GSOC-2018 PROPOSAL

Project Name

Visualizing learner playthroughs

Reason for interest in Oppia

I, at a fundamental level, really like the concept of Oppia and believe in community driven learning. I started my journey with Oppia 1 year ago when I was browsing through the organisations for GSOC 17, and I really liked the work that was being done here. Couple that with my familiarity with the technology being used to build the project, Oppia felt like the right place to be.

What interests me about this project and why do I think it's worth doing

Oppia has two major infrastructural sides to it - the creator view and the learner view. The lessons created by the creator are subsequently used by multiple learners across the globe. The framework that connects the creator to the learners is the Statistics framework, which makes it extremely important since this is the only way for a creator to analyze performance of learners in the lesson content, and to make it better for the learners.

Playthroughs, in particular, are a new perspective to this statistic visualization concept. So far, the creators got statistical data in the form of numbers and counts of all the learner actions. Sometimes, this might not be enough as this does not account for problematic issues an individual learner might face in an exploration. Identifying useful playthroughs provides a solution to this.

I think this will be a very fun project to work on, since it will be used at scale by all creators, and also because it is really satisfying to know that my contribution makes a huge impact on users.

Prior Experience

The core skills required to work on this project are Python, AngularJS and HTML/CSS. I have been exposed to working with Python for a few years now and I am extremely

comfortable working on projects involving Python. I have been working on my front-end skillset recently and have some experience of working with AngularJS as well.

I have been leading the Statistics team of Oppia since August, 2017. As the team lead, I designed the new Statistics framework (which is currently being used in production) and also implemented a lot of the major parts of the framework.

I was also a member of the ‘Answer Classification’ project during the summer of 2017, and was an active contributor to the project, especially during the migration of the training process from GAE to a dedicated VM.

