Oppia

GSOC 2021 Proposal

Name: Nikhil Agarawal

Project: Implement Schema Validation for handler param.

About You

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

Working with oppia is one of the most exciting tasks because its mission to provide and create lessons in a fun and exciting way fascinates me very much.

After working with various projects regarding Apis, Json handling and request responses, this project seems interesting and committable.

Prior experience

Considering the scope of this project, I have worked with similar ones which requires json handling, web requests and responses, etc.

Previous experience

Link to some of my previous pull requests are as follows:

No.	Title	Link
1	Allow exploration owners to remove users from their exploration.	Link: here
2	Adds async keyword in read-only-backend-api service	Link: here
3	Convert object factory to model class	Link; here
4	Adds decorator in run_in_transaction	Link: here
5	Migrate admin_prod_mode_activities_tab directive to angular components	Link: here

Contact info and timezone(s)

Contact Gmail: nikhil.agarwal.2019@gmail.com
Hangouts: nikhil.agarwal.2019@gmail.com
Optional gmail: nikagarwal093@gmail.com

Timezone: Indian Standard Time(IST) or GMT + 5:30

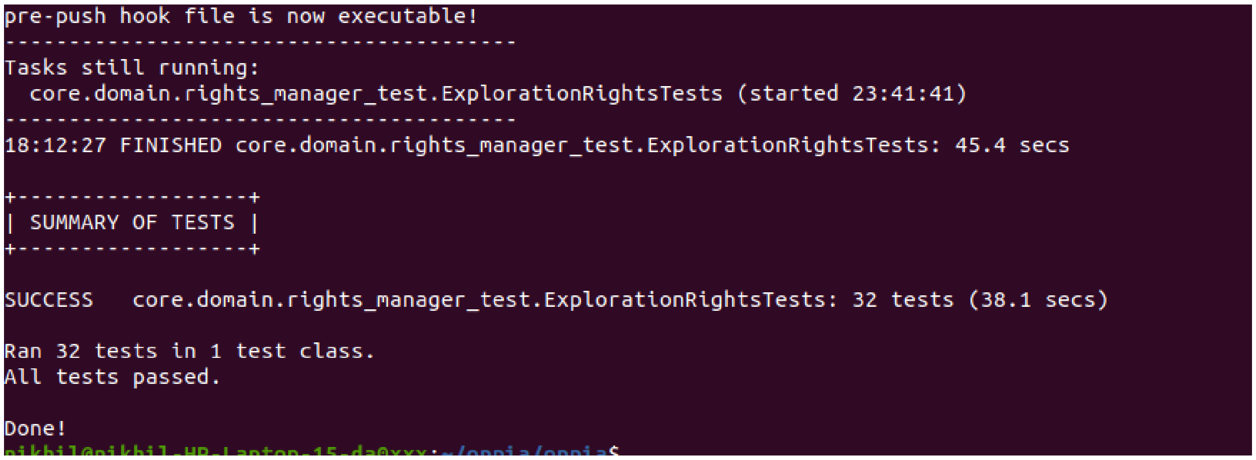
Time commitment

I plan to work approx 25 hours a week. The time commitment is planned with a view to cover GSoC criteria as well as considering uncertainties that may occur during the coding phase.

Essential Prerequisites

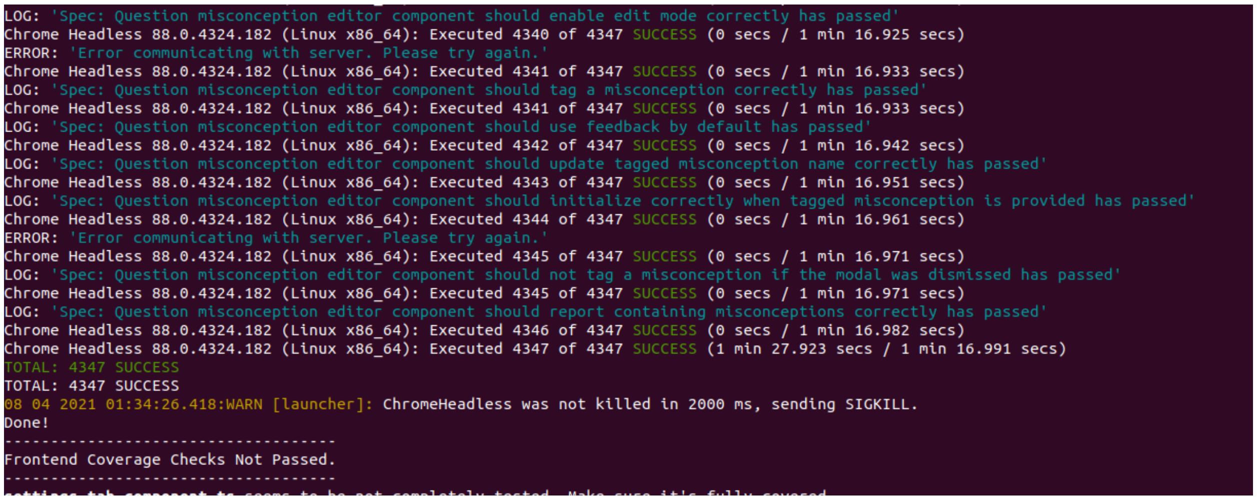
Answer the following questions (for Oppia web GSoC students):

