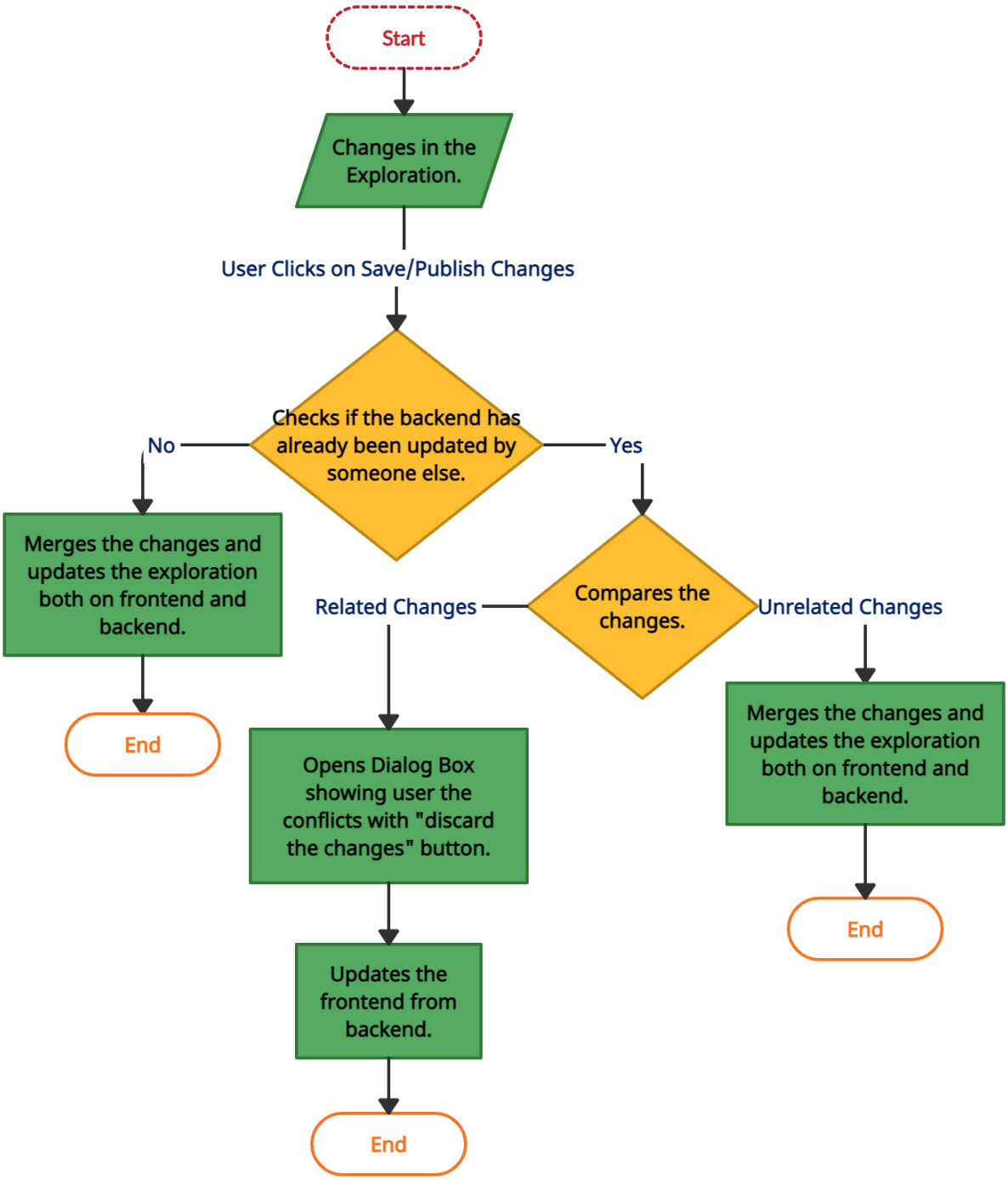
Fig 1: *Dialog showing the conflicts*

So after this project completes, it will start syncing the edits in the background and will merge the changes if the changes are unrelated and will show the conflicts for all the related changes just after saving it.
It will compare the changes from the backend after each save without even publishing the changes.

So if A makes some changes in the description of state 1 and then click on the save button and publishes it, the changes will get saved at the backend server. If B also makes some changes in the description of the same state related to A's changes and then clicks on the save button, it will immediately open a dialog (as shown above in **Fig 1**) showing the conflicts discarding the changes.

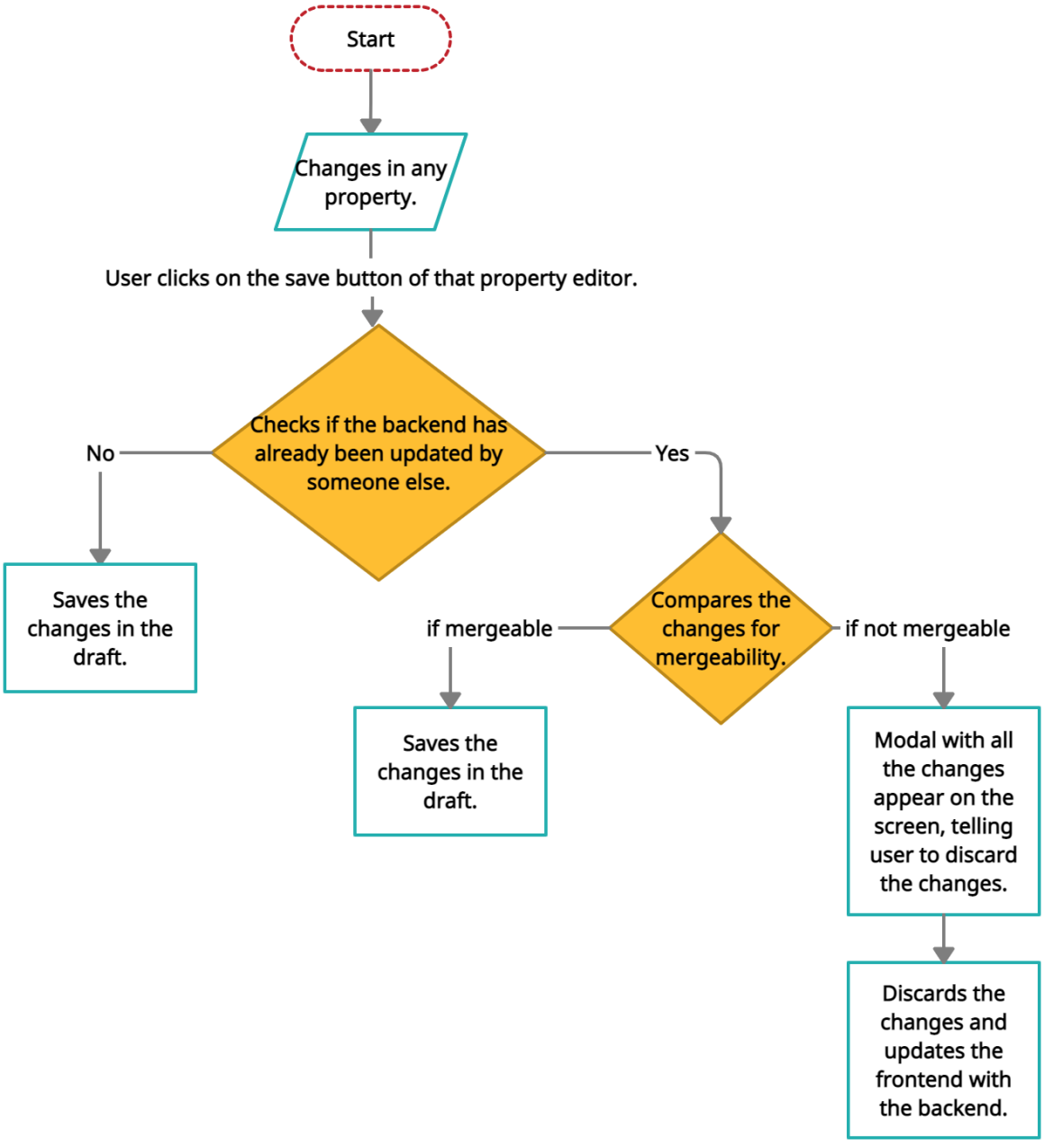
But suppose B changes something in the other states or anything else in the same state, which is unrelated to A's changes *(as often creators work on different parts of the exploration, and it may not intersect)*. In that case, the changes will be merged automatically without any delay and will also be updated in the backend.