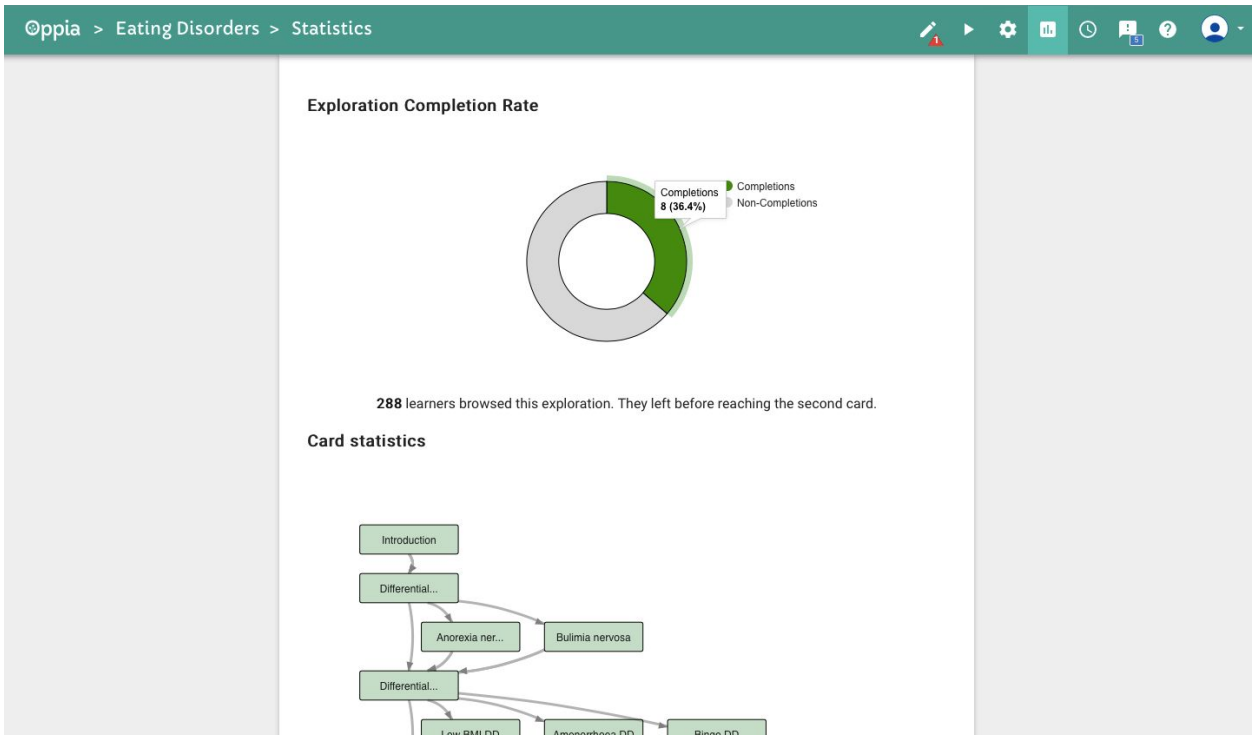
PRs

The complete list of PRs I have made to the organisation can be viewed at: Pranav's Pull Requests.

Playthrough visualization in Exploration Editor UI

Representing learner playthroughs is a very crucial part of this project. The representation needs to be simple, yet efficient. The creator should be able to traverse the lesson journey the learner was involved in by just viewing the playthrough. To achieve this purpose, a text list of learner actions will be ideal. Each learner action will be some kind of learner activity during the lesson.

The current statistics tab design:



Given below is a link to the proposed design for representing the playthroughs in the Exploration Editor UI.

https://docs.google.com/document/d/1RhaFmoA6FI6Xpo1KXUgMmwZ13GprOKePJfjNcRRpmT8/edit

Definitions and Creator Workflow

This section will focus on defining the various types of playthroughs the creator will be shown and also the procedure to resolve playthroughs.

Early quit playthrough:

This playthrough will encompass any learner that quit the exploration very early (specific values are discussed below). Also, this will count only learners who have actually started an exploration and will not be counting the learners who just browsed the exploration. To resolve a playthrough of this type, we would encourage the creator to look at all states from the start of the exploration leading up to the state at which the learner quit, and let the creator deduce if the initial parts of the exploration has any elements that a learner might find tough to understand.

Multiple incorrect playthrough:

This playthrough will encompass any learner that quit the exploration after answering multiple incorrect answers to the same state. To resolve a playthrough of this type, we encourage the creator to look at the states that explain the concept of the question at which the learner failed to get through.

Cyclic playthrough:

This playthrough will encompass any learner that visited a set of states repeatedly, basically going around in circles through the same set of states. To resolve a playthrough of this type, we encourage the creator to look at the states involved in the cyclic pattern.

If there is a case where we find that the same learner qualifies under multiple playthrough types, we will be following this priority order to record playthrough types: Cyclic > Multiple incorrect > Early quit.

Admin-defined constants

1. early_quit_threshold_in_secs: int. This could be 60 seconds after the exploration was actually started. Actually starting an exploration means a learner progressed past the first card.
2. number_incorrect_answers_threshold: int. This could be 5 wrong answers to the same state.
3. number_repeated_cycles_threshold: int. This could be 3 continuous cyclic set of states traversed.
4. max_playthroughs_for_issue: int. This could be 4-5 playthroughs.
5. record_playthrough_probability: float. This could be 0.2/0.25.
6. ALLOWED_ISSUE_IDS: [‘EarlyQuit’, ‘MultipleIncorrect’, ‘Cyclic’]

Playthrough representation

Domain object classes:

class ExplorationIssue {
    issue_id: str. ID of the Issue.
    schema_version: int. Schema version of the customization args dict.
    customization_args: dict. The customization dict. The keys are names of customization_args and the values are dicts with a single key, 'value', whose corresponding value is the value of the customization arg.
}

There will be an issue_registry.py file that would allow specific exploration issue classes to be retrieved from extensions.exploration_issues. These classes are defined below.

class BaseExplorationIssueSpec:
    _customization_arg_specs = [] // This will be overridden in the sub classes.

