GSoC - 2018 Project Proposal

- Akshay Anand

Project Name - Add Functionality for Skills

Why I am interested in working with Oppia.

I found about Oppia in December 2017, when I was searching for organizations to start for Open Source contributions, and from then on, have been a regular contributor to the Oppia platform. My initial reason to start contributing to Oppia was that I knew the languages used by the platform (AngularJS and python) well and so decided, this would be the best place to start with open source development. As I got involved in the project, I became more and more invested in the ideals of the Oppia Foundation, which is to provide a simple and easy to use learning platform in which anyone can share their knowledge about a subject to the world.

The thing that stood out most to me about the entire structure of the Oppia website was it simplicity. I have used some learning platforms, but most of those would not be ideal for anybody to use. With Oppia, as seen from the recently conducted RCT, even primary school children were able to use Oppia with ease, which is something to be really proud of.
Coming to the RCT, I was also involved in developing certain new features which was tested in the recently conducted RCT. From the review that we had after it, everyone shared a lot of insights on how to make Oppia better and even more easy to use and intuitive for people to use. Also, interacting with the community as a whole, it is evident that everyone, from different parts of the world, are working toward a common goal for providing free education to anyone in a simple and easy to access platform.

My interests about this project and why I think it's worth doing.

Going back to the RCT, as I mentioned, I was involved in developing a new feature for it, which was refresher explorations. This feature, which although served it purpose to some extent, was not a very streamlined approach to the problem, and hence, when I found a project which replaced that with a much better alternative, I thought, this would be the perfect project for me to do.
This would also greatly increase the learner experience as currently, with refresher exploration, a lot of redirection to and from pages are present, which could confuse learners, while with Skills, it is going to be played as a part of the exploration itself, and therefore would make the transition to and from a "refresher skill" seamless.
Coming to the introduction of topics and stories to Oppia, to replace collections, these would also greatly increase the site's functionality. One problem, that was seen in the RCT, that could

be fixed is that the learner was jumping to later explorations in a collection, without completing the previous ones. With the skills construct added to stories, this could be prevented and as such, make sure that the learner knows all the prerequisites before starting a lesson.

Prior Experience

I was part of the web operation teams at the technical festival (Shaastra) of our institute (IIT Madras) in my 3rd semester in college (I am currently in my 4th Semester). I had started getting involved with web development as a preparation for this in my first year itself. In this team, we had used MEAN stack for creating the web portals and hence I got hands-on experience with general backend development (node js in particular) as well as AngularJS. I had already learnt python as a part of a course in our college, and thus combining this with my knowledge on backend development as well as AngularJS, I was able to grasp the overall flow of control in the Oppia codebase, as well as knowing which files to change to implement specific features.

Other than this, I have also participated in hackathons related to web development and also done competitive coding, in which I participated in major competitions like Codechef Snackdown (passed pre-elimination stage), ACM-ICPC (qualified for regionals), Google Code Jam (passed 1st stage) etc. I have also done contests in coding platforms like Codechef and Codeforces. All of this also helped me in tackling bugs or adding features to the codebase as it made me think about effective ways to do the same.

Link to Pull Requests with Oppia

The first major PR I had of fixing a bug in non-inline interactions: #4232
My PRs related to refresher explorations: #4307, #4344, #4373, #4611, #4644
The complete list of pull requests that were merged with Oppia is here.

Project Plan

Current skills implementation and additions done in this project

Currently, the exploration title for each node in a collection is taken as the skill acquired when that node is completed and that is stored in the skill ids section for a collection. For explorations, for certain answer groups, redirection is currently being done to other independent explorations called refresher explorations.

Some shortcomings of the above method are that, the refreshers need not be full-fledged explorations, but just a concept card explaining the lacking skill, and a series of questions that test that skill. Also, the multiple redirections that happen during an exploration play, might confuse the learner, and coming to collections, the skills should not be something restricted to a collection, but it should be a global construct, that is uniform across a general topic.

Hence, the existing collections are to be replaced with topics and story constructs. To achieve this, this project lays the groundwork by adding the topics and stories constructs in the backend and hence, once these are fully implemented in the future, all the collections can be migrated to stories and topics, and collections can be removed.

Also, in this project, the usage of refresher exploration is being replaced by the general skills construct which consists of a concept card and a series of questions testing that concept and the redirections to these skills during exploration play would not require a page reload, as the skill states can be loaded beside the current exploration itself, and played in the same window.

Milestone 1: Complete the backend required for skills, topics and stories and the frontend for a story and topic editor

In this milestone, the backend models, domain objects and controllers related to both the creator and learner experience for skills, and the creator experience for topics and stories is implemented. In addition to this, a simple topic and story editor is also done.

The way the UI is implemented is as follows:

- In the dropdown in the main header, a topics and skills editor link would be there, if the logged in user is an admin.

- Clicking this would go to the topics and skills editor library page, where a fixed number of topics and skills currently in the database would be displayed.
- Here, there would be a Create Topic and Create Skill button which opens the topic and skill editor respectively.
- Then, inside either the Topic editor, the Create Story button would be present to add a story to a topic.

After this milestone, stories and topics editor would be done, and hence new topics and stories can be added to the database. Though, prerequisite skills won't be there, as the skills editor is done as part of the second milestone.

As far as testing its working, a new topic can be created for Fractions, and the current series of explorations in the Fractions collection can be split up into discrete storylines and can be added as stories (that may or may not be linear) to this topic.

Topics

A topic is high-level concept that consists of one or more stories and skills and ties them all in to completely explain a concept. Eg: Fractions.

Schema for Topic

- **Name**: The name of the topic
- **Description**: This is the content to be displayed on the landing page that will explain what the learner will learn through the topic.
- **Canonical story IDs**: These will be a list of ordered story ids that are linked to each other i.e there will be an overall story arc across the list and the learner would play them in a specific order.
- **Optional story IDs:** These will be a list of independent story ids that are optional and are not directly related to the main storyline.
- **Skill IDs:** These consist of the set of skills that the learner will acquire as he/she progresses through the topic.
- **Version:** The current version of a particular topic (will increment after each update).

Schema for TopicSummary

This model would be responsible for storing the summary details of topics, to be displayed as tiled in the topics and skills editor page

- **Name**: The name of the topic
- **topic_id**: the id of the topic that this summary corresponds to.

Schema for TopicCommitLogEntry

This model would be responsible for storing the log of all commits to topics. Every time, an edit is made to a topic, an entry is made in this model.

- **User_id**: The id of the user making the commit.
- **Username**: The username of the user at the time of making the commit.
- **Topic_id**: the id of the topic that was updated during this particular commit.
- **Commit_type**: The type of the commit. Currently, this would contain one of ‘create’, ‘edit’ or ‘delete’. Once, the topic editor becomes more full fledged, ‘revert’ can also be added.
- **Commit_message:** The message given by the user regarding the changes in this commit.
- **Commit_cmds:** The commit command dicts for this commit.
- **Version:** The version number of the topic after the commit.

Files to be Added

The folder /core/storage/topic is to be created in which the following files are to be added to define the Topic Schema in the database.

- *__init__.py :* An empty file to initialise skill model
- *gae_models.py:* This would contain :
    - Class TopicModel: The main schema of the topic and its associated functions
    - Class TopicCommitLogEntryModel for commit history.
    - Class TopicSummary for storing summary details of topics
    - Class TopicSnapshotMetadataModel and TopicSnapshotContentModel: These would be extensions of their respective parent classes in base_models

Stories

A story is similar to the current implementation of collections in that it contains a list of connected explorations. The main difference being that this will implement the new Skills construct as well as the fact the learner is discouraged to skip to later explorations. This can be done by just showing the explorations that the learner can attempt, based on the skills that he/she gained from completing the initial explorations, instead of showing the entire exploration map of the story.
Also, a story has a nonlinear connection between its exploration nodes.

Schema for StoryNode

This will contain the properties related to a single story node.

- **node_id:** A unique identifier for each node.
- **prerequisite_skill_ids:** The list of prerequisite skills needed to start the exploration.
- **acquired_skill_ids:** A list of skill ids that the learner will gain once he/she completes the node.
- **exploration_id:** The exploration that is shown as the story node.
- **Annotations:** These are free form annotations that a story creator can provide so that new explorations can be created inline with the story, by other creators.
- **destination_node_id:** The node that can be accessed (provided prerequisites are completed) after the current node is done.

Schema for Stories

This will store how the various nodes are connected with each other.