This is an overall flow diagram of a process for a single user.



Flow Diagram 1: *Flow diagram for updating the explorations process*

Below I am adding a separate flow diagram for what happens when draft changes are saved.



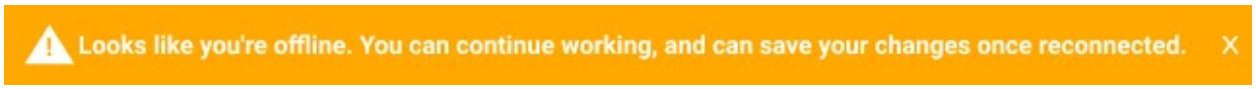
Second Part:

Sometimes we get stuck because of our network problems, and then we need to postpone our work which may lead to not completing the work before deadlines or getting a lot of work piled up at the last minute. Suppose, If you are editing some explorations and suddenly your internet connection goes down, what will happen next?

Presently, suppose any creator is making any changes in their exploration, and suddenly their network connection starts behaving weirdly (like it becomes flaky or disconnects the user). In that case, all the progress and changes made by the user are lost.

After completing this project, if any creator is making any changes in their exploration and suddenly loses their network connection, all their changes will be saved locally until they find reliable network access. When they get a reliable network source, their changes will get saved to the backend and frontend both and will show the conflicts (if any).

So when a user is working on any exploration and suddenly if his network connection loses, he will get a notification on his screen as shown below.



After this, the user can make the changes without worrying about losing them, he can save them by clicking on the save button (shown in the image below), and all his changes will be saved locally.

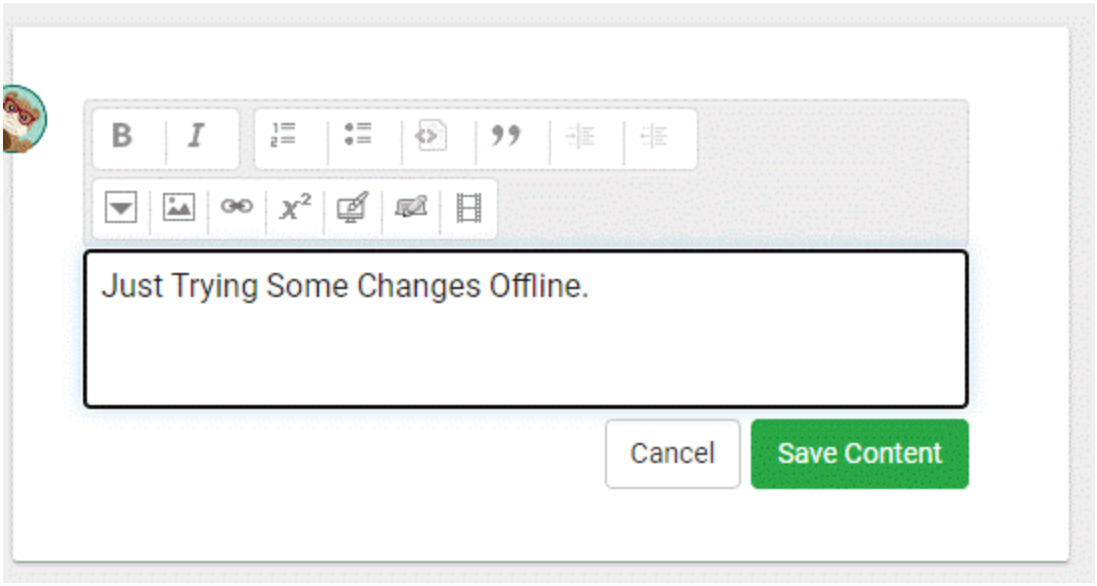


Fig 2: *Making changes in the description of any exploration*

The Save Draft / Publish button will be disabled for users while being offline because there is no use in clicking on that button as all the changes need to be checked (as in subproject 1) before saving. So we'll just autosave the draft.

In case the user try to leave the page (possibly accidentally) then he will get an alert toast as shown below

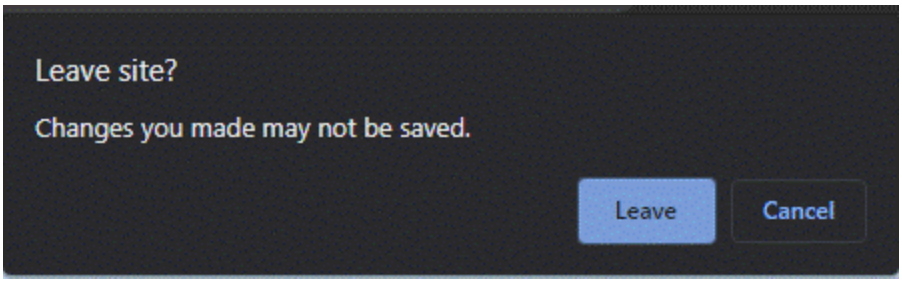
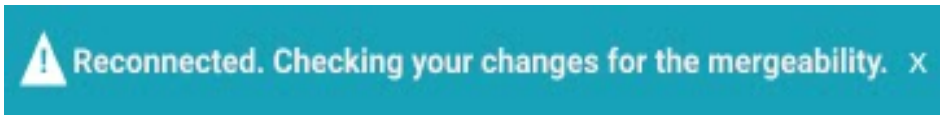


Fig. 3: *Leave the page notification.*

and if the user clicks on leave then he will lose all his progress.

After this, when the user gets a good connection back, he will get one more notification on the screen.



Now if the changes are not mergeable then the user will simply get the modal showing the changes to be discarded and telling him that changes can not be merged similar to what happens normally in a good internet connection.
And if the changes are mergeable and merged then he will get this notification as shown below.

After this, **Fig 2** process will start again, and if he gets any conflicts, he will get the same dialog shown as in **Fig 1** just after this notification shown above**;** otherwise, his progress will get saved. And this is how the offline autosaving process will work.