- I am able to run a single backend test target on my machine.



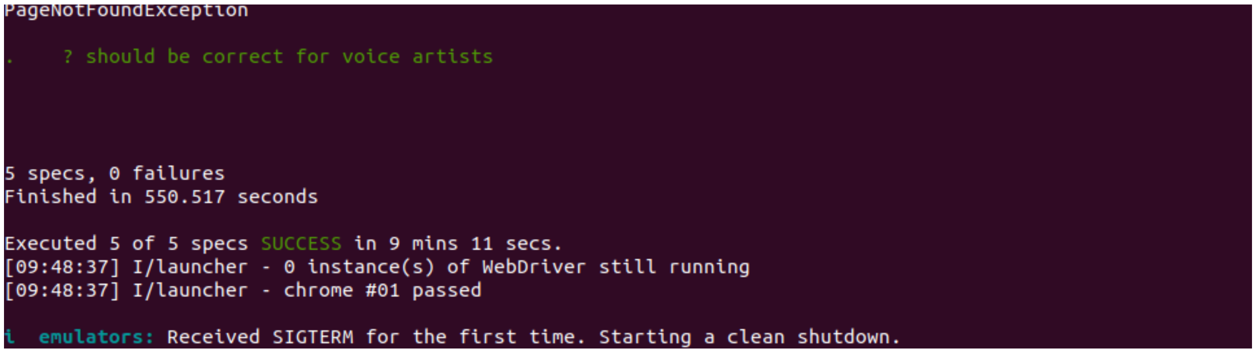
```
pre-push hook file is now executable!
----------------------------------------
Tasks still running:
  core.domain.rights_manager_test.ExplorationRightsTests (started 23:41:41)
----------------------------------------
18:12:27 FINISHED core.domain.rights_manager_test.ExplorationRightsTests: 45.4 secs

+------------------+
| SUMMARY OF TESTS |
+------------------+

SUCCESS   core.domain.rights_manager_test.ExplorationRightsTests: 32 tests (38.1 secs)

Ran 32 tests in 1 test class.
All tests passed.

Done!
nikhil@nikhil-HP-Laptop-15-da0xxx:~/oppia/oppia$
```

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



```
LOG: 'Spec: Question misconception editor component should enable edit mode correctly has passed'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4340 of 4347 SUCCESS (0 secs / 1 min 16.925 secs)
ERROR: 'Error communicating with server. Please try again.'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4341 of 4347 SUCCESS (0 secs / 1 min 16.933 secs)
LOG: 'Spec: Question misconception editor component should tag a misconception correctly has passed'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4341 of 4347 SUCCESS (0 secs / 1 min 16.933 secs)
LOG: 'Spec: Question misconception editor component should use feedback by default has passed'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4342 of 4347 SUCCESS (0 secs / 1 min 16.942 secs)
LOG: 'Spec: Question misconception editor component should update tagged misconception name correctly has passed'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4343 of 4347 SUCCESS (0 secs / 1 min 16.951 secs)
LOG: 'Spec: Question misconception editor component should initialize correctly when tagged misconception is provided has passed'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4344 of 4347 SUCCESS (0 secs / 1 min 16.961 secs)
ERROR: 'Error communicating with server. Please try again.'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4345 of 4347 SUCCESS (0 secs / 1 min 16.971 secs)
LOG: 'Spec: Question misconception editor component should not tag a misconception if the modal was dismissed has passed'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4345 of 4347 SUCCESS (0 secs / 1 min 16.971 secs)
LOG: 'Spec: Question misconception editor component should report containing misconceptions correctly has passed'
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4346 of 4347 SUCCESS (0 secs / 1 min 16.982 secs)
Chrome Headless 88.0.4324.182 (Linux x86_64): Executed 4347 of 4347 SUCCESS (1 min 27.923 secs / 1 min 16.991 secs)
TOTAL: 4347 SUCCESS
TOTAL: 4347 SUCCESS
08 04 2021 01:34:26.418:WARN [launcher]: ChromeHeadless was not killed in 2000 ms, sending SIGKILL.
Done!
------------------------------------
Frontend Coverage Checks Not Passed.
------------------------------------
settings-tab.component.ts seems to be not completely tested. Make sure it's fully covered.
```

