SiteWide ACL ReFactor

(OPPIA)

Name : Yogesh Sharma
Email : yogesh.sharmaakaruler@gmail.com, yogesh.sharma@research.iiit.ac.in
Github handle : 1995YogeshSharma (link to profile)
University : International Institute of Information Technology, Hyderabad
Program : Dual degree (btech + MS) in Computer Science

I am currently pursuing my 3rd year of BTech in Computer Science. IIIT Hyderabad is among the *best institutions for computer science in India*. I am performing well in academics and plan to keep on improving. I have a cgpa of **9.0/10** (top 5% in the batch of 250 students). I am also doing research in Robotics Research Center at our college. My current research area involves around scene detection in self driving car.

Project Details

- Project name

  Sitewide ACL refactor

- Why are you interested in working with Oppia?

  I found Oppia while searching for the orgs to work on for GSoC17 based on the list of orgs selected last year for GSoC16. I have following main reasons to work for Oppia:-

  - I find their area of work to be very *interesting*. I am a fan of online learning concept. Improvements in this area reach to millions of people across the globe.
  - When I saw oppia.org and a few explorations, I saw a huge potential for it to grow into an awesome learning platform which may really help people to learn *without losing interactivity*. Humans learn better when they are allowed to make mistakes and correct them which is exactly what explorations help them with.

- And I see, with the involvement of machine learning and deep learning, we may be able to provide better and better experience to the users.
- I find Oppia team members to be very *friendly*, *welcoming and helpful*. The org is also very *active*. Getting a personal mail from Sean himself after registering on the site is also energy booster for new contributors.
- Being a part of such a dynamic team which is trying to make a great impact in the field of online learning is also very satisfying.

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

  Why is the Project worth doing ?

  - Access Control is a two step process - **authentication**(checking *who* the user is) and **authorization**(checking *what* the user can do). We are handling how the authorization works in Oppia in this project.
  - Authorization refers to '*what a user can do*' in the website.
  - In Oppia, we currently have a lot of different positions like *collection_creator*, *viewer*, *moderator*, *admin*, *super_admin*.
  - As the site has grown and will grow, it is necessary to have a ***flexible* authorization system** which handles the permission management in a way that it can easily be modified and extended. So, this project is very essential to Oppia.

  What interests me about doing this project ?

  - The project is important for the Oppia organization. The current implementation is *pseudo role based* and will improve greatly if we are able to shift it to *action based* access control (some reasons mentioned for preferring action based approach in other section).
  - I hadn't studied much about ACL (only studied a bit in my Operating System classes) and never implemented one. But I asked Sean and he told that it's not a prerequisite. So, I began reading about access controls and got interested in the topic. I find the project interesting and a great learning opportunity of an important concept.
  - The process requires a high amount of logical and analytical skills which got me all the more interested to do it.

- This project is mentored by Sean himself. I like the way he deals with people. He is always calm and encouraging in his replies and suggestions and has a great attention to detail while reviewing code. I feel that working under him over the summer will help me in a lot of ways.

- `Prior experience (especially with regards to technical skills that are needed for the project)`

  I ->
  - have been working with python in some way or the other for the past 2 years.
  - have worked with some frameworks in python (django and web2py)
    - Made a blog app in which bloggers can publish a blog and reviewers can review it to improve their writing skills (as a personal project) - django
    - Made a hotel locating app (as a part of course project) - web2py
  - Worked as an intern for EnhanceEdu (they are making an educational platform for college students)
    - Met them through my course project and they offered me the internship at the end of project.
    - The website is built on *moodle* platform. I solved some of the existing issues with the site, added plugins and implemented some features.
    - They are working on learning by doing methodology and use butterfly model (they had published a research paper on this).
  - worked as an intern for zetagile company (they are working on making a *health care and health monitoring* website)
    - Site is based on AngularJS with java as backend.
    - I worked on the frontend.
    - Implemented features auto-complete search bar, file upload handling and worked with some of the APIs they made.
  - Made a *bot* for playing ultimate tic-tac-toe in python
    - Implemented *minimax* algorithm with *alpha beta* pruning and an *evaluator* function adjusted to the game rules.
  - Made some games (*basic version of donkey kong and fruit ninja*) using *pygame* module in python.
  - Have studied courses in Algorithms, Data Structures, Operating Systems etc which will help with the implementation of the project.

Source code for some of my work can be found at my github profile.