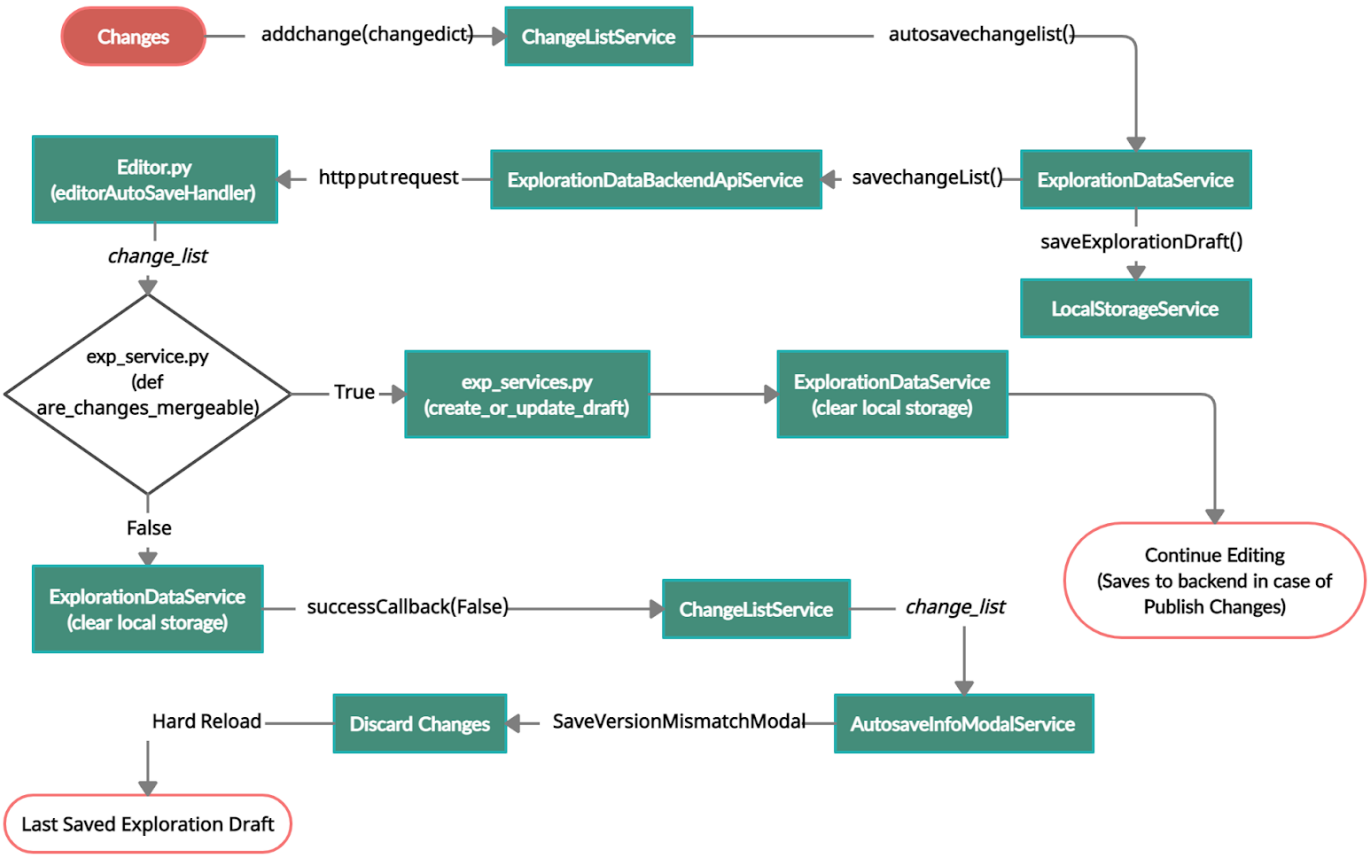
Technical Design

Here also, as there are two sub-projects so I'll complete the technical design in two sections. Let's just start with the first one.

First Sub Project:

Architectural Overview

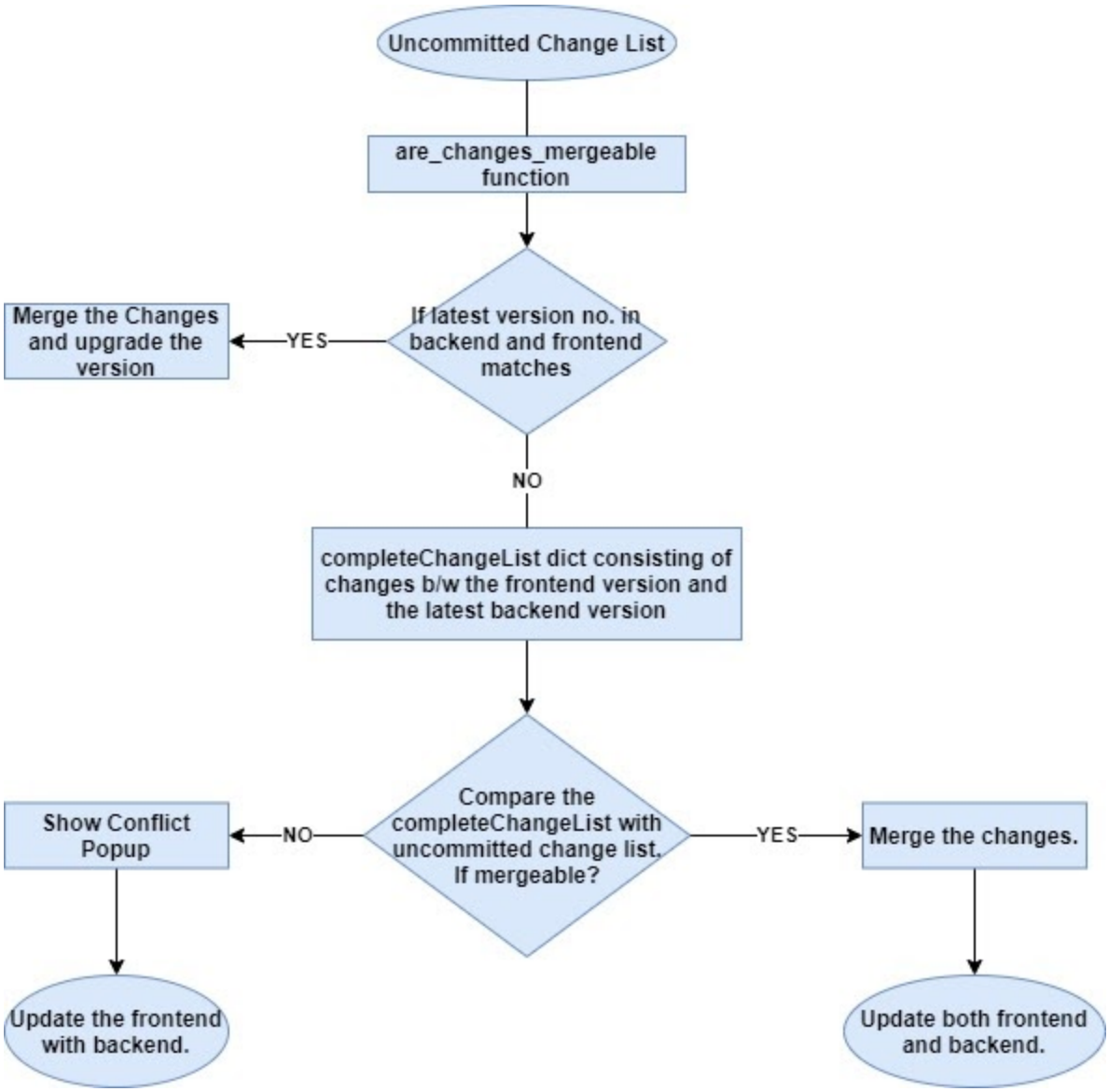
I will try to explain the architecture through the sequence diagram given below.
In ***exp_services.py,*** instead of ***is_version_of_draft_valid,*** we will introduce a new function called ***are_changes_mergeable,*** which will return **true** if the changes can be merged into the backend or will return **false.**
To check the condition stated above, we will first compare the latest version number of the exploration in the backend with that on the frontend. If the versions are the same, then the uncommitted changes can be easily merged, and if not, then we will compare the changes done in the backend with the uncommitted change list on the front end and then decide accordingly.