The BaseExplorationIssueSpec will also have classes inheriting from it. There will be one of these subclasses for each playthrough type. This way, adding a new playthrough type to be recorded will be as simple as adding a class that inherits from the BaseExplorationIssueSpec class.

```
class EarlyQuit(BaseExplorationIssueSpec):
        CURRENT_SCHEMA_VERSION = 1

        _customization_arg_specs = [{
                'name': 'state_name',
                'schema': {
                        'type': 'unicode',
                },
                'default_value': ''
        }, {
                'name': 'time_spent_in_exp_in_msecs',
                'schema': {
                        'type': 'int',
                },
                'default_value': 0
        }]

class MultipleIncorrectSubmissions(BaseExplorationIssueSpec):
        CURRENT_SCHEMA_VERSION = 1

        _customization_arg_specs = [{
                'name': 'state_name',
                'schema': {
                        'type': 'unicode',
                },
                'default_value': ''
        }, {
                'name': 'num_times_answered_incorrectly',
                'schema': {
                        'type': 'int',
                },
                'default_value': 0
        }]

class CyclicStateTransitions(BaseExplorationIssueSpec):
        CURRENT_SCHEMA_VERSION = 1
        _customization_arg_specs = [{
                'name': 'state_names',
```

```
        'schema': {
            'type': 'unicode',
        },
        'default_value': ''
    }]
```

```
class LearnerAction {
    action_id: str. ID of the learner action.
    schema_version: int. Schema version of the customization args dict.
    customization_args: dict. The customization dict. The keys are names of
customization_args and the values are dicts with a single key, 'value', whose
corresponding value is the value of the customization arg.
}
```

There will be an action_registry.py file that would allow specific learner action classes to be retrieved from extensions.actions.

The following class will be the super class for all learner actions. Each particular learner action will be a class that inherits from this super class.

```
class BaseLearnerActionSpec {
    _customization_arg_specs = [] // This will be overridden in the sub classes.
}

class ExplorationStart(BaseLearnerActionSpec):
    CURRENT_SCHEMA_VERSION = 1

    _customization_arg_specs = [{
        'name': 'state_name',
        'schema': {
            'type': 'unicode',
        },
        'default_value': ''
    }]


class AnswerSubmit(BaseLearnerActionSpec):
```

```
CURRENT_SCHEMA_VERSION = 1

_customization_arg_specs = [{
        ‘name’: ‘state_name’,
        ‘schema’: {
                ‘type’: ‘unicode’,
        },
        ‘default_value’: ‘’
}, {
        ‘name’: ‘dest_state_name’,
        ‘schema’: {
                ‘type’: ‘unicode’,
        },
        ‘default_value’: ‘’
}, {
        ‘name’: ‘interaction_id’,
        ‘schema’: {
                ‘type’: ‘unicode’,
        },
        ‘default_value’: ‘’
}, {
        ‘name’: ‘submitted_answer’,
        ‘schema’: {
                ‘type’: ‘unicode’,
        },
        ‘default_value’: ‘’
}, {
        ‘name’: ‘feedback’,
        ‘schema’: {
                ‘type’: ‘unicode’,
        },
        ‘default_value’: ‘’
}, {
        ‘name’: ‘time_spent_state_in_msecs’,
        ‘schema’: {
                ‘type’: ‘int’,
        },
        ‘default_value’: 0
}]
```

```
class ExplorationQuit(BaseLearnerActionSpec):
    CURRENT_SCHEMA_VERSION = 1

    _customization_arg_specs = [{
        'name': 'state_name',
        'schema': {
            'type': 'unicode',
        },
        'default_value': ''
    }, {
        'name': 'time_spent_in_state_in_msecs',
        'schema': {
            'type': 'int',
        },
        'default_value': 0
    }]
```