- ## PRs made
    - I have got following PRs merged to the Oppia codebase.
        - https://github.com/oppia/oppia/pull/3055
        - https://github.com/oppia/oppia/pull/3068
        - https://github.com/oppia/oppia/pull/3077
        - https://github.com/oppia/oppia/pull/3216
        - https://github.com/oppia/oppia/pull/3211
        - https://github.com/oppia/oppia/pull/3192

- ## Project plan and implementation strategy

This project will implement ***Action based access control***, removing the current pseudo role based access control.

Here are some of the *reasons for choosing Action based implementation* instead of role based implementation :-

- ***Requirements Growth and Change*** **:** With time requirements change. With role based approach, there is high coupling between authorization and roles. So, making changes is very difficult. Eg.

```
if not rights_manager.Actor(self.user_id).is_moderator():
    raise self.UnauthorizedUserException(
        'You do not have the credentials to access this page.')
..
user_services.record_user_logged_in(self.user_id)
```

--This is a snippet from base.py.

  Suppose, we want to add some other role in such cases, we have to look through all similar occurrences in codebase and make appropriate changes. On the other hand, if we want to add new role in the action based setup, we can add the role in group hierarchy and attach its unique actions to it. There is no need to modify other code.
- Actions give us a more ***intuitive abstraction*** and make the authorization and role management ***decoupled***.
- The code can speak for itself representing itself as an activity that can be performed. Relationships between the roles and permissions can be ***modelled separately***.

- Relevant information is in one place, so the code is ***self documented***. Instead of looking throughout the code to see if a role is checked, we provide a single, consolidated location for the role's permission.

With the new implementation, adding a group will have nothing to do with the existing actions, we would ***append the group*** in group hierarchy to the parent whose actions it inherits and then ***attach to it unique actions*** that belong to that group.

Similarly, adding an action requires designing a ***decorator*** for it and ***attaching the action to the group*** that will be the least required position to perform that action.

I have read a multitude of sources to come up with the idea for new system. I have studied various *sources for access controls*, including a ***research paper*** (link to the research paper) on action based access control. Although they are implementing it in network layer but the idea still helpful.
The ***meteor-roles package*** (https://github.com/alanning/meteor-roles/) and the ***blog*** (link to blog) are very helpful as they support a very similar approach used here (using strings of actions and attaching them to roles).

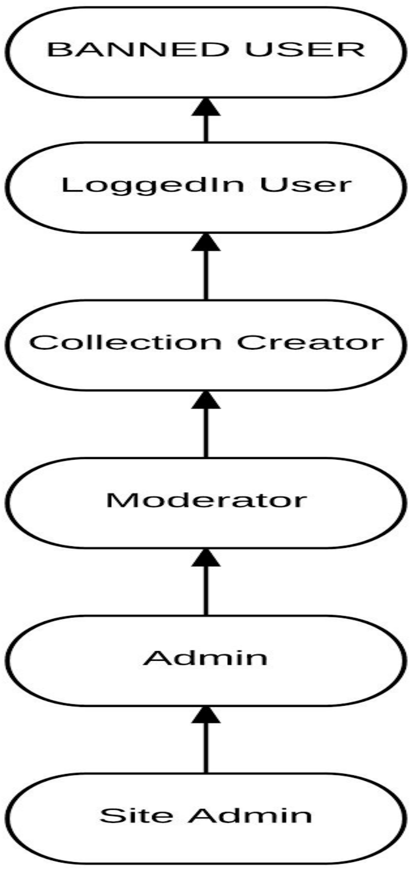
Overview of idea of implementing Action based Access Control

- **Actions** are *strings* which define the action to be performed eg. create_a_collection and so on.
- **Group** is a *set of actions* with its name representing the *role*. Actions here pertain to this role which are not inherited from parent.
- Groups will exist in a **hierarchical order** in a ***Directed Acyclic Graph*** (DAG) where an edge from A to B implies that A *inherits* all permissions from B. Edge points towards the parent.
- **Decorators** will be hooked on the controllers to do authorization before performing the action.
  A separate module ***action_decorators.py*** will be made to store all the new decorators. When a decorator is applied to a function it will act as *wrapper* and *extend the functionality* to include checking whether the user belongs to the group having permission to perform this action. This will be done from services made for the group object.
  All the ***additional checks***, like whether user is owner of collection or not, will also be done here.
- A new field will be introduced to the ***UserSettingsModel*** model representing the group/role of user. When a user is registered, he will be assigned with a

role. By default the role is logged in user. Admins can change the role of the user.