Flow Diagram 2: *Explaining how the whole process of saving changes take place.*

Implementation Approach

Let's start with a flow diagram of how the uncommitted change list will travel and the exact process. These all the checks mentioned in this flowchart will take place in the backend.



Flow Diagram 3: *Flow diagram to show the functionality inside are_changes_mergeable function.*

We already have the **uncommitted change list**, including all the changes done until now on the front end, and haven't been saved in a draft or published. We'll pass this change list into ***are_changes_mergeable*****,** where the comparison will be made.

are_changes_mergeable function will have a structure like this:

```python
def are_changes_mergeable(exp_id, change_list, version):
    """Checks if the changes made by present user can be merged or not.

    Args:
        exp_id: str. The id of the exploration.
        change_list: dict. changes made by the users
        version: int. The draft version which is to be validate.

    Returns:
        bool. Whether the change has merge conflict or not.
    """
    if exp_fetchers.get_exploration_by_id(exp_id).version == version:
        return True
    else:
        latest_version = exp_fetchers.get_exploration_by_id(exp_id).version
        completeChangeList = exp_fetchers.getCompleteChangeList(exp_id, version, latest_version)
        """Here comparison between change_list and completeChangeList will be done.
        .
        .
        .
        Will return True if changes will be mergeable, otherwise will return False."""
        return False # Default rejection for mergeability.
```

Before writing the conditions for the comparison, let's discuss the **completeChangeList** and how we can get it. So, now to compare the changes done both in the backend (already made by some other users) and frontend (non-committed change list), we'll need all the changes done in the backend and frontend so far. We already have the uncommitted change list and to get the changes already done in the backend by other users, we'll take all the versions' (between latest on the backend and on which the user is working) change lists and then concatenate them into one complete change list called **completeChangeList**.

To get the **completeChangeList,** we'll create a function called ***getCompleteChangeList(exp_id, v1, v2)*** in ***exp_fetchers.py*** with arguments:

- **exp_id:** Id of exploration user currently working on.
- **v1 and v2:** Version numbers of the exploration of where to where the user wants to find the exact complete change list.

getCompleteChangeList will be the exact backend version (the code will be in python for the backend) of this function shown below. Here in frontend we have used version tree service to get the change list of any version for the current exploration but while writing this function in the backend, we'll pass the current exploration id as argument and then get the change list of any version from the snapshots stored of that exploration.

```
// v1 is the current version of the exploration on the frontend.
// v2 is the latest version  of the exploraion in the backend.
var _getCompleteChangeList = function(v1, v2 ) {
      // Stores the path of version numbers from v1 to v2.
      var versionPath = [v1+1, ... ., v2];

      // The full changelist that is applied to go from v1 to v2.
      var completeChangeList = [];
      versionPath.forEach(function(version) {
        var changeListForVersion = VersionTreeService.getChangeList(version);
        completeChangeList = completeChangeList.concat(changeListForVersion);
      });
      return completeChangeList;
    };
```

After getting the completeChangeList, now, let's focus on the comparison of the changes. So the changes may include the changes of state names also and as there's a lot of dependency on statenames, let's discuss separately about tracking the state identity process. I have explained it in the steps below:

- Create a new temporary dict called **changed_statenames,** where we will store the new state names mapped to the old state names, a **new_states** list where we'll store all the new states added and a **deleted_states** list to store all the deleted_states names..
- We'll iterate through the **completeChangelist** and look for the changes in which the state name is changed, state is added or state is deleted. Now let's discuss the various conditions:
    - **New state is added:** In this case we'll push the name of the state in the **new_states** list.
    - **State is deleted:** In this case we'll push the name of the state in the **deleted_states** list.
    - **States renamed once or multiple times:** In this case we'll simply store the new name (as in the latest version on the backend) mapped to the old name (as in the browser's version), we'll not need the names changed in between as we have to merge the changes in the latest version from the browser version.
    - **State deleted and then added again:** In this case when a same state is added again, we'll treat it as a new state only because we don't know whether the properties inside it are different or same, like maybe a user has created a different state with the same name again, therefore we'll need to treat that state as a new state only. So we'll save that name in both **new_states** as well as **old_states** list if it does not exist in that list.

- **State added and then deleted:** In this case we'll remove that state name from the **new_states** list as it is not in the latest version at the backend and hence is not a problem for us.

- Now we'll iterate through the uncommitted change list and look if the current user has made any change in any state's name or added or deleted any state.
  - If renamed, then we'll check for that state in **changed_statenames,** if found, then that means that some other user also changed that state's name so the current user will get the conflict.
  - If added or deleted any state, then we'll simply check no new state is added and no old state is deleted in the backend otherwise will show the conflict because addition and deletion of the state changes the flow of the exploration and it will be very complex to handle the new addition or deletion of any state from another change list.

While applying the other changes, we will use the changed state name from the **changed_statenames** dict created above to avoid future conflict and will also look for that state in **deleted_states** list to check if that state is not deleted.
changed_statenames will have a structure like this:

```
changed_statenames = {
  "state1_old_name": "state1_latest_name",
  "state2_old_name": "state2_latest_name",
}
```

Moving further, we need to simplify the comparison process, so, to do that, we can create a temporary dict called **changed_properties,** where a key will be the latest state name (taken from the **changed_statenames** dict created above), and the value will be a dict containing the name of the properties changed in that state along with the changes made in that property from the completeChangeList.
The **changed_properties** dict mentioned above will have a structure like this:

```
changed_properties = {
    "latest_state_name1": {
        "property1": [changes...],
        "property2": [changes...]
    },
    "latest_state_name2": {
        "property1": [changes...],
        "property2": [changes...]
    },
}
```

I am basically dividing the changes from the completeChangeList on the basis of states and properties. For example if the user changes the content of an introduction state from "hello" to "hello world", then the changed properties dict will look like this:

```
changed_properties = {
  "Introduction": {
    "content": [{
            "new_value": {
                "content_id": "content",
                "html": "<p>hello world</p>"
            },
            "cmd": "edit_state_property",
            "old_value": {
                "content_id": "content",
                "html": "<p>hello</p>"
            },
            "state_name": "Introduction",
            "property_name": "content"
        }]
    }
}
```

We will just iterate through our uncommitted latest change list and check that if the property we are updating now is already changed in the backend or not. For this, we'll look for the property in the **changed_properties** dict, and if found, we'll start comparing its changes.