- **id:** This uniquely identifies a story
- **title:** The title of the story
- **topic_id:** The topic in which the story appears. This is created so as to make it faster to check, in the frontend, whether an exploration is part of a topic, so that skills can be linked to the exploration. (The exploration will have a story_id field as mentioned in the Schema migration section).
- **story_graph:** This would be the list of nodes within the story and the list of nodes that become accessible after a node is completed is defined in the node schema (though the next node can be started by a user only after he/she has completed all the prerequisites for that node). An adjacency list style implementation can be done here, as stories can be non linear as well.
- **Version:** The current version of a particular story (will increment after each update).

Schema for StoryCommitLogEntry

This model would be responsible for storing the log of all commits to stories. Every time, an edit is made to a story, an entry is made in this model.

- **User_id**: The id of the user making the commit.
- **Username**: The username of the user at the time of making the commit.
- **Story_id**: the id of the story that was updated during this particular commit.
- **Commit_type**: The type of the commit. Currently, this would contain one of ‘create’, ‘edit’ or ‘delete’. Once, the story editor becomes more full fledged, ‘revert’ can also be added.
- **Commit_message:** The message given by the user regarding the changes in this commit.
- **Commit_cmds:** The commit command dicts for this commit.
- **Version:** The version number of the story after the commit.

Schema for StorySummary

This model would be responsible for storing summary details of stories, to be displayed as summary cards.

- **Title**: The title of the story
- **Topic_id:** The topic in which this story is a part of.
- **Story_id**: the id of the story that this summary corresponds to.

Files to be Added

The folder /core/storage/story is to be created in which the following files are to be added to define the Story Schema in the database.

- *__init__.py :* An empty file to initialise skill model
- *gae_models.py:* This would contain :
    - Class StoryModel: The main schema of the Story.
    - Class StoryCommitLogEntryModel for edit history.
    - Class StorySummaryModel: For the summary tiles to be displayed in a topic page summarising the stories in a topic.
    - Class StorySnapshotMetadataModel and StorySnapshotContentModel: These would be extensions of their respective parent classes in base_models

Skills

These are short, concise, global constructs that consist of a single concept card with worked out examples, followed by a series of questions testing the same concept. These are aimed at teaching the learners some specific concept that could be part of a larger topic, and hence are used as prerequisites for starting an exploration.

Schema for Skills

- **id**: Unique identifier for a skill.
- **Description**: A short concise description of the skill in less than 100 characters .
- **concept_card**: This would be a dict with the following two fields:
    - **Explanation:** This would be an explanation of how to apply the skill.
    - **Worked_examples:** This would a list of worked out examples for the skill.
- **misconceptions:** A list of misconceptions associated with a skill that could be tagged to answer groups of a question, so as to ensure that the question linked to the skill does indeed give targeted feedback for all known misconceptions for that skill. This would be modelled as a dict with the following values:
    - **Tag_name:** This would be the short name of the misconception that can be used by a question creator to tag an answer group to a skill.

    - **Description:** This would not be shown to the learner, but would serve as advice for creators on what the misconception is about.
    - **Default_feedback**: This would be the default feedback that would populate for any answer group that is linked to this misconception.

Schema for SkillCommitLogEntry

This model would be responsible for storing the log of all commits to skills. Every time, an edit is made to a story, an entry is made in this model.

- **User_id**: The id of the user making the commit.
- **Username**: The username of the user at the time of making the commit.
- **Skill_id**: the id of the skill that was updated during this particular commit.
- **Commit_type**: The type of the commit. Currently, this would contain one of ‘create’, ‘edit’, ‘delete’ or ‘revert’
- **Commit_message:** The message given by the user regarding the changes in this commit.
- **Commit_cmds:** The commit command dicts for this commit.
- **Version:** The version number of the skill after the commit.

Schema for SkillSummary

This model would be responsible for storing the summary data for a skill, to be displayed in summary tiles.

- **Description**: The description of the skill
- **skill_id:** The id of the skill of which this is a summary of.

Schema for QuestionSkillLink

This model would contain the questions that are waiting to be linked to a skill.

- **Question_id:** The id of the question to be linked to a skill.
- **Skill_id:** The id of the skill to which the question is to be linked to.
- **Review_status:** The status of the linkage between the question and the skill. This could be a string with values ‘Pending’, ‘Approved’ or ‘Rejected’, or it can be modelled as a number with 0 for pending, 1 for approved and -1 for rejected. The latter case might prove to be faster when searching for all questions with a particular status, later on, and hence, would be using that throughout the doc.
- **Difficulty:** The difficulty level of the question (for now, this is made same for every question, which can later be edited).

Files to be Added

The folder /core/storage/skill is to be created in which the following files are to be added to define the Skill Schema in the database.

- *__init__.py :* An empty file to initialise skill model
- *gae_models.py:* This would contain :
    - Class SkillModel: The main schema of the skill and its associated functions
    - Class SkillSummaryModel: The short summary data of the skill for use in displaying the related skill tiles in the topic page.
    - Class SkillCommitLogEntryModel for commit history.
    - Class SkillSnapshotMetadataModel and SkillSnapshotContentModel: These would just be extensions of the base class BaseSnapshotMetadataModel and BaseSnapshotContentModel with no extra functions, for retrieving the snapshots for commit history, to be used in getting history in the skills editor.

Implementation steps for creating a Skill, Topic or a Story in the backend

- **feconf.py**: New constants have to be added. Some of which are:
    - NEW_SKILL_URL: '/contributehandler/create_new_skill'
    - NEW_TOPIC_URL: '/contributehandler/create_new_topic'
    - NEW_STORY_URL: '/contributehandler/create_new_story'
    - SKILL_EDITOR_URL_PREFIX: '/skill_editor/create'
    - STORY_EDITOR_URL_PREFIX: '/story_editor/create'
    - TOPIC_EDITOR_URL_PREFIX: '/topic_editor/create'

- **main.py:** New routes are to be created to create a new skill, topic or story in the database as well as for viewing the skill, topic and story editor. These are:
    - */contributehandler/create_new_skill*: The class that this route calls would be a part of controllers/topics_and_skills_editor.py. A new class called NewSkill can be declared in this file which will call the required functions from skill_domain.py and skill_services.py in core/domain (which will also be created) to create the new skill.
    - */contributehandler/create_new_topic*: The class that this route calls would be a part of controllers/topics_and_skills_editor.py. A new class called NewTopic can be declared here which will call the required functions from topic_domain.py and topic_services.py in core/domain (which will also be created) to create the new topic.

- */contributehandler/create_new_story*: The class that this route calls would be a part of controllers/topics_and_skills_editor.py as the Create Story button is accessible from the topic editor page. A new class called NewStory can be declared here, which will call the required functions from story_domain.py and story_services.py in core/domain (which will also be created) to create the new story.
- */skill_editor/create/<id>:* This will call the SkillPage class in skill_editor.py, to initialize the editor with the data in the backend corresponding to the passed skill_id.
- */story_editor/create/<id>:* This will call the StoryPage class in story_editor.py, to initialize the editor with the data in the backend corresponding to the passed story_id.
- */topic_editor/create/<id>:* This will call the TopicPage class in topic_editor.py, to initialize the editor with the data in the backend corresponding to the passed topic_id.
- */skill/explorehandler/init/<skill_id>* : This will call skill_player.SkillHandler
- */questionskillhandler/init/* : skill_player.QuestionSkillLinkHandler
- */skill_editor_handler/data*: skill_editor.SkillHandler
- */topic_editor_handler/data*: topic_editor.TopicHandler
- */story_editor_handler/data*: story_editor.StoryHandler
- */skill_editor_handler/snapshots*: skill_editor.SkillSnapshotsHandler
- */get_questions/<skill_id>*: skill_editor.QuestionSkillLinkHandler
- */skill/metadata_search*: skill_editor.SkillMetadataSearchhandler

- **controllers/skill_editor.py:** In the classes defined below, the ones that are directly related to creating a skill or rendering the skill editor page are defined. Those would be similar for both **controllers/topic_editor.py** and **controllers/story_editor.py**..