- The **RoleHierarchy DAG** needs to be loaded ***only once*** when the server *starts* and will suffice for the queries. So, database needs to be queried only when ***server starts*** and ***when a change is made to actions/groups*** . Thus ***saving*** a lot of requests to the database. When a change is made to group structure or actions, changes will be written to database and DAG will be reconstructed from database to replace the current DAG.
- When a user logins, a single call to the service *get_actions_for_user()* will be made which returns a list of all the actions the user can perform. We'll store this list in the user session. Also, we will store the role of user in the session. This way, we have to traverse the graph and find out actions only once and can use the ***actions and role*** in the ***session*** for all further queries of the user.

Initial Structure of Role Hierarchy DAG :-



Here, arrow points to parent/parents. The inheritance of actions take place in opposite direction of arrow i.e - *perm(collection_creator) = perm(logged in user) U perm(Banned User) U perm(collection_creator).*

The list of actions which will be finally used and will be kept updated can be found at below link :-

➔ [list for actions]

Diagrams representing the User and related unique actions :

Blue - no need to check

Green - currently checked

Red - Not checked

BANNED USER

Play_an_exploration
Play_a_collection
Give_feedback_on_an_exploration
Playtest_an_exploration
Embed_an_exploration
View_exploration_stats

LOGGED IN USER

create_an_exploration
Publish_owned_exploration
Delete_unpublished_exploration
Add_collaborators_to_exploration
Auto_accept_incoming_suggestion_without_review
Edit_exploration
Accept_incoming_suggestions
Suggest_edit_to_exploration
Track_personal_learning_progress
Track_exploration_with_edit_access
Track_collection_with_edit_access
Rate_an_exploration
Propose_for_slot_in_collection
Propose_for_new_question_in_practice_session_in_collection
Adjust_email_preferences

COLLECTION CREATOR

Create_a_collection
Edit_a_collection
Propose_collection_for_slot_in_course
Accept_exploration_to_collection
Reject_exploration_from_collection
Define_skills_for_exploration_in_owned_collection
Accept_questions_to_collection
Reject_questions_from_collection
Publish_owned_collection

MODERATOR
Ban_user
Unpublish_exploration
Unpublish_collection
Delete_private_collection
Delete_private_exploration

ADMIN
Create_topic_outline_for_courses
Accept_collection_to_courses
Accept_exploration_to_courses
Remove_exploration_from_courses
Remove_collection_from_courses
Define_skills_for_collection_in_courses
Assign_role_to_user

SUPER ADMIN
Add_action_to_group
Delete_action_from_group
Remove_action
Add_group
Remove_group
Assign_user_as_admin

Sidenotes ->
Actions for release_ownership, ResolvedAnswersHandler, ExplorationResourcesHandler, ImageUploadHandler, EditorAutosaveHandler are not present.

For a ***non logged-in*** user, permissions are similar to that of banned user. We don't need to put checks on their actions as those can be performed by any user. So, they don't need authorization.

Timeline for implementation :-

April 10 to May 4 :- I'll be free starting April 10. I'll continue my work with pending issues. Update the list for actions and technical design doc for groups so that they are complete before the work starts.

May 5 to May 30 :- Although this is mentioned as community bonding period but I'll start working in this period so that we may easily cover everything in time.

MILESTONE 1 :

Complete technical design doc.

Audit the existing rights_manager.py.

Complete the list of actions.