Here is an **example** below to explain all the terms used above with the help of flowchart (on the next page).
So there are two users A and B starting together from version 5 of an exploration. A makes some changes and publishes them while B is still making the changes without committing or publishing them. So after A changes, backend version upgrades to 7 while B is still on version 5 so when he will try to make the changes, his changes will be checked before merging and for that:

- **completeChangeList =** ChangeList 1 (changes from version 5 to 6) + ChangeList 2 (changes from version 6 to 7)
- **changed_statenames =**

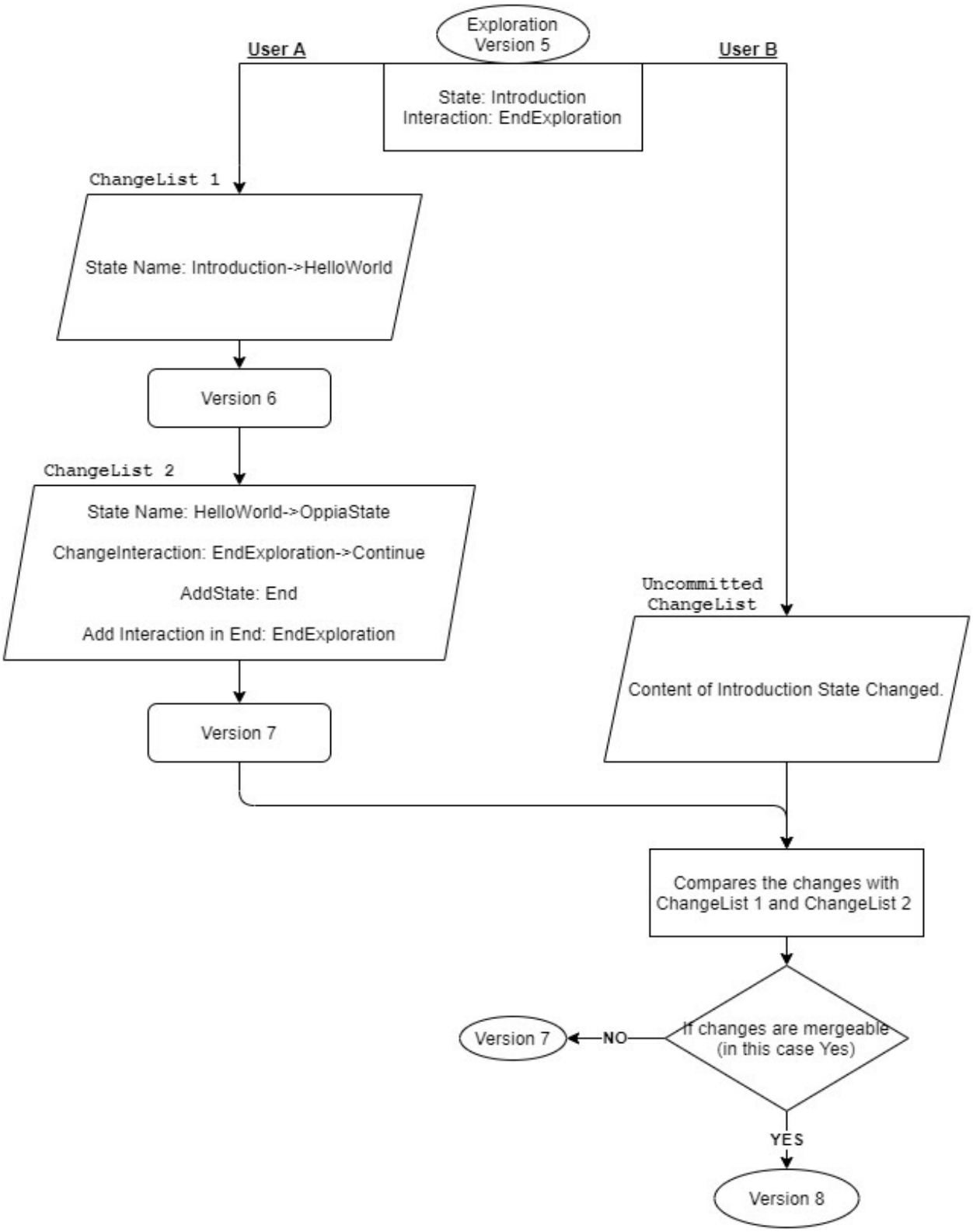
```
changed_statenames = { "Introduction": "OppiaState"}
```

- **changed_properties =**

```
changed_properties = {
    "OppiaState": {
        "state_name": [changes...],
        "interactionId": [changes...]
    },
    "End": {
        "content": [changes...],
        "interactionId:" [changes...]
    }
}
```

- **new_states =**

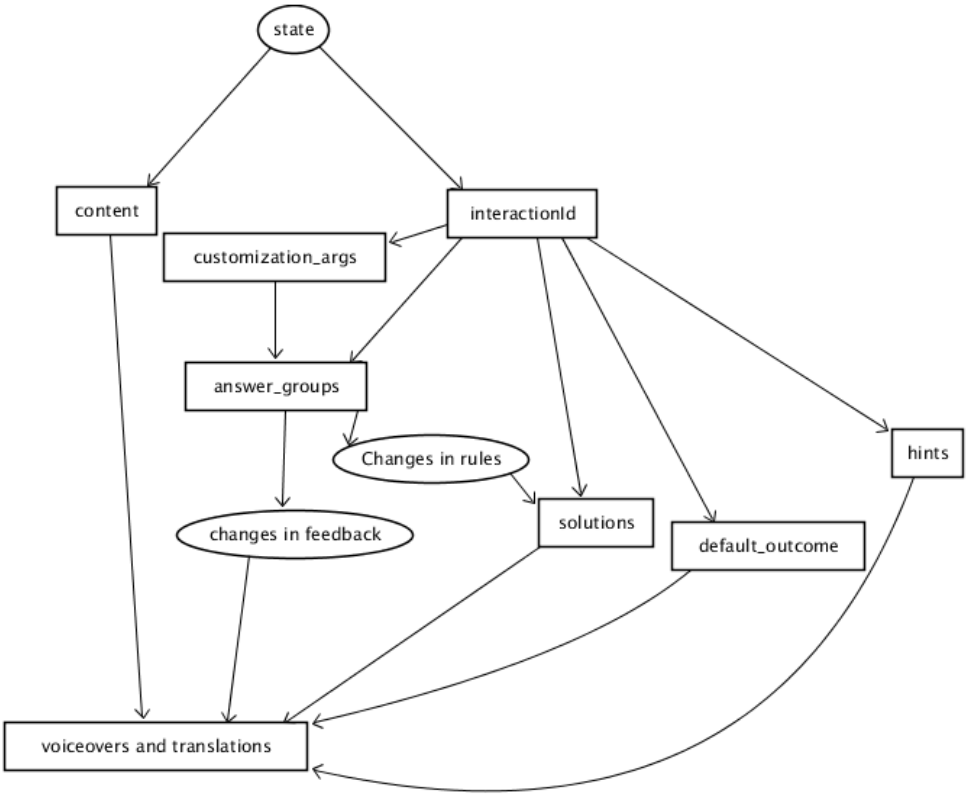
```
new_states = [ "end" ]
```



Flow Diagram 4: *Explaining the process*

Now let's discuss all the conditions for the **True** and **False** of **are_changes_mergeable** function**.** So here I have defined the loci of the changes i.e. if the changes are in these properties then what all other properties will be affected:

- **Content:** Affects Translations & Voiceovers only.
- **Interaction ID:** Affects everything related to interaction i.e. solution, answer_groups, customization_args, confirmed_unclassified_answers and hints.
- **Customization_args:** In interactions like image region, item selection, multiple choice, drag and drop sort, all the maths interaction, code editor and music interaction, customization_args affects answer_groups and solutions, and therefore affects voiceover and translations.
- **Answer_groups**: Affects the solutions if the change is in the rules of the answer group and affects the voiceovers and translations if the change is in the feedback of the answer group.
- **Default_outcomes**: Affects feedback and therefore voiceovers and translations.
- **Hints**: Affects only voiceovers and translations.
- **Solutions**: Affects only voiceovers and translations.
- **Exploration properties like Title, Goal, Language, Category, tags and the name of the first card**.: The name of the first card is affected by the change in the state name but can be easily handled with the **changed_statenames** dict created above.