  - Class SkillPage: Render the frontend html page when the url for creating a skill is called. This will fetch the global information about the skill (using the id) and render their respective html page for the editor using the function *self.render_template()*.
  - Class SkillHandler is to be created with the functions: _get_skill_data() which will return a dict of field values based on the id and this will be called by a get() function which will then return the json values to the frontend. This will also have delete() and put() functions for updating and deleting the skill.
  - Class QuestionSkillLinkHandler: This will have the get(skill_id) function, which when given a skill id, will search through the QuestionSkillLink class and return all questions linked to a skill, with their respective review_status.
  - Class SkillSnapshotsHandler: This will have the get() function, which will query skill_services.get_skill_snapshots_metadata(id), to get the history of edits to a skill.

- Class SkillMetadataSearchHandler: This will call summary_services.get_skill_metadata_dicts_matching_query() used for displaying a dropdown of related skills when adding the skill to an exploration.

As a test file for this, skill_editor_test.py can be created with some tests like:

- ➔ *test_access_skill_editor_page():* This will make sure that only admins can access the skill editor page.
- ➔ *test_editable_skill_handler_put_cannot_access():* This will make sure that non-admins cannot do a put request to the skill handler.
- ➔ *test_editable_skill_handler_put_can_access():* This will make sure that admins can do a put request to the skill handler.
- ➔ *test_publish_collection():* This would test the working of the skill publication process.
- ➔ *test_question_skill_link():* This will create some questions, assign it to a skill, and test whether the QuestionSkillLinkHandler does return the correct set of questions.

- **controllers/skill_player.py:**
  - Class SkillHandler: This will have the get(skill_id) function which will return the skill for the skill player using its id.
  - Class QuestionSkillLinkHandler: This will have a get(list(skill_id)) function which will search through all the QuestionSkillLink models and return a minimum of 3 questions such that all skills are represented at least once.

    Some tests that can be written for this file (in skill_player_test.py) are:

  - ➔ *test_refresher_skill_player():* First, a sample skill can be created with some questions and then, the SkillHandler class can be tested to make sure ir returns the correct skill.
  - ➔ *test_pretest_player():* Both of the above mentioned cases can be tested, by creating multiple skills and checking whether the pretests returned are correct.

- **controllers/topics_and_skills_editor.py:**
  - A new class NewSkill which will create a new skill with default values for all fields (skill_domain.Skill.create_default_skill()) with a randomly generated skill id (skill_services.get_new_skill_id) and finally saves it, would be defined here. After that, the skill_id of the newly created skill is returned.
  - Class TopicsAndSkillsEditorPage is to be defined this which does the job of rendering the topics and skills editor page.
  - Also, new classes called NewTopic and NewStory to be created for handling topic and story creation. NewStory is created here itself, as inherently, it is a part of the topic creation process.

In the test for this,, the above creation buttons can be tested to make sure they redirect correctly.

➔ *test_new_skill_id(), test_new_topic_id() and test_new_story_id():* The only thing that the post function in their respective classes would return is the skill ids and hence, those can be tested to make sure valid skill ids are returned.*:*

The following are the new files to be added related to the various functionalities of skills throughout the doc in *core/domain* but similar files are to be added for topics and stories as well.

- **skill_services.py:**
  - *get_new_skill_id*(): This will call the get_new_id() function in SkillModel (which is actually part of base_model).
  - *save_new_skill*(): This will call the _create_skill() function which will create and save the skill in DB. It will also call user_services.py to record the commit made by the user.
  - *_create_skill*(): This will create a new SkillModel object with all the relevant fields and save it to the DB. Also, incrementing the version of the skill. Also, in this function, the summary of the skill is created and stored in the SkillSummary Model.
  - *get_skill_by_id*(id): This would return the concept card and worked out examples of the skill, as well as search through the QuestionSkillLink class and select 3 random approved questions for a skill.
  - *get_pretests_from_skills*(list(skill_id)) : This would search through all the approved questions linked to all the skills and randomly select 1 from each skill, if number of skills is more than 3, else, it randomly takes 3 questions from the given 1 or 2 skills.
  - *get_questions*(skill_id): This would search through the QuestionSkillLink model and return all questions and their review_status linked to a particular skill
  - *get_skill_snapshots_metadata*(skill_id): Gets history of the skill by querying the get_snapshots_metadata() function in the SkillModel which will use the SkillSnapshotMetadataModel to get the results.
  - *update_skill*(): update the skill with changelists and update the snapshots.
  - *Get_skill_ids_matching_query()*: A list of skill titles is returned corresponding to given query.

    In the test for this, test functions can be written in separate classes to make sure each function works properly

  ➔ *test_no_errors_are_raised_when_creating_default_exploration*(): The skill creation process can be tested at once by calling exp_domain.create_default_skill() and making sure that no errors are raised.

  ➔ *test_retrieval_of_skill():* The *get_skill_by_id(id)* function can be tested to make sure it returns correctly.

- ➔ *test_retrieval_of_multiple_skill_versions():* In this a default skill can be created, after which updates can be done on this and making sure the skill does change with updates and the version returned is correct.
- ➔ *test_record_commit_message():* can test the *get_skill_snapshots_metadata()* function.
- ➔ Also, 2 more tests can be written to make sure all questions related to a skill are returned as well the pretests for a list of skills are returned.

- **skill_domain.py:**
  - class Skill would be present to handle all function related directly to the Skill model. The constructor of which will create a Skill object having all related fields and return it.
  - the create_default_skill() function would just call this constructor with all default values of the Skill.

  In the test for this, test functions can be to make sure that the interaction of the skill with the models are proper and that all the functions inside this work as expected.

- **Summary_services.py:**
  - *get_skill_metadata_dicts_matching_query*(): This would call *skill_services.get_skill_ids_matching_query*() and gets the list of ids of skills, from which the description of the skill is fetched.

Common files to be changed for adding the new schemas

- **platform/models.py:**
  - skill, topic and stories have to be added to the NAMES list to be enlisted as valid model names.
  - In the function import_models() in class _Gae, the above 3 models have to added to the if-elif statements to actually import the created gae_models.py file for each in core/storage.

Schema migration for explorations and states

The new skill model that is being created can be used to do two things in the frontend:

- Link from certain answer groups of an exploration so that the skill would be fetched when the user enters that answer group. This would be similar to the refresher exploration implementation (as far as backend is concerned).
  This would require a schema migration for states to remove the refresher_exploration_id field and replace it with skill_id field. Also, as questions are using the same states, another field should be added for tagged_misconceptions.

- As pretests for a skill. The skills required for the pretests are mentioned in the exploration schema by adding another field called ‘prerequisite_skill_ids’, which would contain a list of skill ids that would be considered for the questions in the pretests for an exploration.
  Also, another field is to be added to the exploration to record whether an exploration is part of a story or not. This field (‘story_id’) will contain the id of the story that it is part of (if applicable). These would require a schema migration for exploration.

Topic and Story Editor

- In the frontend, in the dropdown, there would be a link to an Topics and Skills Editor Page’, (which would be visible only when the logged in user is an admin). This would redirect to a page where all the topics and skills currently in the database, would be shown as their summary tiles. Here, clicking on a topic would open ‘/topic_editor/<topic_name>’.
  This page would also have the ‘Create Topic’ and ‘Create Skill’ buttons which opens their respective editors.
  The ‘Create Story’ button would be shown when a topic is selected from this page to add a story directly to a topic.

Related Files

  - ➔ *components/top_navigation_bar/top_navigation_bar_directive.html:* Another element to be added to the dropdown called topics and skills editor
  - ➔ A folder (topics_and_skills_editor) to be created in /pages, whose implementation will be identical to the existing library page, wherein summary tiles for a fixed number of topics and skills in the database would be shown.

- This page would have the Create Topic and Create Skill buttons. (The Skill creation part would be done in the next milestone)

Related Files

  - ➔ *pages/topics_and_skills_editor/topics_and_skills_editor.html:* A button for creating the topic and skill is to be added, which when clicked calls a function createTopic() or createSkill() in *topics_and_skills_editor*.js in the same folder.

➔ *pages/topics_and_skills_editor/topics_and_skills_editor.js:* The function createTopic() is to be added to initiate the topic creation process, which will call TopicCreationService.createNewTopic().

➔ *pages/topics_and_skills_editor/TopicCreationService.js:* Create this file which has the function createNewTopic() which does a POST request to the backend (/contributehandler/create_new_topic) to actually create the Topic. This returns the id of the created topic. Now, the url is redirected to '/topic_editor/create/<id>' using UrlInterpolationService.js. This url is then, redirected by the backend to render the html page at topic_editor folder to show the initial topic editor window.