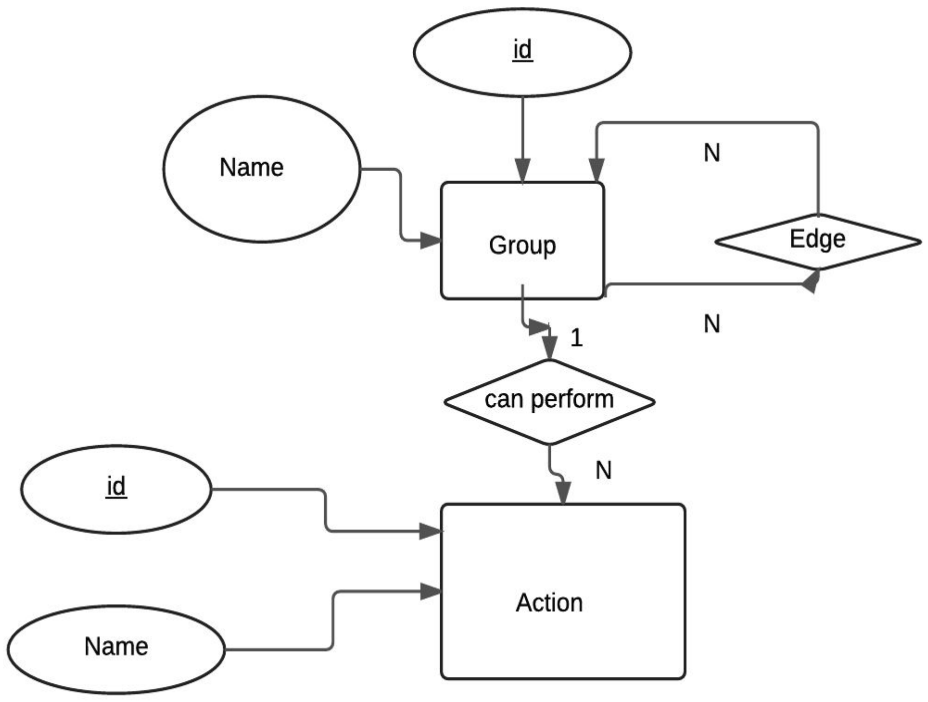
Implement new storage layer, domain objects and services. Implement corresponding tests.

Implement one-off job to fill the role field to the UserSettingsModel. Give user the default role when user registers.

Implement one-off job to fill the new tables created (Groups, Actions etc) on the basis of current roles hierarchy.

The technical design that'll be used and kept updated can be found on below link

➔ link to the tech design for groups and actions



Models Required :-

Groups - to store the name of the role

GroupId	GroupName

Edges - to store the relationship order between connected groups (A -> B)

EdgeId	GroupIdTo (destination)	GroupIdFrom (source)

Actions - to store name of actions

ActionId	ActionName

GroupToActionsMap - to map group to the actions it can uniquely perform

GroupId	ActionId

A field Role representing the name of the user's group will be added in UserSettingsModel

Domain objects to be added :-

Group - object for each node in the DAG

RoleHierarchy - object for the DAG

Instance of RoleHierarchy class will be a dictionary of Group objects. This will represent DAG.

```
Eg. RoleHierarchyInstance = {
        'Group_name1' : GroupObjectInstance1,
        'Group_name2' : GroupObjectInstance2,
        .
        .
        }
```

Services To Be Added :-

Add_group, Remove_group, Attach_user_to_group, Get_user_group, Add_action_to_group, Delete_action_from_group, Remove_action, Get_actions_for_user

```
Add_group :-
    Adds a new group to role hierarchy
    Input : group_name, parent_name, permissions

    Create new Group object with permissions, parent_names
    Append it to role hierarchy DAG
    Write to database

Get_actions_for_user :-
    If present in session:
        Return user_actions
    Else :
        find all paths from source to the target node in the DAG
        take union of all the permissions along the path.
        Store value in session and return

Remove_group :-
    A = [ all incoming edges to the group node to be removed ]
    B = [ all outgoing edges from the group to be removed ]
    Now, remove entries of A and B from the database
    remove the group and related permissions from database
    insert new group and its permissions and insert edges for all sources in A to
    all destinations in B.
    Construct the DAG again and replace existing structure with this.
    Remove user_actions from session

Attach_user_to_group :
    Update the role field of user
    Get new permissions for the user from DAG
    Update the variables in session

Get_user_group :
    If present in session :-
        Return user_group
    Else
        Query database and find user_role
        Update session variable and return

Remove_action_from_group:
    Traverse DAG to find group
    Remove action from the set
    Remove groupId - actionId relationship in table GroupToActionsMap
    Invalidate session
```

Audit Strategy :

- Go through the existing code file by file checking the use of rights_manager.
- Make audit doc mentioning every use of rights_manager and the way in which it will be replaced by the new system.
- Check if the use case is covered with the existing test cases. If not, design and implement tests for this.

One Off Jobs to be made :

Two one-off jobs need to be made for migrating the existing role related data to new system and making the storage for groups and actions.

First, ***roles_job_one_off.py***. This will involve following steps :

Get list ADMIN_IDS, MODERATOR_IDS, BANNED_USERNAMES, WHITELISTED_COLLECTION_EDITOR_USERNAMES from config_domain -> Get all the users from database -> update user entries to fill their role field accordingly.

For testing, make a random entries in the lists -> run the job -> check whether data stored correctly.

How to run the job -> Stop all the jobs using the userSettingsModel and then run this job.

second, ***groups_job_one_off.py***. This will involve the following steps :

Have list of actions, list of groups, relationships between actions and groups and hierarchy between groups in different files.