Flow Diagram 5: *Chart of the properties affected by other properties.*

Let me just explain the various scenarios. So below are the changes made by the second user after the first one has updated the backend. I have explained the cases in which the user will get into a conflict.

- **Changes in the content:** These changes are not affected by any other property changes, therefore can be easily merged if someone has not changed the content of the same state in the backend .

- **Change in the interaction id:** This change also is not affected by any other property changes except for the changes in the interactionId itself

- **Changes in the solutions of an interaction:** If someone changes any answer_group in the same state irrespective of the solution's relation with it, the conflict will be shown because checking every answer group in the change_list to satisfy the solution is not a good method to follow here. Also affected by interactionId so if someone changes the interaction then the solution can not be changed for the obvious reasons that for every interaction the type of solutions are different and also if in the backend someone changes the interaction twice and restores back the original interaction then we'll check if the answer_groups or customization_args are changed or not, if changed then we'll show the conflict else we'll merge the changes.
  In interactions like image region, item selection, multiple choice, drag and drop sort, all the maths interaction, code editor and music interaction, they are also affected by customization_args and hence will show the conflict if the args are changed.

- **Changes in the customization_args:** Affected by interactionId only. So any changes in the interaction id (i.e. Interaction type) or the customization_args itself in the backend will show the user a conflict error.

- **Changes in the answer_groups:** Affected by the interactionId always and by the customization_args in interactions like image region, item selection, multiple choice, drag and drop sort, all the maths interaction, code editor and music interaction. So any changes in these two properties or in the answer_groups itself in the backend will show the user a conflict error.

- **Changes in the default_outcomes:** These are not affected by any other property but in case of the change in the interactionId, we'll need to show the conflict as the default_outcome may be different according to the interaction.

- **Changes in the Hints:** Affected only by the interactionId. So any changes in interactionId or in the hints itself will show a conflict error in this case New hints can be added easily.

- **Changes in the voiceovers and translations:** Affected by changes in the content, feedbacks, hints, solutions, and default outcomes only. So if any of these properties are changed in the backend and we try to change the voiceovers and translations of those properties then the user will get a conflict error.

- **Changes in the exploration properties like title, goal, language, category, tags or the name of the first card:** These changes are not affected by any other changes except the state name change, which can be handled by the new and old names of the states I am storing and therefore the conflict will not be shown until or unless these values are itself changed.

So above I have covered all the changes and when they will show the conflicts.
Now, we'll check the two conditions:

- If any two changes (one from backend completeChangeList and one from user's change list) lie in any of the scope mentioned above. If they lie then that means we should show the conflict.
- Now we will compare the value of that property from the browser's version with the value of that property in the latest version at the backend. This is done to check that the net effect of the change list is to change that property. For example: Suppose if the net effect of one changelist is to change the content from A to B and then back to A (as well as do some other things on the exploration), and the net effect of the other changelist is to change the content from A to C. So in this case the changes should be merged.

Therefore if the values in the second condition are not the same i.e. net effect is changed and the first condition is satisfied too, then the user should get a conflict error and then the ***are_change_mergeable*** will return **False** and will show the conflict, telling the user to discard the changes.

Also, I will add the ***are_changes_mergeable*** condition in ***EditorAutosaveHandler*** in editor.py so that changes will be saved to draft only if there aren't any conflicts and this will avoid losing all the changes at last i.e. I will restore the last saved draft in case of merge conflicts.

```
    are_changes_mergeable = exp_services.are_changes_mergeable(
                exploration_id, change_list, version)
    try:
        if can_edit and not are_changes_mergeable:
            exp_services.create_or_update_draft(
              exploration_id, self.user_id, change_list, version,
              datetime.datetime.utcnow())
        elif can_voiceover and not has_merge_conflicts:
            exp_services.create_or_update_draft(
              exploration_id, self.user_id, change_list, version,
              datetime.datetime.utcnow(), is_by_voice_artist=True)
    except utils.ValidationError as e:
        # We leave any pre-existing draft changes in the datastore.
        raise self.InvalidInputException(e)
```

After this, in case the changes are not mergeable, we will show the conflict modal from **autosave-info-modal.service,** where the function will look similar to this:

```
    showVersionMismatchModal: function(lostChanges) {
      $uibModal.open({
        templateUrl: UrlInterpolationService.getDirectiveTemplateUrl(
          '/pages/exploration-editor-page/modal-templates/' +
          'save-version-mismatch-modal.template.html'),
        // Prevent modal from closing when the user clicks outside it.
        backdrop: 'static',
        resolve: {
          lostChanges: () => lostChanges
        },
        controller: 'SaveVersionMismatchModalController',
        windowClass: 'oppia-autosave-version-mismatch-modal'
      }).result.then(function() {
        _isModalOpen = false;
      }, function() {
        _isModalOpen = false;
      });

      _isModalOpen = true;
    },
```

Second Sub Project:

Architectural Overview

As discussed above in the product design section, we here need to enable offline functionality so that creators can work offline too and their changes can be updated to the backend when the creator is online.