Datastore NDB classes:

To represent the playthroughs as displayed in the design mock above, the recorded playthrough data will have the following structure.

```
class PlaythroughModel {
    playthrough_id: str. exp_id.<random_hash_16_characters> - This is the model ID.
    exp_version: int. Version of the exploration.
    issue_id: str. ID of the issue.
    issue_customization_args: dict. The customization args dict for the given issue_id. The specs for this dict are as specified above in the different subclasses of BaseExplorationIssueSpec.
    playthrough_actions: list(dict). This will be a list of dicts. Each dict will be of the form:
        {
            action_type: str. Type of the learner action.
            action_customization_args: dict. The customization args dict for the given action_type. The specs for this dict are as specified above in the different subclasses of BaseLearnerActionSpec.
```

}

is_valid: bool. Whether a Playthrough is valid. A playthrough is invalidated if it's unresolved and its problematic state has been deleted by an exploration commit. For cyclic issues, a playthrough is invalidated if it's unresolved and the deleted state is any one of the states involved in the cycle.
}

We also need a model that maps an exploration_id to a list of all the playthrough IDs associated with it. This helps for efficient retrieval. An instance of this will be created when an exploration is created.

class ExplorationIssuesModel {
exp_id: str. ID of the exploration, also the ID of the model.
unresolved_issues: list(dict). Each dict would be of the form
{
issue_type: str. Type of the issue
issue_customization_args: dict. The customization args dict for the given issue_type. The specs for this dict are as specified above in the different subclasses of BaseExplorationIssueSpec.
playthrough_ids: list(str),
}
}

Job for creating mapping instances

There will be a one off MR job implemented: ExplorationIssuesModelCreatorOneOffJob. This job will iterate over all the ExplorationModel instances and create a ExplorationIssuesModel in the datastore with exp_id as the model ID and unresolved_issues list as empty. If there are exploration models with existing ExplorationIssuesModel, we skip the exploration.

Controllers for storing/fetching playthroughs

Controllers would be implemented to store playthrough data recorded in the front end, to retrieve the list of issues and to retrieve playthrough data to display to the creator. Any calls to these controllers will access/modify the datastore. There won't be any interleaving calls to the datastore. The datastore is not accessed/modified too frequently to cause race conditions.

StorePlaythrough:

This controller is used by the front end to store the playthrough identified to be useful.

POST method:

1. Extract exp_id from incoming request.
2. Extract playthrough_type, exploration_issue and playthrough_data from the incoming request.
3. Retrieve ExplorationIssuesModel instance for exp_id and make a domain class object from it.
4. Check whether the exploration_issue exists in unresolved_issues by checking if the two dicts are equal. If it does not exist, create a new element in the unresolved_issues list.
5. If it exists and the num_playthroughs >= max_playthroughs_for_issue, do not record the playthrough.
6. Else, create new PlaythroughModel datastore entry with id as 'exp_id.<16_character_hash>'.
7. Add the playthrough ID generated in step 6 to the corresponding element in unresolved_issues field of ExplorationIssuesModel instance.

FetchIssues:

This controller is used by the front end to retrieve the list of issues for an exploration.

GET method:

1. Extract exp_id from incoming request.
2. Retrieve ExplorationIssuesModel instance for exp_id and get the list of all unresolved issues.
3. Send the unresolved_issues list to the front-end.

FetchPlaythrough:

This controller is used by the front-end to retrieve a particular playthrough for an exploration.

GET Method:

1. Extract playthrough_id from incoming request.
2. Retrieve PlaythroughModel instance from datastore (using get_by_id() method) and send it to the frontend after converting to dictionary.

ResolveIssue:

This controller is used by the front end to mark an issue as resolved.

POST method:

1. Extract exploration_issue (dict containing issue type and list of playthrough ids) from incoming request.
2. Extract exp_id from the incoming request.
3. Delete all PlaythroughModels for the list(playthrough_ids).
4. Retrieve ExplorationIssuesModel instance for exp_id and remove the playthrough_issue from unresolved list.