➔ *pages/topics_and_skills_editor/StoryCreationService.js:* Create this file which has the function createNewStory() which does a POST request to the backend (/contributehandler/create_new_story) to actually create the Story. This returns the id of the created story. Now, the url is redirected to '/story_editor/create/<id>' using UrlInterpolationService.js. This url is then, redirected by the backend to render the html page at story_editor folder to show the initial story editor window.

- **Story Editor:**
    - In the first page of story editor, there can be a single RTE, which will input the story notes (which will describe the characters, main storyline, and setting).
    - Then, below this, an Add Node button would be present, clicking which would show the fields required for the node of a story below it. These fields are: acquired_skill_ids, prerequisite_skill_ids, exploration_id and destination_node_id.
    - These constituent nodes will be visible below in a table format, with an 'X' beside it for removing the node.

Related Files

➔ */pages/story_editor:* This new folder has to be created which will contain the various files related to the story editor.
➔ */pages/story_editor/story_editor.html:* This will be the HTML file that will be rendered when /story_editor/create url is called.
➔ *story_editor/StoryEditor.js:* This initializes the story editor page by calling *StoryEditorStateService.loadStory().* fetching all the data from the backend about the story to be edited.
➔ *story_editor/StoryEditorStateService.js:* Apart from the functions below, this will also have functions to search for exploration ids and skill ids to be displayed in the dropdown in the story editor to make it easier for a user to add a node to a story.

- *loadStory():* This loads the topic information for the editor. This can be done through a service (EditableStoryBackendApiService.fetchStory()), which will get the data from the backend using the url /story_editor_handler/data/<story_id> and initialises the story editor page with these values.
- *publishStory():* This will update the story in the backend, with the contents from the editor via changelists. A function EditableStoryBackendApiService.updateStory() can be declared for this.

- **Topic Editor:**

  - In the first page of topic editor, there can be a single RTE, which will input the topic metadata (like the introduction and content html to be displayed in the landing page). There will also be a text box to enter the topic title here.
  - There would be an 'Add Story' button on the header for this, which would open up the story editor in a different window and add that story to the current topic. The list of current stories in the topic would be shown as a summary tile list clicking which the story editor for that story can be shown.
  - There can also be text box above this, to manually add a story using its id.
  - Below this can be a text box, with an Add Skill button beside it, so that each skill can be added to the list, through the text box and clicking the button. The skills part of the topic will be visible under it, with an 'X' beside it to remove the skill from the topic.
  - Once, these are added, the topic can be published.

Related Files for topic creation

➔ */pages/topic_editor:* This new folder has to be created which will contain the various files related to the topic editor.

➔ */pages/topic_editor/topic_editor.html:* This will be the HTML file that will be rendered when /topic_editor/create url is called.

➔ *topic_editor/TopicEditor.js:* This initializes the topic editor page by calling *TopicEditorStateService.loadTopic().* fetching all the data from the backend about the current topic.
This will also have the createStory() function which will create a story (with default parameters) and add it to the topic as well as open the story editor.

➔ *topic_editor/TopicEditorStateService.js:*
- *loadTopic():* This loads the topic information for the editor. This can be done through a service (EditableTopicBackendApiService.fetchTopic()), which will get the data from the backend using the url /topic_editor_handler/data/<topic_id> and initialises the topic editor page with these values.

- *publishTopic():* This will update the topic in the backend, with the contents from the editor via changelists. A function EditableTopicBackendApiService.updateTopic() can be declared for this.
- This will also have *fetchStorySummaries(storyIdList)* and *fetchSkillSummaries(skillIdList)* to get the stories and topics linked to a particular topic.

As tests for the above, the spec files that should be created are:

- *EditableTopicBackendApiServiceSpec.js*: In this, the various functions in the service like fetchTopic() and updateTopic() can be tested, after creating a sample topic to make sure that the functions work as expected.
- *EditableStoryBackendApiServiceSpec.js*: Similar to the above, test functions can be written for fetchStory() and updateStory() as well.

Notes: For topics, when a story is removed from a topic, the topic_id field for that story should be made false or null, and when an exploration is removed from a story, its story_id field should also be made false or null.
Also, if a topic is deleted, all stories that constitute the topic should have their topic_id field nullified. Similarly, with explorations for stories. This is because, only an exploration, part of a story, which is part of a topic is allowed to have refresher skills and prerequisite skills.

Timeline

Our final exams for the semester get over by May 8th, so I can start work before May 14th, if required.
Also, the review and subsequent review changes of one section would be done, either in its period itself, or can be done in the next period, along with that section.

- Create backend models for topics, skills and stories And perform schema migration for states and explorations - May 14 - 20
- Write all the required backend functions for skills, topics and stories (includes those for skill editor and player, and topic and story editors) - May 21 - 30
- Implement the story editor - May 30 - June 5
- Implement the topic editor - June 5 - 11

June 12 - 15 would be review time for the overall milestone and final changes.

Milestone 2: Editor interface for skills and library pages for skills and topics

In this milestone, the skill editor window with all its functionalities is created. From the skill editor window, a creator would be able to:

- Add a concept card, a list of misconceptions and a list of worked out examples for the skill.
- View the questions that have been requested to be linked to the skill.
- Approve question to be part of a skill, or reject it from being part of the skill.
- View the edit history of the skill.

There is no clear cut way of testing this milestone apart from the check that can be done to see if the skill has been created, as the skill player part of the project is done in the next milestone. One way, the check can be done, is to add the newly created skills as prerequisites or acquired skills to the topics and stories that were created in the previous milestone.

Skill Creator Window

In this section, the skill creator window would be implemented, with basic functionalities like history tab, as well as approving or rejecting questions to be part of a skill.

Various parts of the skill creator window with its technical aspects

- In the topics and skills editor page, a Create Skill button could be present which, when clicked will open up the Skill Editor.

Related Files:

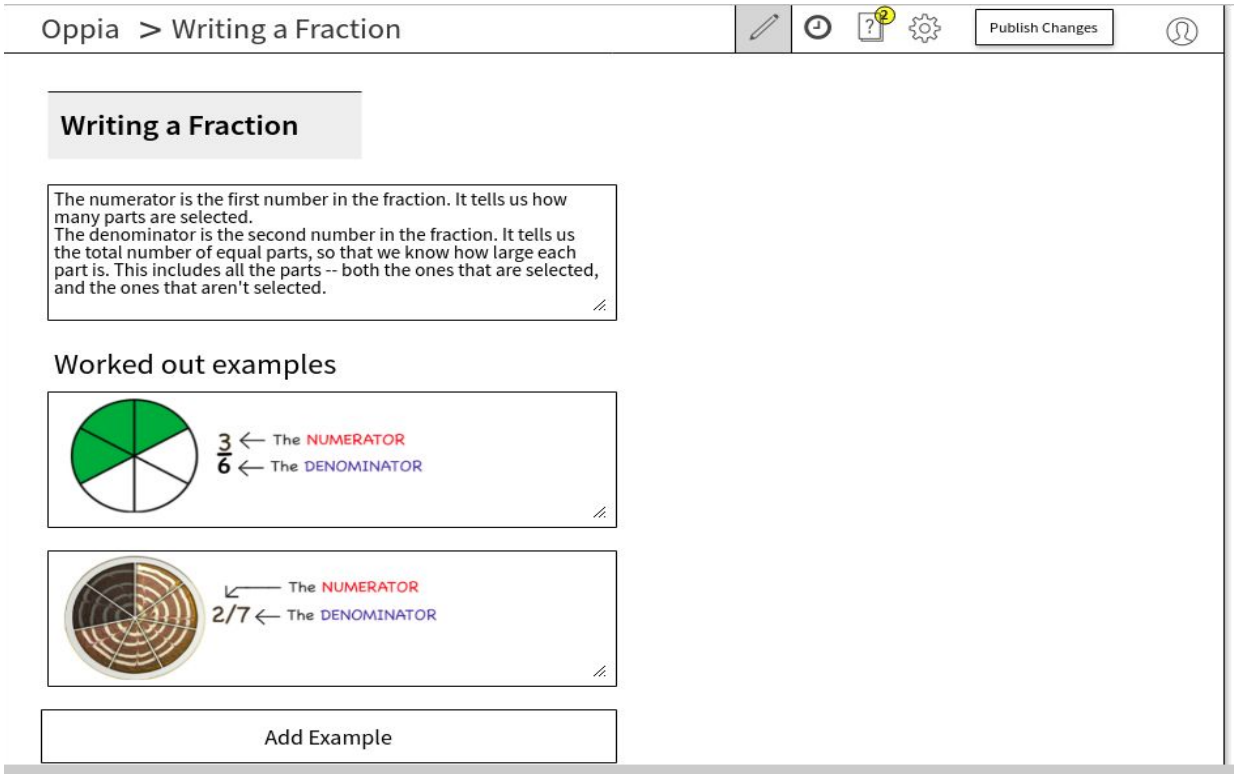
➔ *pages/topic_and_skills_editor/topic_and_skills_editor.html:* A button for creating the skill is to be added, which when clicked calls a function createSkill() in topicsAndSkillsEditor.js in the same folder.