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



```
PageNotFoundException

.    ? should be correct for voice artists



5 specs, 0 failures
Finished in 550.517 seconds

Executed 5 of 5 specs SUCCESS in 9 mins 11 secs.
[09:48:37] I/launcher - 0 instance(s) of WebDriver still running
[09:48:37] I/launcher - chrome #01 passed

i  emulators: Received SIGTERM for the first time. Starting a clean shutdown.
```

Project Details

Oppia's users currently interact with backend servers or datastore by providing data via API. Currently, the handler params are not verified correctly in all the places. This introduces a leakage in backed API structure which leads to data corruption and unexpected server issue(s).

This doc introduces the architecture for validating the params received by the handlers in an unified way and immediately raises an Invalid Input Exception (400) error, if the schema for expected params does not match with the received params.

Product Design

The project adds a backend structure for validating handler's params. It doesn't add/remove/update any user-facing UI features.

Technical Design

This docs is presenting ideas for architecture required for Implementing schema validation for handler params.

Technical requirements

The project requirements can be summarised as follows:

- Unified way to parse and validate params of different types int, string, unicode, list, dict etc.
- Provides a way to add custom validations for common data/dict schema in the codebase.
- Provides a way to handle and raise correct exceptions for invalid params.
- Provides options to enable/disable param validations for a given handler.
- Param validation enabled on at least a part of the current handlers.
- [GTH] Should normalize the given param if needed.

- [GTH] Avoid duplicate code.

*GTH: “Good to have”, these are the requirements which are optional and it would be good to have those in the validation architecture.

The new system to validate params and provide all the requirements stated above can be implemented using a schema validator. There are multiple options available for using schema validators:

1. Oppia’s existing schema validation system (defined in schema_utils.py)
    a. **“*Oppia’s SVS*”** term will be used in the doc to refer the schema validation system defined in schema_utils.py
2. Third party libraries:
    a. Cerberus
    b. Marshmallow

Oppia’s SVS schema API:

A data can be validated using Oppia’s SVS by providing a schema for the data. A schema for a data will be a dictionary with the following fields:

- **type**: The type of the data.
    - examples: bool, int, float, unicode, list, dict, html. The list and dict types have additional fields in their schemas (see below). The type for the data which has defined object class in objects.py will be ‘custom’ and the object class need to be mentioned in the obj_type field of the schema (for more details check other field of the API)
- **choices**: A list of values of the given type. The value entered must be equal to one of elements in the list.
- **validators**: A list of validators to apply to the return value, in order.
- [for type=list] **items**: The schema for an item in the list. A polymorphic list should not use this field.
- [for type=list] **len:** If present, the length of the list; must be an integer greater than 0. No elements can be added or deleted.
- [for type=dict] **properties:** This is a list whose elements are dicts. The item in the dict represents a schema for each key value pair in the dict. Each dict in the list should have two mandatory keys:
    - **name:** The name of the field
    - **schema:** The schema for the value corresponding to this field.
- [for type=dict] **description**: *optional*. If present, this gives a human-readable description of the field.
- [for type=custom] **obj_type**: Contains the type of the class of object, defined in objects.py.

Features provided by Oppia’s SVS

1. Provides a function normalize_against_schema to parse and validate params of different types int, string, unicode, list, dict etc.
2. Provides a system to validate complex structures either by defining a simple validator function or defining a separate object class.
3. Provides a way to combine custom validators with type checks for a given data.
    a. E.g 1: To validate a user id the schema can be:

```
{
    ‘type’: ‘unicode’,
    ‘validators’: [{
    ‘id’: ‘is_valid_user_id’
}]
}
```

E.g 2: To validate a new role for a user the schema can be:

```
{
    ‘type’: ‘unicode’,
    ‘choices: [feconf.ROLE_EDITOR, feconf.ROLE_OWNER ...]
}
```

4. Normalizes the data against the given raw data.
5. Provides a way to define default value for the custom type.

Features provided by third party libraries

- Cerberus
    1. Provides a function to parse and validate params of different types int, string, bool, etc.
       Example to validate user role:

```
schema = {
   'role': {
       'type': ‘string’,
       'allowed': [
           feconf.ROLE_EDITOR, feconf.ROLE_OWNER…] }}
document = { 'role': 'editor' }
v = Validator(schema)
v.validate(document)
```

```
>True
```