Recording playthroughs in the Exploration Player UI

All playthrough data will be recorded in the exploration player UI. After recording the playthrough, if it is deemed to be useful, it is stored.

Front end domain classes:

First and foremost, there needs to be front end domain object classes defined for moving around and performing operations on the playthrough data. The following

classes will be implemented in the front end (the properties of these classes are the same as their properties in the datastore):

- PlaythroughObjectFactory
- LearnerActionObjectFactory

Scrutiny of learner actions:

The following actions in the front end are scrutinized:

- If the learner starts an exploration, ‘start_exploration’ type learner action instance is created and added to the playthrough data.
- If the learner submits an answer, ‘submit_answer’ type learner action instance is created and added to the playthrough data.
- If the learner quits the exploration, ‘quit_exploration’ type learner action instance is created and added to the playthrough data.

PlaythroughService:

In addition to this, a PlaythroughService file will be created and will have the following methods:

- initSession(expId, expVersion). This constructor method will initialize the playthrough data instance.
- recordLearnerAction(learnerActionObject). This method will add learner action instances to the playthrough data according to the section Scrutiny of learner actions.
- endSession(). This destructor method will analyze the playthrough according to the section Identifying useful playthroughs.

Analyzing playthroughs for usefulness can happen whenever the ‘beforeunload’ event gets fired. After analyzing a playthrough, if its deemed useful, we record the playthrough and set a flag in PlayerServices. Since the ‘beforeunload’ event is only a MaybeQuit event, using the flag mentioned above, we can update the playthrough recorded everytime the ‘beforeunload’ event gets fired. This way, we should have the latest playthrough at all times. The procedure for doing this is:

```
$window.addEventListener(‘beforeunload’, function(e) {
        analyzePlaythrough();
});
```

There's another alternative of analyzing playthroughs for usefulness in the backend if the analyzing might be a blocking operation to do in the front end.

Identifying useful playthroughs:

When the learner quits an exploration, the following steps will be executed.

1. Check the time of quitting the exploration (after exploration is actually started). If this time is less than early_quit_threshold_in_secs, set playthrough type to be 'early_quit' and record the metadata.
2. Check the playthrough data for the state the learner quit at. If the learner submitted more incorrect answers than number_incorrect_answers_threshold, set playthrough type to be 'multiple_incorrect' and record the metadata.
3. To identify playthroughs that have cyclic patterns, the following procedure is used.
    a. Create a tuple (cycle_identified) which will have two elements - a string and an integer.
    b. Create an array (visited_states_bitmap) of the same length as the number of states and set all values of the array to zero (this will function as a bitmap array).
    c. Create another empty array (visited_states) which will be holding the states visited by the learner.
    d. When the learner visits a state, check if the visited_states_bitmap value for that state is 0. If it is zero, change it to 1. If it is 1, make visited_states_bitmap list empty. Concatenate all the state names of the visited_states into a string and empty visited_states.
    e. If cycle_identified is empty, set cycle_identified[0] to be the string generated in step D, and cycle_identified[1] to be 1.
    f. If cycle_identified is non empty, check whether cycle_identified[0] is equal to the string generated in step D. If it is not equal, replace cycle_identified[0] with the new string and set cycle_identified[1] to be 1. If they are equal, increment cycle_identified[1].
    g. If cycle_identified[1] is equal to number_repeated_cycles, playthrough type 'cyclic' has been identified.

When a useful playthrough is identified, record it by sending a POST request to the StorePlaythrough handler. This step is performed only with a probability of record_playthrough_probability.