➔ The function createSkill() is to be added in topicsAndSkillsEditor.js to initiate the skill creation process, which will call SkillCreationService.createNewSkill().

➔ *pages/topic_and_skills_editor/SkillCreationService.js:* Create this file which has the function createNewSkill() which does a POST request to the backend (/contributehandler/create_new_skill) to actually create the Skill. This returns the skill_id of the created skill. Now, the url is redirected to

'skill_editor/create/<skill_id>' using UrlInterpolationService.js. This url is then, redirected by the backend to render the html page at skill_editor folder show the initial skill editor window.

Skill Editor



- The skill editor page will be a basic skill editor interface, with a concept card tab, settings tab, history tab and questions tab. The first card that is visible will be the editor window for the concept card where the creator can give an introduction to the skill, define the concepts related to the skill and have worked out examples.

Related Files

➔ */dev/head/pages/skill_editor:* This folder would contain all the files related to the main skill editor page. This will have the skill_editor.html file which forms the basic framework to place the all directives in.

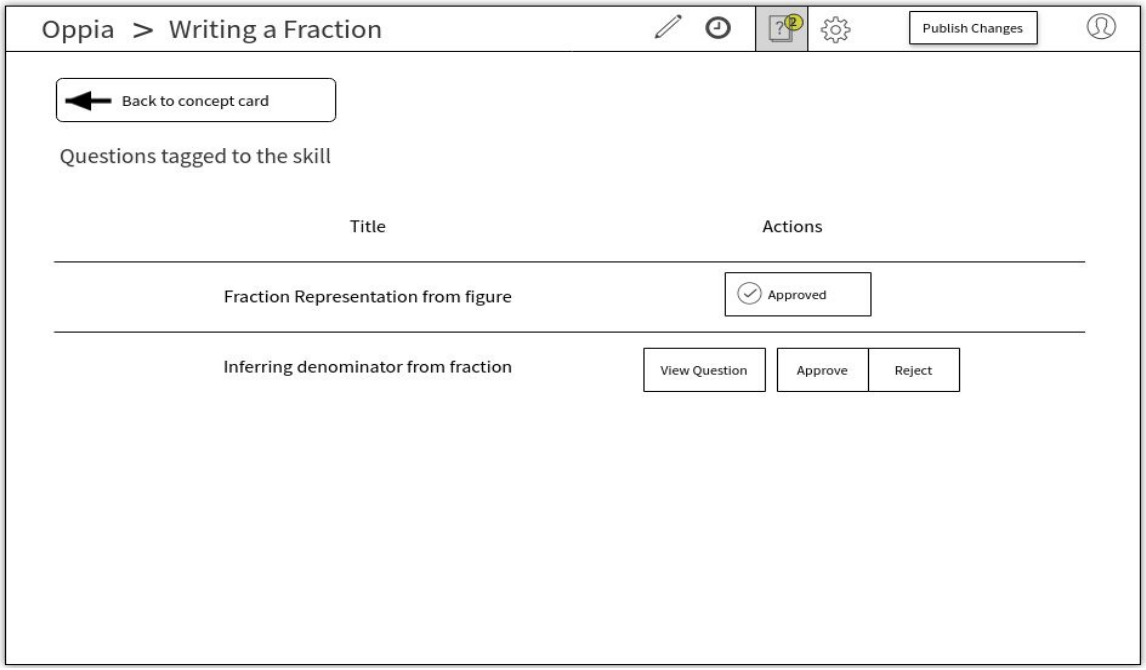
➔ *skill_editor/SkillEditor.js:* This initialises the skill editor page by fetching all the data from the backend about the current skill. This can be done through a service (EditableSkillBackendApiService.fetchSkill()), which will get the data from the

backend using the url /skill_editor_handler/data and initialises the skill editor page with these values.

➔ *skill_editor/concept_card_editor.html:* This will contain the RTE for adding the concept card to the skill. This will be similar to the existing file state_content_editor_directive.html in pages/exploration_editor/editor_tab.
There would also be another RTE below this to add the worked out examples for the skill. There would be an Add Example below it, which will open another RTE with which multiple worked out examples can be added.

➔ *skill_editor/ConceptCardEditorDirective.js:* This file will have all the functions related to concept_card_editor.html. The main action being saving of the RTE contents. This can be done by creating functions similar to that in StateInteraction.js in exploration_editor/editor_tab.

Questions Tab



- For adding questions, another icon can be displayed on the header to switch into questions_tab which will have a list of questions waiting to be linked to the skill (and those already linked as well). The number of pending questions can be displayed as a badge on the link.

Related Files

➔ This is added as a button in skill_navigation_directive.html. This file will also have a button for editor_tab.

- Here, all the questions that have been requested to link to this skill will be shown. The creator can view the question, approve or reject the question. Rejecting the question will remove it from this list. For rejection, a modal would pop up confirming rejection. If a question is approved, its review_status in the class QuestionSkillLink is changed to 1 (accepted) or -1, if rejected.

  Currently, as a question editor is not present yet, the owner of the question wouldn't be knowing when the question is approved or rejected, but once that has been implemented, a separate tab could be made in the questions editor which would list out all the skills that the question is linked to and if any skill creator rejects the question for a skill, some visual identifier could be used to inform the question creator.
  As a question editor is not currently present, this is deferred until that editor is made.

Related Files

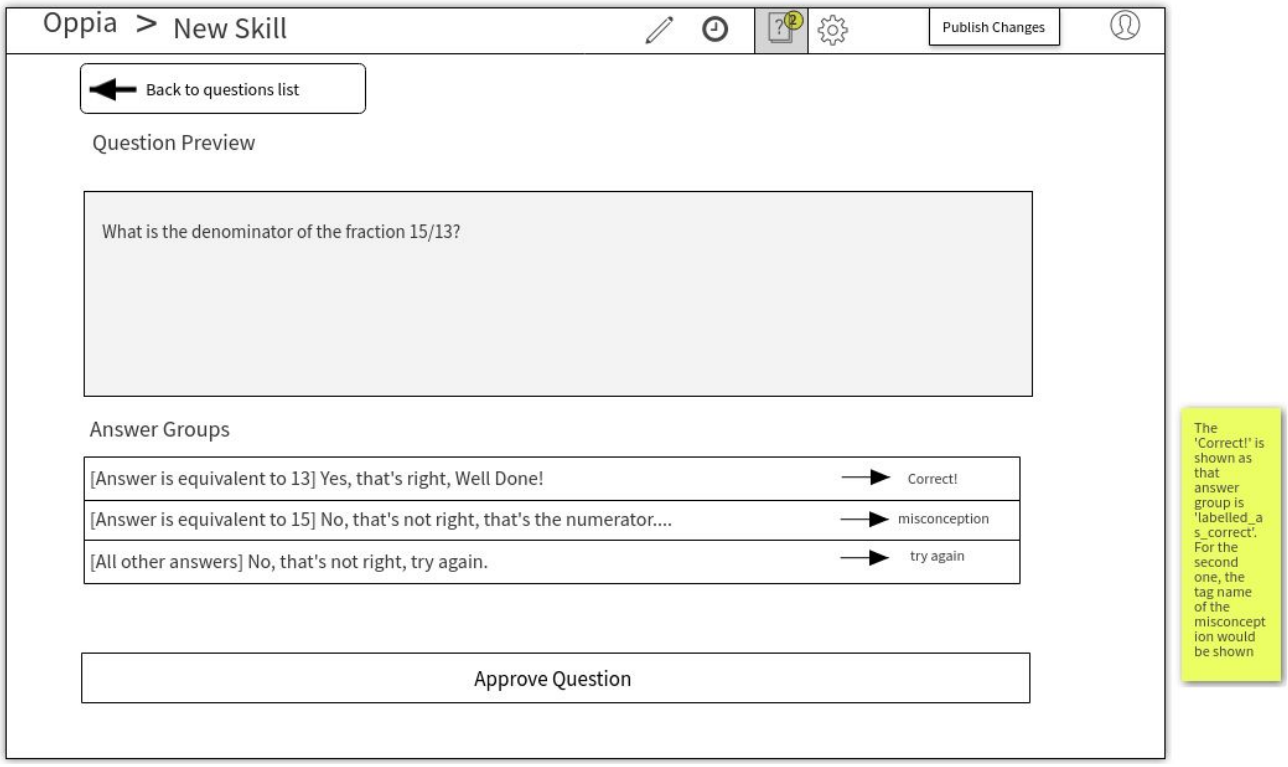
➔ This page is modeled by another directive, QuestionsListDirective.js, and its corresponding template file - questions_list_directive.html (in skill_editor folder).