Load these files -> fill corresponding tables

For testing, make above files with sample data and check for correctness of insertion.

How to run the job -> This doesn't have any dependency with any existing models. So, can be run from admin panel without stopping other jobs.

Coding Part :

- Backend :
  - In oppia/core/storage :
    - Implement group/gae_models.py for the storage of Group, Action, Edges and GroupToActionsMap models explained above.
    - Add the role field in UserSettingsModel in user/gae_models.py

    - In oppia/core/domain :
        - Implement group_domain.py for the domain objects Group and RoleHeirarchy.
        - Implement group_services.py for the services mentioned above.
        - Implement roles_job_one_off.py to fill the entries in the new models created. I.e : fill the roles of user based on the current lists like admin_usernames etc.
        - Implement groups_job_one_off.py to fill the Groups, Actions, Edges and GroupToActionsMap based on the current hierarchy in roles and list of actions prepared.
        - Edit jobs_registry.py to add the jobs.
    - Edit the _create_user method in oppia/core/domain/user_services.py to give role to user at time of register and add the method get_user_role in this file.
- Frontend :
    - Edit oppia/core/templates/dev/head/pages/admin/jobs_tab to add the one off jobs in the list.

Testing part :

- Backend :
    - /oppia/core/storage/groups/gae_models_test.py -> Make sure the data gets correctly stored in database.
    - /oppia/core/domain/group_domain_test.py -> Make sure empty DAG doesn't cause error. Test by creating a random DAG and using member functions.
    - /oppia/core/domain/group_services_test.py -> Make a random DAG and check each service. Cover cases like removal from empty DAG. (Attaching actions to multiple group is not a problem here as union is taken and when the DAG reloads, it is resolved)
    - /oppia/core/domain/roles_job_one_off_test.py -> populate models with random users and roles and run the job to verify whether the field gets populated properly.
    - /oppia/core/domain/groups_job_one_off_test.py -> run the job and check if database is correctly populated.
    - Add test in user_services_test.py for the added service get_user_role.

Breakdown

Date	Work Done
8 May - 11 May	Get Technical doc reviewed and make changes if required.
12 May - 18 May	Make the audit doc. Get list of actions and audit doc reviewed and make changes if required.
16 May - 21 May	Implement the storage layer. Implement tests for storage layer. Add role field in UserSettingsModel
22 May - 28 May	I'll not be available for one week around this time (maybe still be able to work for sometime, but not sure)*
29 May - 4 June	Implement domain objects and services. Make tests for domain objects and services.
5 June - 11 June	Edit _create_user method to give default role. Add get_user_role service to user_services.py and corresponding test in user_services_test.py Implement roles_job_one_off.py Implement roles_job_one_off_test.py
12 June - 18 June	Implement groups_job_one_off.py Implement groups_job_one_off_test.py Make required edits to add these jobs to job list in admin panel. Manually create a sample user set (with all types of users) and test the one-off jobs.
19 June - 21 June	Getting everything reviewed and merged.
22 June - 25 June	Buffer time for milestone 1

MILESTONE 2 :

Make a simple ui to access and modify group/action system.

Implement decorators for actions not currently checked (new actions in list of actions).

Implement tests for above decorators.

Implement any uncovered tests based on audit doc.

Simple UI for making Changes to groups/actions :-

Right now changes can be made to admin or moderator list through ***/admin#config***. A simple UI *similar* to that will be made to make changes (adding/removing actions/groups, changing hierarchy) in the new system.

We'll add tabs for update_groups and update_actions in admin_navbar.

/admin#update_groups -> url that will redirect to page for making changes in groups or user roles.
/admin#update_actions -> url that will redirect to page for making changes in actions.

Coding Part :

- Backend :
  - Make /oppia/core/domain/action_decorators.py - decorators for each action.
  - In oppia/core/controllers
    - Edit admin.py to handle new requests using group_change.py and action_change.py.
    - Implement group_change.py for handling the changes made to the group structure and role change.
    - Implement action_change.py for handling the changes in actions to be made.
    - Along with this edit main.py for redirections for /admin#update_groups and /admin#update_actions
- Frontend :
  - /oppia/core/templates/dev/head/pages/admin :
    - Add update_groups/update_groups_directive.html and update_groups/UpdateGroupsDirective.js for update_groups tab

        - Add update_actions/update_actions_directive.html and update_actions/UpdateActionsDirective.js for update_actions tab
        - Update AdminRouterService.js to add new routes to ADMIN_TAB_URLS
        - Edit other files to add links to these newly created tabs.