PlaythroughDirective:

The PlaythroughDirective.js file will have the following methods and scope variables:

- retrieveIssues(expId). This method will retrieve all the issues from the backend using a call to the controller.
- retrievePlaythrough(playthroughID). This method will retrieve the corresponding playthrough using a call to the controller.
- generateSuggestionForIssue(playthroughIssue). This method will generate the suggestions for any issue type at a particular state.
- ResolveIssue(explorationIssue). This method will post the explorationIssue dict to the backend using a controller.
- highlightLearnerActions(playthrough, explorationIssue). This method will add a boolean property to all learner actions to highlight them.

There will also be a playthrough_directive.html template which will contain all the template code for the visualization of playthroughs.

Ensuring learner anonymity:

All the data sent to the server through the handler has only exploration-related data and actions performed by a learner. There is no personal information being sent, nor can a learner be identified by the playthrough data being stored. Also, the playthroughs get displayed to the creator only if a minimum of two playthroughs get stored. In this case, even if the creator sent the exploration to 2 students and both playthroughs get stored, the creator would not be able to tell which is of which student.

Retrieving stored playthroughs

When a creator navigates to the statistics tab, a call to the FetchIssues controller is made. This returns all the unresolved issues for the exploration in question along with their supplementing playthrough IDs. When the creator navigates to a particular playthrough, a call to the FetchPlaythrough controller is made and the corresponding playthrough instance is returned.

If a creator marks an issue as resolved, a call to the ResolveIssue controller is sent and the issue is removed from the creator's view.

Handling versioning with playthroughs

There are cases where the exploration creator might not have checked the playthroughs recorded at some version of the exploration, but still made changes to the exploration and updated it. All the unresolved playthrough data for an exploration have to be modified when an exploration commit occurs to fit into this system.

When an exploration is updated, all of the commit changes are handled in exp_services._save_exploration() method. We create a method in stats_services (handle_playthrough_changes()) that would handle all the playthrough instance changes, and invoke this method from the _save_exploration() method.

State deletion:

Given the event of a state being deleted, all corresponding PlaythroughModel instances for that state are to be marked non valid. Also, any 'cyclic' type playthrough instances which has the deleted state in its set of states are marked non valid as well.
Also, if the playthrough consists of some other states, we do not take into account the deletion of any of those states since the learner didn't quit at any of those states. We only care about the state the learner quit at, which is our concerned state.

State rename:

In the event of a state rename, we generate the corresponding change logs' exp_domain.ExplorationVersionsDiff domain object and use the state rename mappings to rename the state name of the playthroughs that have the concerned state name as the renamed one. This is a workaround use case since the State ID Mapping job is not complete yet and has not been run in production.

Exploration revert:

In the event of an exploration revert, any unresolved playthrough instances marked non valid in the version that we are reverting to are marked not hidden. Also, we generate the exp_domain.ExplorationVersionsDiff object for the current version and the reverted version, and rename the states accordingly as well.

Steps involved:

1. Invalidate all PlaythroughModel instances with exp_version > revert_to_version.
2. Retrieve all valid playthroughs that are associated with the state that needs to be renamed, and rename them to its older name.
3. Retrieve all invalid playthroughs (with exp_version <= revert_to_version) associated with the deleted state (that needs to be 'not deleted') and mark them valid.

Marked non valid:

When a playthrough is marked non valid, we will be setting a field (is_valid) to True, while actually not removing the playthrough from the datastore. This makes sure that we can return invalidated playthroughs to the creator in case of an exploration revert.

Milestones

Milestone 1:

In this milestone, all the backend code functionality will be implemented. After milestone 1, the backend code will have the ability to fetch/store/resolve playthroughs.

Milestone 1.1 (1 week):

Code:

- Implement the PlaythroughModel and ExplorationIssuesModel classes in core.storage.statistics.gae_models.py

- Implement the Playthrough, Base ExplorationIssue, ExplorationIssuesMapping and LearnerAction domain classes in core.domain.stats_domain.py
- Implement all the Issue subclasses and Learner action subclasses in extensions.issues and extensions.actions respectively.