➔ *questions_list_directive.html:* This will have a 'Back to Concept Card' button at the top and the questions list would be displayed here. The questions list would be fetched from the backend and its titles would be displayed in a table format.

➔ *QuestionsListDirective.js:* The questions are fetched from the backend in this file. To get the questions, a call can be made to '/get_questions/<skill_id>', in which case, the backend functions would search the QuestionSkillLink model and return all the question ids that are linked to the particular skill_id.
As, a question is approved or rejected, PUT requests can be made to the backend for changing the review_status of that particular QuestionSkillLink object to 1 or -1, as required. Thus, adding a question to a skill does not require a publish of the skill, as the backend itself is updated, when a question is approved.

This approach to questions is also a scalable approach, as when a question editor is made, to find the skills that the question creator had requested to be linked, instead of going through the entire skill database, only the lighter QuestionSkillLink model has to be search through, in which there is equal relevance to both the skill and its questions.
As this search occurs in other places also (like skill player and skill editor), this model of using QuestionSkillLink would be a good choice.

View Question window (part of the questions tab itself)



- Viewing the question will open the question details in the same page (as shown in mock) in a non-editable format. There is a Back to Questions List button in this page to return to the questions list page.

Related Files

➔ *questions_info.html:* This file will be very similar to the existing state_editor.html in exploration_editor/editor_tab, except that the answer groups and the interaction will not be editable

➔ *QuestionInfo.js:* This will contain all the functions related to questions_info.html. The main job that this file would do is fetching the state from the backend and rendering the interactions in a non-editable format for viewing in its corresponding html page.

- There is also a Settings tab, identified by the Gear icon in the header, in which the creator can add the list of misconceptions associated with the skill as well as the short description of the skill.

Related Files

- ➔ *settings_tab.html:* This file will have 2 fields, a text box which keeps on adding misconceptions to an array (the contents of which will be displayed below this field) and a description text field where the creator can add a description of maximum 50 characters.
- ➔ *SettingsTab.js:* This file will do the job of actually updating the fields for the skill and creating a changelist for those updates.

- There is also a History tab, identified by the time icon in the header, in which the creator can see the history of edits that were done to the skill.

Related Files

- ➔ *history_tab.html:* This file would have a repeated *div* showing the various updates that were done to the skill, and the commit message that was given for each.
- ➔ *HistoryTab.js:* This file will do the job of fetching the history of edits to the skill, by querying the SkillSnapshotMetadataModel and SkillSnapshotContentModel in the backend using the url '/skill_editor_handler/snapshots/<skill_id>'.

- After the creator is done editing the skill, the publish changes button becomes active. The publishing will be handled similar to explorations where the creator will be asked to add a description and misconceptions before publishing. After which a PUT request will be given to '/skill_editor_handler/data' by EditableSkillBackendApiService.js, which will be directed to the SkillHandler class in editor.py where the update is actually done.

Related Files

- ➔ *skIll_editor/SkillEditor.js:* A publishSkill() function can be defined, which checks if all the required values like misconceptions, worked out examples and description is provided or not, and then calls EditableSkillBackendApiService.updateSkill(). The parameters passed to backend to update the skill will be in the form of a changelist, which will be updated as concept card or misconception is added in the editor. The questions will not be part of a publish, as each question statuses were updated as they are accepted or rejected in the QuestionSkillLink class itself.
  Another service could be created for handling the changelist creation and updation as the creator updates the skill in the editor.

Some tests that can be added in the skill creation process are:

- ➔ *EditableSkillBackendApiServiceSpec.js*: Similar to topics as done in milestone 1, this could have tests for fetchSkill() and updateSkill().
- ➔ *QuestionListDirectiveSpec.js:* This file would test the return of questions linked to a skill and proper updation on approving or rejecting question.

More tests can be added, as functions are created in the service files declared along the skill creation process.

Linking the skill to an exploration

In both the methods mentioned below, the field for adding a skill id will only be visible if the logged in user is an admin and if the exploration being edited is part of a topic.

Related Files

- This check of whether the exploration is part of a topic can be made when the exploration is loaded itself (in the init() function in exploration_player/PlayerServices.js), in which the story corresponding to the story_id field in the exploration (if it exists) can be checked (through a call to the backend) to see if it is part of a topic. If so, a global variable (or one that is restricted to the main editor directive) can be set to show that exploration is part of a topic, and based on this, the following can be done.

As Pretests

An exploration can be given pre-requisite skills in the settings tab in the exploration creator window or just before publishing the exploration, another not-required field would be shown in the modal that pops up for giving the prerequisite skills for the exploration.

Related Files

- Another factory ExplorationPrereqSkillsService is to be created in exploration_editor/ExplorationPropertyService.js having properties similar to that of ExplorationTagsService (as both are a list of values).
- Also, the showPublishExplorationModal() function in exploration_editor/ExplorationSaveService.js has to be edited to add the above mentioned prerequisite skills field in the modal (though it need not be added in the isAdditionalMetadataNeeded() function in the same file as this field is not a required field).

- Another field has to be added to exploration_editor/settings_tab/settings_tab.html to input the prerequisite skills in the settings tab of the exploration itself.
- The corresponding fields in SettingsTab.js (in the same folder) is to be updated also to handle the prerequisite skills field.

As refresher skills, attached to an answer group

In addition to the above, skills can be associated with a card of an exploration also, similar to the way refresher explorations work now.

The creator can attach a skill to a specific answer group for a question. When this is done, if the user gives an answer corresponding to that answer group, the skill would be fetched and shown to the user (as mentioned in the skill player part of the doc).

Implementation

- This can be done in a similar way to which existing explorations are added to a collection, in which, as skill description is being typed, a dropdown a list of existing skills with that word in the description would be shown so that the user can add a skill to an answer group effectively.
- For this, a SearchSkillsBackendApiService.js can be created in /dev/head/domain/skill, which will call '/skill/metadata_search?q=<query>'

Timeline

- Create the skill editor window
  Concept card editor - June 15 - 18
  Settings and history tab - June 19 - 24
  Questions tab - June 25 - July 1
- Add functionality to link skills to explorations - July 1 - 7

Same as before, the rest of the time can be for final review and changes.

Milestone 3: Implement the learner experience with skills

In this milestone, the skill constructs that were created will be used for enhancing the learner experience. This is implemented in two ways:

- As pretests for an exploration, in which, if a list of prerequisite skills is given for an exploration, the learner would have to take a pretest before the actual exploration, testing those skills and after he passes this, the exploration starts.
- The skills would also replace the existing refresher exploration functionality, wherein, instead of redirecting to another exploration, the concept card and 3 questions of the skill, would be played through as a part of the exploration itself, thus making the entire ‘refresher experience’ more streamlined.