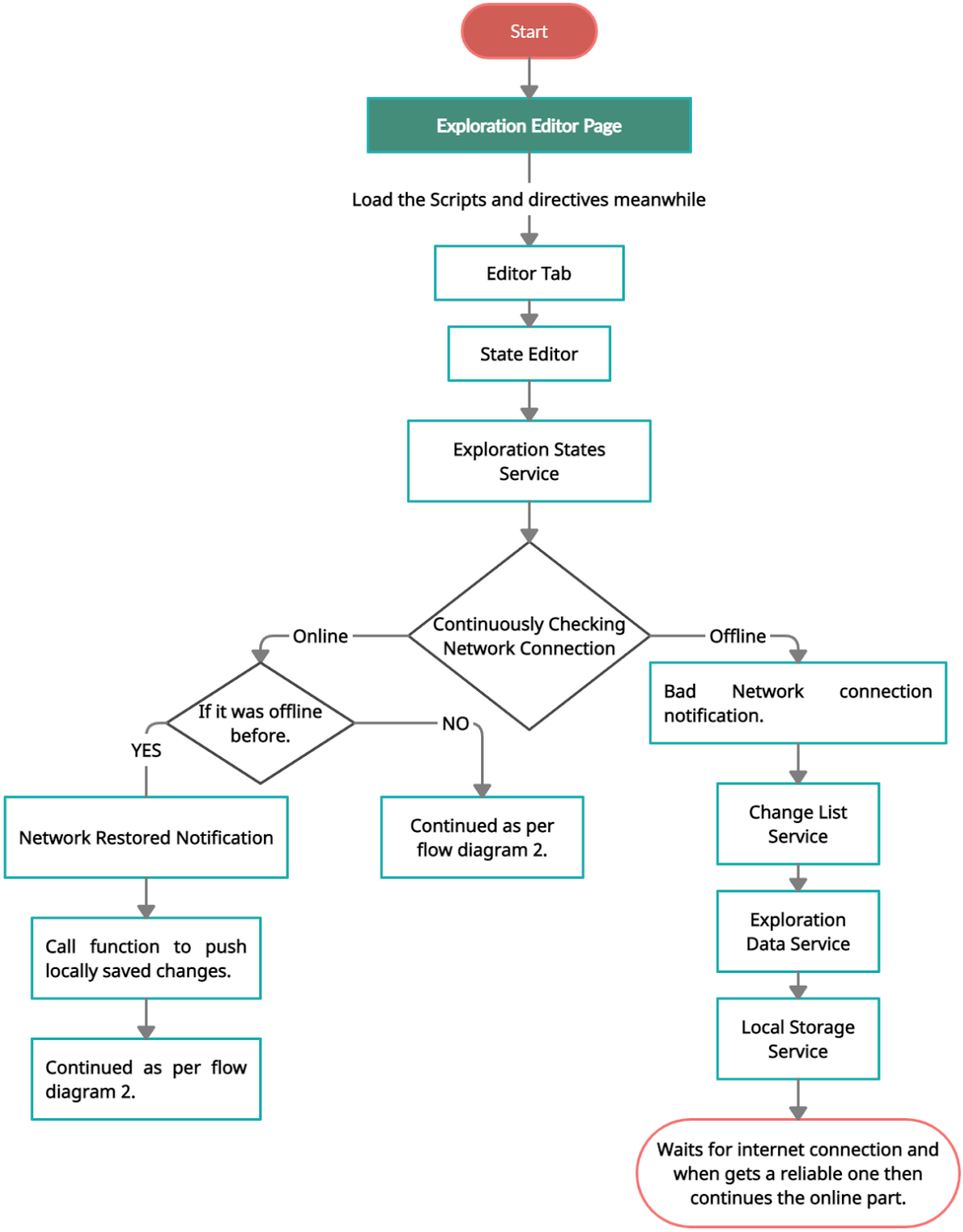
Here first we need to continuously monitor the network connection of the user and then display the alert notification based on the offline and online status. For this, I'll add a functionality in which I'll send the continuous get requests at the fixed intervals to check for internet connection and will also use browser's online/ offline events and will listen to them using **fromEvent** listener to check whether the device is connected to any network source or not. I have explained the functionality of this in the implementation approach.

Also, simultaneously I need to import all the services, directives/components, modals, and scripts which a creator will need during editing exploration. Because when he will be offline then he will not be able to load any modal so we'll need to preload them.

After that in the exploration data service, we will save only in the local storage, and then when the network is back we'll call the function to save the data in the backend and then the changes saving process will work as explained in the first sub project technical design.

At last, we will add an event listener to check if someone tries to close the window and as told in the product design, it will ask the user whether they are sure about closing the window because it will lose their progress.

So here is a sequence flow diagram to give you a better understanding:



Flow Diagram 6

Implementation Approach

Firstly, in order to monitor network connection we will create a new service called ***ConectionService,*** which will continuously check for network connection using ***fromEvent*** listener to listen to window offline and online events and also for internet connection by sending the get request at a fixed interval (kept it 3 sec for now as it will neither be too frequent, avoiding any load on the server and it neither be too slow, but this can be changed so will discuss it with the mentor and then decide) and based on response will show whether the user is offline or online. So in order to check network state, i'll add a function called ***checkNetworkState*** in this service, which will look like this:

```
  private checkNetworkState() {
    this.onlineSubscription = fromEvent(window,
'online').subscribe(() => {
      this.currentState.hasNetworkConnection = true;
      this.checkInternetState();
      this.emitEvent();
    });

    this.offlineSubscription = fromEvent(window,
'offline').subscribe(() => {
      this.currentState.hasNetworkConnection = false;
      this.checkInternetState();
      this.emitEvent();
    });
  }
```

And also I will add a function to check internet connection which will look like the snippet below. This ***ConnectionService*** can also be used in future to check the network connection while making other things workable offline too.

```
private checkInternetState() {

    if (!_.isNil(this.httpSubscription)) {
      this.httpSubscription.unsubscribe();
    }

    if (this.serviceOptions.enableCheckingConnection) {
      this.httpSubscription = timer(0, 3000).pipe(
          switchMap(() => this.http[this.serviceOptions.requestMethod]
            ('UrltoSendGetRequest', {responseType: 'text'})),
          retryWhen(errors =>
            errors.pipe(
              // log error message
              tap(val => {
                console.error('Http error:', val);
                this.currentState.hasInternetAccess = false;
                this.emitEvent();
              }),
              // restart after 5 seconds
              delay(5000)
            )
          )
        )
        .subscribe(result => {
          this.currentState.hasInternetAccess = true;
          this.emitEvent();
        });
    } else {
      this.currentState.hasInternetAccess = false;
      this.emitEvent();
    }
  }
```

We will also create ***ConnectionServiceModule*** to declare ***ConnectionService*** and it will look like this:

```
import {NgModule} from '@angular/core';
import {ConnectionService} from './connection-service.service';
import {HttpClientModule} from '@angular/common/http';

@NgModule({
  imports: [HttpClientModule],
  providers: [ConnectionService]
})
export class ConnectionServiceModule {
}
```

After this, we will just import ***ConnectionServiceModule*** in ***exploration-editor.page.module.ts*** and then declare it in the imports.
Now, we'll need to inject ***ConnectionService*** in ***exploration-editor.page.component.ts's*** constructor and in this component, we'll subscribe to ***monitor()*** to get notification whenever the internet status is changed.
It will look like this:

```
import { Component } from '@angular/core';
import { ConnectionService } from 'ng-connection-service';
import { AlertsService } from 'services/alerts.service';
.
.
.

@Component({
  selector: 'explorationEditorPage',
  templateUrl: './exploration-editor.component.html',
  styleUrls: []
})
export class ExplorationEditorPageComponent {
  let isConnected: boolean = true;

  constructor(private connectionService: ConnectionService,
              private alertsService: AlertsService) {
    this.connectionService.monitor().subscribe(isConnected => {
      this.isConnected = isConnected;
      if (this.isConnected) {
        this.alertsService.addWarning("Looks like you're offline." +
            "Your changes will be saved once reconnected");
      }
      else {
        this.alertsService.addInfoMessage("Reconnected." +
            "Saving Your Progress.");
      }
    })
  }
  .
  .
  .
}
```

Now, we'll need to import some scripts and directives too so I made a list of them.

- Ckeditor
    - Config.js **static/ckeditor-4.12.1/config.js?t=J5S9**
    - Skin.js static/ckeditor-bootstrapck-1.0.0/skins/bootstrapck/skin.js?t=J5S9
    - Editor.css static/ckeditor-bootstrapck-1.0.0/skins/bootstrapck/editor.css?t=J5S9
    - en.js static/ckeditor-4.12.1/lang/en.js?t=J5S9
- Delete-interaction-modal.template.html
- Outcome-feedback-editor.directive.html
- Customize-interaction-modal.template.html
- All the interaction inputs
- Add-answer-group-modal.template.html
- State-solution-editor.directive.html
- Save-validation-fail-modal.template.html
- Confirm-leave-modal.template.html
- Add-hint-modal.template.html
- Delete-hint-modal.template.html
- Select2-dropdown.directive.html
- Response-header.directive.html
- save-validation-fail.modal.ts

So we need to import and load all these files when the exploration editor page loads for the very first time.