Testing Part :

- Backend :
    - Make /oppia/core/domain/action_decorators_test.py - test each action for cases of authorized and unauthorized access.
    - /oppia/core/controllers/group_change_test.py and /oppia/core/controllers/action_change_test.py -> Create random DAG and apply multiple changes to it using functions in controller and verify that output is as expected.

Breakdown

Date	Work Done
26 June - 2 July	Write backend code for the simple UI. Write tests for this code.
3 June - 9 July	Write frontend code for the simple UI. Manually test the working of frontend.
10 July - 16 July	Implement the decorators related to each action in the list. Implement test corresponding to each decorator. Implement uncovered tests (if any) based on the audit doc.
17 July - 21 July	Get everything reviewed and merged
22 July - 24 July	Buffer time for milestone 2

MILESTONE 3 :

Migrate all permissions and functionalities related to rights_manager to new system.

Do refactoring for the code where admin, moderator etc are checked from the lists in config domain. Replace them by checking from session variable.

Create audit doc for remaining code refactoring that can be done.

- This phase will handle migration of code for the rights_manager to new system.
- The audit document created in milestone 2 will have details of each occurrence of rights_manager
- We have the list of actions that will be the replacement of the rights_manager by now.
- So, decorator for each action will be implemented and the occurrence of rights_manager removed.
- After doing this for one file, the tests will be run to check if something went wrong.
- Shift the test to new test suite containing tests for all actions.
- Replace the old functionality of adding users to admin, moderators and others.
- Now, the new system is in place. Roles are assigned by the new system. So, replace all occurrences where role was checked with the help of lists (from config_domain) with the session variable role.
- Also, there are cases where there is redundancy in code that can be solved eg ExplorationRightsHandler and ExplorationModeratorRightsHandler can be merged into one.

  A doc of all changes that can be performed to refactor the code base will be made, where in occurrence and possible way of removal has to be mentioned.

Coding Part :

- ❖ Backend :
  - ➢ Add actions one bye one to action_decorators.py made in previous milestone.
  - ➢ Remove the use of rights_manager from corresponding places.
  - ➢ Replace the code for changing roles that uses config_domain as of now to using role variable. Eg - update_admins in config_domain or set_admins in test_utils
  - ➢ Replace code that uses these lists to know user's role to checking the session variable role.

- ❖ Frontend :
    - ➢ Remove the config functionalities to add users to admin_usernames etc. by editing code in templates/dev/head/pages/admin/

Testing Part :

- ❖ Backend :
    - ➢ Add test one by one to action_decorators_test.py.

Breakdown

Date	Work Done
25 July - 3 Aug	Replacing the current functionalities performed by rights_manager. Pick case one by one from doc. Implement the action for the change. Remove the existing code. Run tests to test functionality. Implement test for new action made.
4 Aug - 10 Aug	Replace/remove code mentioned in backend part and frontend part above (regarding the config_domain lists).
11 Aug - 14 Aug	Getting everything merged
15 Aug - 20 Aug	Make doc for further refactorings that can be performed.

* Apart from this I'll not be available for 3-4 days in June or July but the date is not fixed yet.

FUTURE WORK :

Do the refactoring that is proposed in the last doc in milestone 3.

Summer Plans

- Time Zone

IST (India Standard Time)

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

  During May to July, I will be able to spend 6-8 hours a day 6 days a week i.e 36-48 hours a week. I'll make sure to put at least 40 hours a week during this time.
  After that (i.e during august) I will be able to spend 4-5 hours a day 7 days a week i.e 28-35 hours a week. I'll make sure to put at least 30 hours a week during this time.

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

  Our college has vacations from May to July so I have no commitments during that time and classes start from Aug 10.

Communication

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

  E-mail : yogesh.sharmaakaruler@gmail.com, yogesh.sharma@research.iiit.ac.in
  Mobile no. : +91-9951617335
  Github handle (gitter) : 1995YogeshSharma

  Oppia is very active on **gitter**, so preferred method for most of the communication and meetings will be gitter.
  I'll keep daily devlogs as recommended by mentors to document my daily work.

- How often do you plan on communicating with your mentor?

  We'll remain in continuous touch with gitter whenever I need advice.
  There will be **biweekly** (or weekly based on mentor's consent) meetings to discuss the workflow to be followed. Meetings can be on gitter.