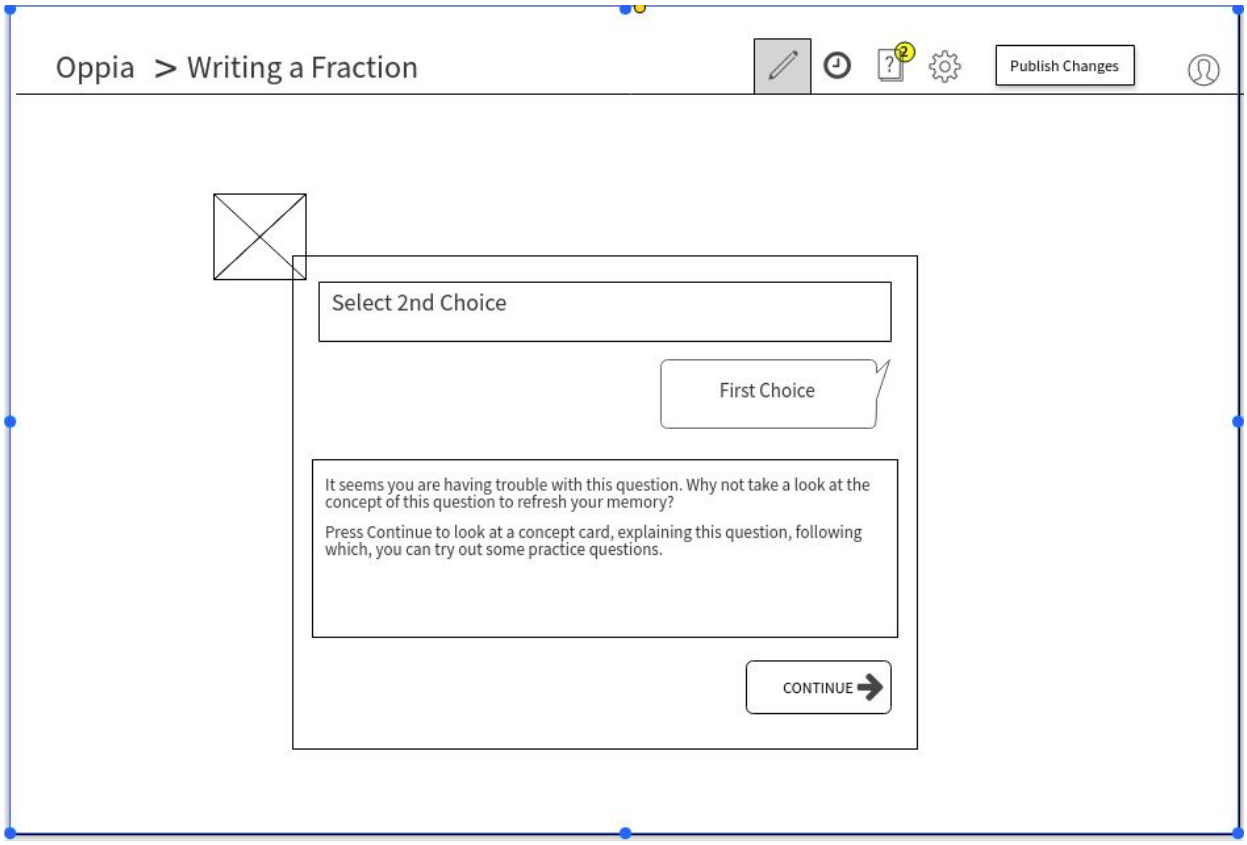
The outcome of this milestone will be visible to the learners and can be tested with the existing Fractions collection. For the refreshers, the skills created in the previous milestone can now be used, replacing the refresher exploration in the frontend.
Also, pretests can be added to any of the above explorations.
Note that these can be done only if the exploration is part of a story, that is part of a topic.

Invoking Skills in Explorations

This section explains the different ways a learner will get to experience skills as part of his/her exploration playthrough.

Redirecting to skills in between explorations

When an answer group for a question in the exploration is given by the learner such that the redirection to skill is triggered, a prompt can be given to the user that he/she is about to do a skill in the form of a feedback from Oppia (as shown in mock below) and the player would shift to a revision mode, in which, initially the concept card corresponding to the skill would be fetched and displayed, following which 3 questions corresponding to the skill would be shown to the learner. If the learner successfully completes this question set, then he/she will be redirected back to the main exploration and it will continue.



Related Files

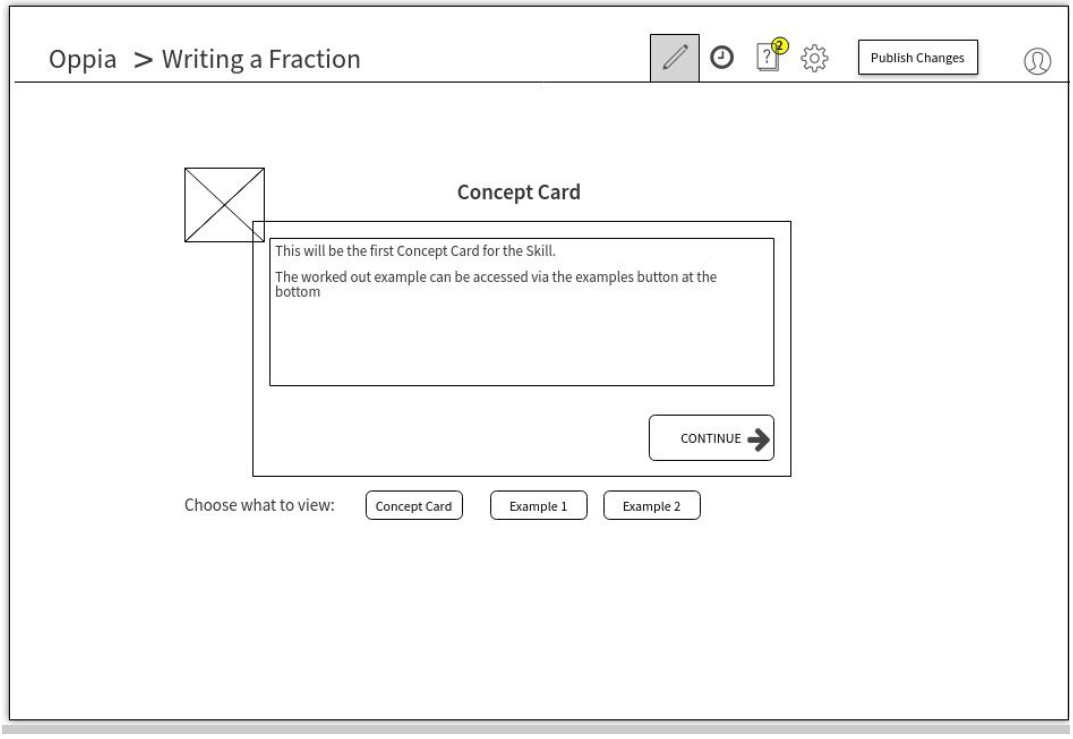
- *exploration_player/ConversationSkinDirective.js:*
    - **submitAnswer():** The redirection to skills can be done by adding on the already existing code for refresher explorations. If the current answer group is found to have a skill_id field, then the above feedback can be made to be displayed to the learner as on Oppia feedback. After which, this skill_id could be passed to ReadOnlySkillBackendApiService.fetchSkill() and the createFromBackendDict() function to get the concept card for the skill, following which, a query can be made to QuestionSkillLink model (ReadOnlySkillBackendApiService.fetchQuestions()) to get three approved questions for the skill.

      Then, similar to pretests (as explained below), another variable $scope.isRefresher can be set to true and ExplorationPlayerStateService.setSkill() can be called which will set a local variable in it (called Skill). The outcome management for each question, would be similar to pretests, the difference being after the last question, the next card in the exploration has to be loaded.

Also, as the initial card to be displayed will be the concept card with the list of worked out examples, when loading this card, as the backend model only contains the contentHtml for the above, when interactions are taken, the interaction id for the Continue Button interaction can be passed to the getInteractionHtml() and getInteractionInstruction() functions so that the content of the concept card would be rendered in a continue button interaction.

After this, the _addNewCard() will be called with the parameters of the skill cards.

Then, in the same function (for subsequent answer submissions in the skill), if $scope.isRefresher (this check can be made together with isPretest) is true, only the skill construct would be called on to get the new card and that way the skill would be played through.