_autosaveChangeList function in ***exploration-data.service.ts*** at present initiates the change list to save in both backend and localstorage. It looks like this:

```
private _autosaveChangeList(
      changeList: ExplorationChange[]): Promise<DraftAutoSaveResponse> {
    this.localStorageService.saveExplorationDraft(
      this.explorationId, changeList, this.draftChangeListId);
    return this.explorationDataBackendApiService.saveChangeList(
      this.explorationDraftAutosaveUrl,
      changeList,
      this.data.version,
    ).pipe(
      tap(response => {
        this.draftChangeListId = response.draft_change_list_id;
        // We can safely remove the locally saved draft copy if it was saved
        // to the backend.
        this.localStorageService.removeExplorationDraft(this.explorationId);
      })).toPromise();
  }
```

As we can see above, first it saves to local storage and then updates backend draft, so in case of offline we need to save only to local storage so we'll not run rest of the function and when the network reconnects, we'll just send the change list to backend again or we can call this whole function again.

Now we need to keep the check for if someone does not close the window also. For that we can just add an event listener as shown below.

```
@HostListener('window:beforeunload', ['$event'])
    unloadNotification($event: any) {
        if (this.hasUnsavedData()) {
            $event.returnValue =true;
        }
    }
```

Now in order to show this alert when the navigation is changed also, we need to implement a ***can deactivate*** guard.
First we need to implement the ***CanDeactivate*** interface. It will look like this:

```
@Injectable()
export class CanDeactivateGuard implements CanDeactivate<UserFormComponent> {
  canDeactivate(component: UserFormComponent): boolean {

    if(component.hasUnsavedData()){
        if (confirm("You have unsaved changes! If you leave, your changes will be lost.")) {
            return true;
        } else {
            return false;
        }
    }
    return true;
  }
}
```

After this we just need to add ***CanDeactivateGuard*** to the ***ngModule*** providers.

Third-party Libraries*

I am not using any third party libraries.

Testing Approach

First Sub Project:

To test this feature, we need to do the following steps.

1. Create two users A and B. Login as A.
2. Create a new exploration and add B as a collaborator in that exploration.
3. Add some content and add endExploration interaction in the Introduction state and then publish it.
4. Let A keep the editor open while login B on incognito in the same browser because we can't login simultaneously using another account on the same window so we'll need to use incognito.
5. Open the same exploration(in which B's a collaborator) in B's window. Open the exploration editor.
6. Remove the endExploration interaction from the introduction state and add a continue button interaction.
7. Add a new state with name "end" and set the destination of the continue button to end state. Save the changes and publish them.
8. You can see that on A's side the exploration version is still the old one. So now, make some changes from the A' side in the content and then save and publish it. We'll see that it will not show any error and after publishing it will load the latest version with all the changes(including B's changes).
9. A's changes haven't been updated on B's side. So now make some changes in content of "introduction" from B's side.
10. B will get a conflict error message. And he will need to discard his content changes and the screen will automatically reload with the latest version (including A's and B's).
11. Let's take one more example. Change continued interaction from "introduction" state to numeric input interaction with hints solutions etc. And add destin. "end "state to its correct answer group.
12. Save and publish the changes and then reload the exploration on both windows to load all the changes.
13. Now change the rule in answer_group from A's side and accordingly the solution and then save and publish the changes.
14. B's side is still on the old version so change the translations of the feedback from the answer groups and then save and publish them. Changes should be merged easily.

15. After B's merging, B's on the latest version with all the changes so far. Change the rules again, like if it was "number should be less than or equal to" before then change it to "number should be greater than or equal to" now. Update the solution accordingly and save and publish it.
16. A is still on the old version so now change the solution only and save it. A conflict popup will appear on screen as the changes are related. Discard the changes and the exploration will reload to the latest version.

Second Sub Project:

To test this feature, we need to do the following steps.

1. Create a user and login.
2. Click on the create button in the navbar to create the new exploration.
3. Once the exploration editor page loads completely, then disconnect the internet. A warning notification should appear on the screen telling the user, "*Looks like you're offline. Don't worry, your progress will be saved once reconnected.*".
4. After that, click on the content box, type some content, and then click on the Save button. Also, add an endExploration Interaction to it.
5. Connect back to the internet. An information notification should appear informing the user that his progress is being saved. Now Close the window and reopen that same exploration. Content written before, and an end exploration interaction should be there.
6. Now disconnect the internet again, and the same notification should appear as appeared above in the 2nd point. Make some changes in the content and then click on the close window icon, and then an alert box should appear informing the user that he will lose all his progress.
    a. Click on the leave button, and the window should have been closed. Again open that exploration after connecting to the internet, and the changes you made in the content should not be there.
    b. Click on the cancel button and then try step 4

Milestones

Milestone 1

Key Objective: Introduce functionality such that edits made by a user should be propagated to all clients. The changes should be applied if the changes are unrelated or else the user should be informed of the merge conflict.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Added the completeChangeList functionality in the backend to get all the changes from version v1 to v2 in one array.	-	14/06/2021	18/06/2021
1.2	Added the are_changes_mergeable function in exp_services to check whether the changelist can be merged or not.	1.1	25/06/2021	30/06/2021
1.3	When a user saves an exploration, their changes are applied directly if they are mergeable; otherwise, they get a merge conflict popup if their changes cannot be merged.	1.2	04/07/2021	08/07/2021

Here is the list of all the files I am planning to edit in this milestone.

- **change-list-service.ts** to change the response functionality of add change function.
- **Exploration-data-service.ts** to change the successCallback function on autosave and save of exploration.
- **auto-save-info-modal.service.ts** to change the discard change modal as we needed it.
- **exp_services.py, editor.py, exp_domain.py, exp_fetchers.py** to include all the functionalities at the backend.

Milestone 2

Key Objective: Enabling the exploration to work offline in case of connectivity issues.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged

2.1	Scripts and components needed will be preloaded and all the modals in the exploration editor will be workable offline.	-	15/07/2021	20/07/2021
2.2	Checking continuously for network connection functionality	-	20/07/2021	25/07/2021
2.3	Saving the data only to local storage in case of offline and then updating it on the backend after the internet reconnects.	2.1 and 2.2	30/07/2021	6/08/2021
2.4	Added event listeners to check all the leaving editor window attempts.	-	5/08/2021	10/08/2021

Here is the list of all the files I am planning to edit in this milestone.

- We will create the new **connection-service.ts, connection-service.spec.ts, connection-service-module.ts** files to check the network connection.
- **exploration-editor-page.module.ts, exploration-editor-page.mainpage.html, exploration-editor-page.component.ts, exploration-editor-page.component.html** to preload all the scripts and components needed, to add the connection service check and the event listeners for window leaving attempts.
- **local-storage.service.ts** to complete the locally saved changes functionality.

Future Work

I will try to make other services workable offline too like we can work for learners to play exploration offline or maybe we can also work on the topics and skills creator to do the changes offline and also syncing edits in the background for them too.