Tests:

- Add tests for PlaythroughModel and ExplorationIssuesModel in core.storage.statistics.gae_models_test.py
- Add tests for the Playthrough, Base ExplorationIssue, ExplorationIssuesMapping and LearnerAction domain classes in core.domain.stats_domain_test.py

Milestone 1.2 (2 weeks):

Code:

- Implement the ExplorationIssuesModelsForExistingExplorationsOneOffJob in core.domain.exp_jobs_one_off.py
- Implement the StorePlaythrough, FetchIssues, FetchPlaythrough and ResolveIssue handlers in core.controllers.reader.py

Test:

- Test that the one off job works as expected.
- Add integration tests for all the handlers in core.controllers.reader_test.py

Milestone 1.3 (1 week):

Code:

- Add functionality for handling versioning in a method implemented in core.domain.stats_services.py

Test:

- Test thoroughly that playthroughs get updated along with explorations during a commit
- Test that state renames modify all corresponding playthrough instances to have the new state name
- Test that state deletions marks all corresponding playthrough instances hidden

Milestone 2:

In this milestone, all infrastructure required to record useful playthroughs in the Exploration Player UI will be implemented. After this milestone, all useful playthroughs can start being recorded (though creator will be able to address them only after Milestone 3).

Milestone 2.1 (1 week):

Code:

- Implement PlaythroughObjectFactory.js and LearnerActionObjectFactory.js

Tests:

- Add Karma tests for the PlaythroughObjectFactory and LearnerActionObjectFactory classes

Milestone 2.2 (2 weeks):

Code:

- Implement all required helper methods in PlaythroughService.js
- Identify learner actions in the exploration player and call corresponding methods in PlaythroughService to record them
- Implement method in PlaythroughService that identifies and stores useful playthroughs

Test:

- Add Karma tests for PlaythroughService

Milestone 2.3 (1 week):

Code:

- Implement the retrieveIssues and retrievePlaythrough methods of PlaythroughDirective.js
- Implement a basic playthrough_directive.html template which displays raw playthrough data

Test:

- Add a Protractor e2e test to ensure that useful playthroughs get stored and is consequently displayed on the statistics tab

Milestone 3:

In this milestone, all infrastructure required to display the playthrough data to the creator will be implemented. Also, the proposed design for displaying the playthroughs will be coded. The functionality to resolve playthroughs will be implemented as well. After this milestone, creators will be able to visualize useful learner playthroughs!

Milestone 3.1 (1 weeks):

Code:

- Implement the remaining methods in Playthrough_directive.js

Tests:

- Add Karma tests for the above functionalities

Milestone 3.2 (2 weeks):

Code:

- Complete implementation of playthrough_directive.html, ensuring that the designs in the mock are replicated

Test:

- Modify the protractor test implemented in Milestone 2 to fit into the new design for representing playthrough data to the creator

Milestone 3.3 (1 week):

- Double check all documentation for all code written

- Write a Wiki page to explain the process of adding more criteria for identifying a playthrough as useful
- Wrap up any other work that might potentially be pending

Summer plans

Time zone

Throughout the summer, I'll work in the UTC+5:30 time zone.

Time to commit to the project

Throughout the duration of the GSOC term, I plan to commit 6-8 hours on all weekdays and around 2-3 hours on weekends.

Obligations/Commitments

Throughout the summer, I have no major obligations or commitments. The only commitment I have apart from GSOC is to prepare for GRE, which would be self study.

Communication

Contact information

My preferred method of communication would be through hangouts. Otherwise, I can be contacted through email.
Full Name: Pranav Siddharth Sreekanthan

Email address: pranavsid98@gmail.com
Contact number: +919865166625
Github profile: https://github.com/pranavsid98
LinkedIn profile: https://www.linkedin.com/in/pranavsiddharth98/

How often do I plan on communicating with my mentor?

I will provide a status update to my mentor every few days (can communicate as and when required as well).