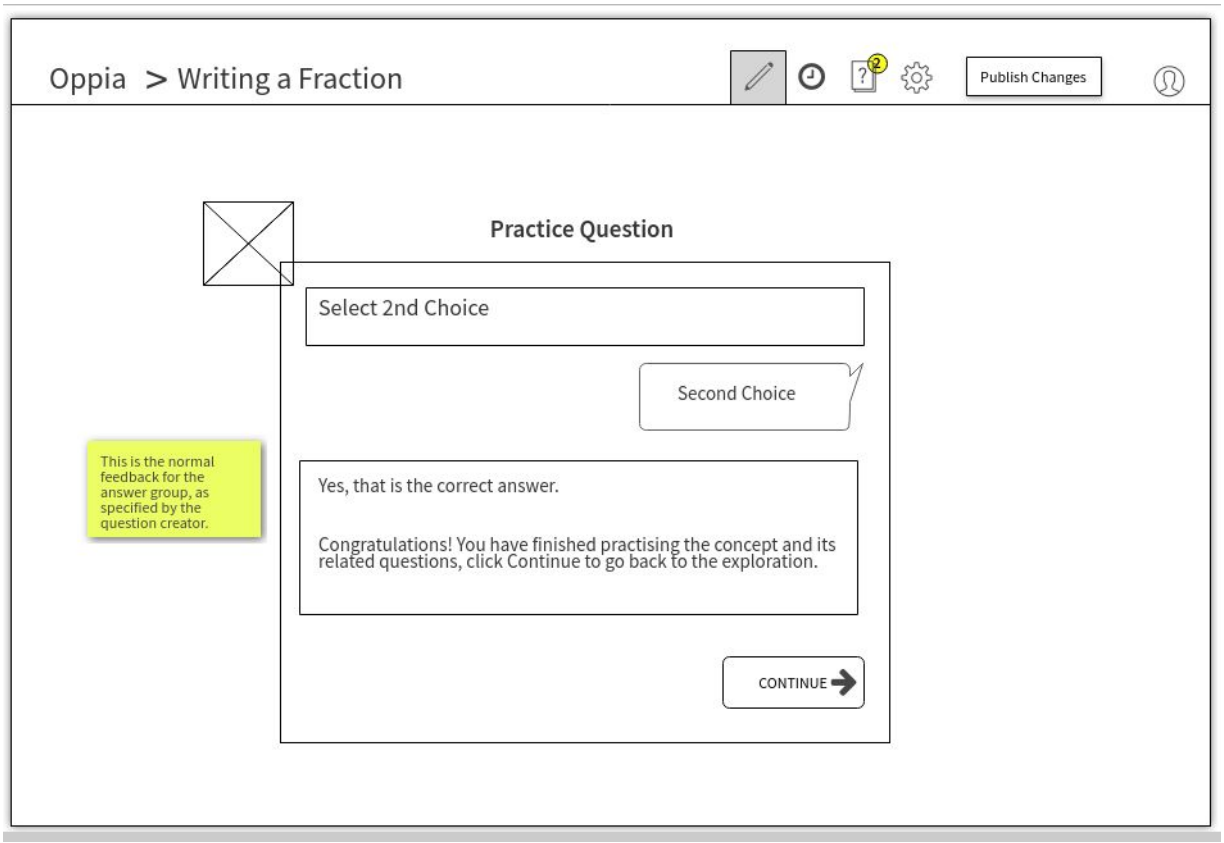
- 

This would be the first concept card view, wherein, the user can see some worked out examples by clicking on example 1 or example 2, and the title would change from ‘Concept Card’ to ‘Example 1’ and so on, clearly telling the viewer what is being shown. Also, not showing the worked out example initially itself could encourage the viewer to understand the concept thoroughly before looking at the examples or questions.

Related Files

➔ To get this, conversation_skin_directive.html has to be edited to have these buttons, and the corresponding directive (ConverstionSkinDirective) should have functions that view the respective contents. These can be obtained from ExplorationStateService, as the skill contents were stored there, when they were loaded.

- In the end, the isRefresher variable would be set to false and thus, normal exploration play would resume. In the below mock, the first line is the feedback, as set by the question creator, the lines following that can be appended to the feedback and shown to the learner to inform him/her that exploration play is about to resume.



Some tests that can be added for these changes are:

- *ReadOnlySkillBackendApiServiceSpec.js:* Test functions can be added to this file to make sure that.*fetchSkill(), fetchQuestions() and fetchPretests()* successfully fetches the skill and pretests from the backend
- *SkillObjectFactorySpec.js:* Tests can be added to make sure the correct fields are returned after converting from the backend dict.

As Pretests before an exploration

In this implementation, before the exploration starts, if the creator of the exploration had set any prerequisite skills for the exploration, the questions related to those will be played as pretests for the exploration. These have to be successfully completed for the learner to start with the exploration.
If the number of prerequisite skills are greater than 3, then one question from each skill will be chosen so that every prerequisite skill can be tested at least once, else more than 1 question from a single skill would be chosen such that a total of 3 questions are shown in the pretests.
Also, since in the below implementation, the loading of new card is still done by the _addNewCard() function in ConversationSkinDirective, the history of pretests would be stored in PlayerTranscriptService. (same for refreshers also)

Related Files

- */dev/head/domain/skill/ReadOnlySkillBackendApiService.js* : This file will be used to obtain the backend dict corresponding to a skill (fetchSkill()) as well as the questions for pretests corresponding to a list of skill ids (fetchPretests()), using the rules mentioned above.

- */dev/head/domain/skill/SkillObjectFactory.js:* This file will have the Skill.createFromBackendDict(), which will return the actual skill once the skillBackendDict is passed (this will have an internal call to StatesObjectFactory.createFromBackendDict() to get the actual states corresponding to the exploration). This file will also have the Skill.prototype functions to get the states corresponding to a skill or pretests anywhere else in the program.

- *exploration_player/PlayerServices.js:*
    - **init():** Just after the exploration is fetched from the backend, if prerequisite_skills are specified, then ReadOnlySkillBackendApiService.fetchPretests(skillIdList) can be called to obtain the list of questions (the questions would be obtained as state objects), after which the Skill.createFromBackendDict() function can be called, whose return value will be stored in a local pretests variable.
      After this, these states would be iterated through to add the statename for each question as ‘question1’, ‘question2’ etc.
    - **submitAnswer():** When, this is called from ConversationSkinDirective, another variable can be passed to check pretest mode and if so, the condition for the various outcomes can be as follows:
        - If labelled_as_correct, take next question from pretests variable in *ExplorationPlayerStateService.*

- If answer group has a tagged misconception, the skill linked to that question should be loaded, like that done with refreshers.
- If neither of the above, it is the default outcome.

- *exploration_player/ExplorationPlayerStateService.js*: Another variable called pretests, can be created here, along with the functions getPretests() and setPretests() as the states corresponding to the pretests are still part of the current exploration playthrough.

- *exploration_player/ConversationSkinDirective.js:*
  - **initializePage():** The ExplorationPlayerStateService.setPretests() can be called just after setExploration() is called, after which, _addNewCard() would be called with state parameters of pretests. Also, a $scope variable called isPretestMode would be set to true.
  - **submitAnswer():** Another else-if condition to be added (already there are 2 for next card loaded or outcome is same card itself), to check if in pretest mode. If so, the checks for the next card should be done with SkillObjectFactory functions.

- Thus, the initial states that will be called for playing would be that corresponding to the question, and after the questions are done, once the nextStateName is found to be exploration.initStateName, normal exploration play can be resumed.

Timeline

- Add refresher skills functionality - July 13 - 23
- Add the pretests functionality to explorations - July 24 - August 2

Finally, this last period (August 2 - 6) can be for the final review for this milestone as well as an overall check for the entire project.

Future Projects

- The topic and story player can be implemented, after which the collections construct can be taken down by moving all existing collections to stories and general topics.
- Also, statistics can be implemented for number of skill redirections, pretest fails etc.
- Once the questions frontend is done, misconceptions of a skill can be tagged to question answer groups.
- Finally, an independent skill player can be made, with which a learner can just go through a skill at his/her leisure, not as part of an exploration.

Summer Plans

Timezone

I would be in India throughout the duration of summer and hence would follow the Indian Standard Time (GMT +5:30).

Time to commit for the project and other obligations

At the beginning of summer, for about a week, I would be conducting some sessions in summer classes in our college. On these days, I would be able to put in around 5-6 hours a day on working in the project.
After this, it is semester break for us and thus, I would be able to put in at least 7-8 hours a day, which could be more if the project demands it.
On weekends also, I'll put at least 4-5 hrs, which could also be more if the project requires it.
Our semester starts on August 1st, but it is not a heavy semester for us and in the beginning, not much work would be there also. Hence, during this time, I would be able to put 4 - 5 hours on weekdays, depending on the day and on the weekends, I could put more time (6-8 hours), to make up for any pending work.

Communication

Contact Information

Full Name: Akshay Anand
My phone number: +91 9446024519
Email ID: akshayanand681@gmail.com
Github Profile: https://github.com/aks681

I am active on Gitter, Whatsapp and Hangouts. I also check my mail regularly and hence can be communicated by any of the above methods

Communicating with mentor

I plan to provide periodic updates on my progress to the mentors via Hangouts or Gitter. Over my time with Oppia, I've seen that many mentors are active on the above two platforms and

hence I would be using those. Also, a video call can be done every week or so, to discuss more about the developments in the project.