2. Provides a feature to validate complex structures by defining custom rules. Example to validate category string.

```
class MyValidator(Validator):
                def _should_start_with(self, field, value):
               if feild[0] != '(' :
               self._error(field, "Must start with parenthesis.")

schema = {'category': {'type': 'string' }}
document = { 'category': '(xyz)' }
v = MyValidator(schema)
v.validate(document)

>True
```

3. Normalizes the data against the given raw data.
4. Provides a way to define default value for the custom type.

- Marshmallow
    1. Provide a function to parse and validate params of different types like int, string, bool, etc.
        - Example for validating user roles.

```
class UserSchema(Schema):
   role = fields.Str(validate = OneOf(
               [ feconf.ROLE_EDITOR, feconf.ROLE_OWNER, .. ]
           ))

document = { 'role': 'editor' }
try:
   result = UserSchema().load(document)
except ValidationError as err:
   print(err.messages)
   print(err.valid_data)
```

    2. Provide a feature to write custom classes for validating params.
        - Example to validate category string.

```
class BandSchema(Schema):
    category = fields.Str()

    @pre_load
    def unwrap_envelope(self, data, **kwargs):
        if data[0] != '(':
            raise ValidationError('Must start with parenthesis.')
        return data


sch = BandSchema()
try:
   sch.load({"category": "(xyz)"})
except ValidationError as err:
   err.messages
```

3. Normalizes the data against the given raw data.
4. Provides a way to define default value for the custom type.

Similarities between Oppia's SVS and Third party libraries like cerberus and marshmallow

Point of similarity	Oppia's SVS	Third Party
Validating simple params like string, boolean, Integer.	Simple data types like string(unicode), int, bool are present in existing oppia's SVS. And the schema is further modified by adding validators.	Validation of simple data types like String, Integer, Boolean can be done by using the syntax as guided in official docs. Link: marshmallow. Link: Cerberus
Validating by writing custom validators	Apart simple types of params custom validators can be added in schema utils like is_atleast()	Apart from simple types of params custom validators can be added. Link: marshmallow. Link: Cerberus.

Distinction between Oppia’s SVS and Third party libraries like cerberus and marshmallow.

Point of distinction	Oppia’s SVS	Third Party
Consistent schema structure	Oppia’s SVS is already being used in different parts of the codebase. Using the same schema pattern for the params will help us have consistent schema structure in different parts of the codebase.	Oppia already has its own schema validation system, adding new libs for validation will introduce inconsistent schema patterns for validating data in the codebase.
Learning curve	Oppia developers, working with parts of the codebase like schema_utils, interaction, etc are already familiar with Oppia’s SVS. It will be quite easier for developers to write/review/maintain param schema validation architecture built using Oppia’s SVS.	Introducing a new schema pattern in the codebase will make it comparatively hard for developers to write/review/maintain the param validation system as now developers will have to learn two different schema validation structures.
Dependency	Oppia’s SVS is independent of any other third party libraries so it is modified according to needs.	Using third parties like cerberus or marshmallow over schema utils, increases dependency for the oppia project.
Tracking security/regression issues	Oppia's SVS functionality is being implemented/maintained by Oppia developers, so it will be easy to track major issues and prioritize fixing major issues.	The issue thread discussion as referred to in the examples shows that continuous track should be needed if their feature is working correctly or not. Example 1 Example 2 Example 3 Example 4
Usable validators	The validators in schema_utils.py can be used in other places. Eg: **has_length_atmost** is a	Custom Integration for handlers param, from third parties can be used by writing private methods,

	validator written in schema utils and it is used for validating client data edited by schema_based_editor.directive.ts and music_phrase_editor.directive.ts	like '_xyz()' so they can not be used in other places.
Schema compatibility on client side	The current Oppia's SVS schema structure is used in client side for client-side-validation. Using Oppia's SVS for params validation can provide us an easy way to validate data on the client side before making https requests.	According to official documentation of marshmallow (here), marshmallow can be used to validate package.json.

Conclusion: As both the Oppia's SVS and third-party libraries fulfils the major requirements for the project and from the above similarities and distinction it can be concluded that the Oppia's SVS would be better to use for the project because of the following major reasons:

1. Using Oppia's SVS will provide consistent schema structure for data validation in the codebase.
2. Adding a new validation function for params using Oppia's SVS will provide a reusable validation function for the oppia codebase.
3. Oppia's SVS schema structure can be easily used on the client-side and Oppia already has a structure to validate data using Oppia's SVS schema. In future, param validation can be easily implemented in the client side so that the data is validated in the frontend before reaching the server.

Architectural Overview

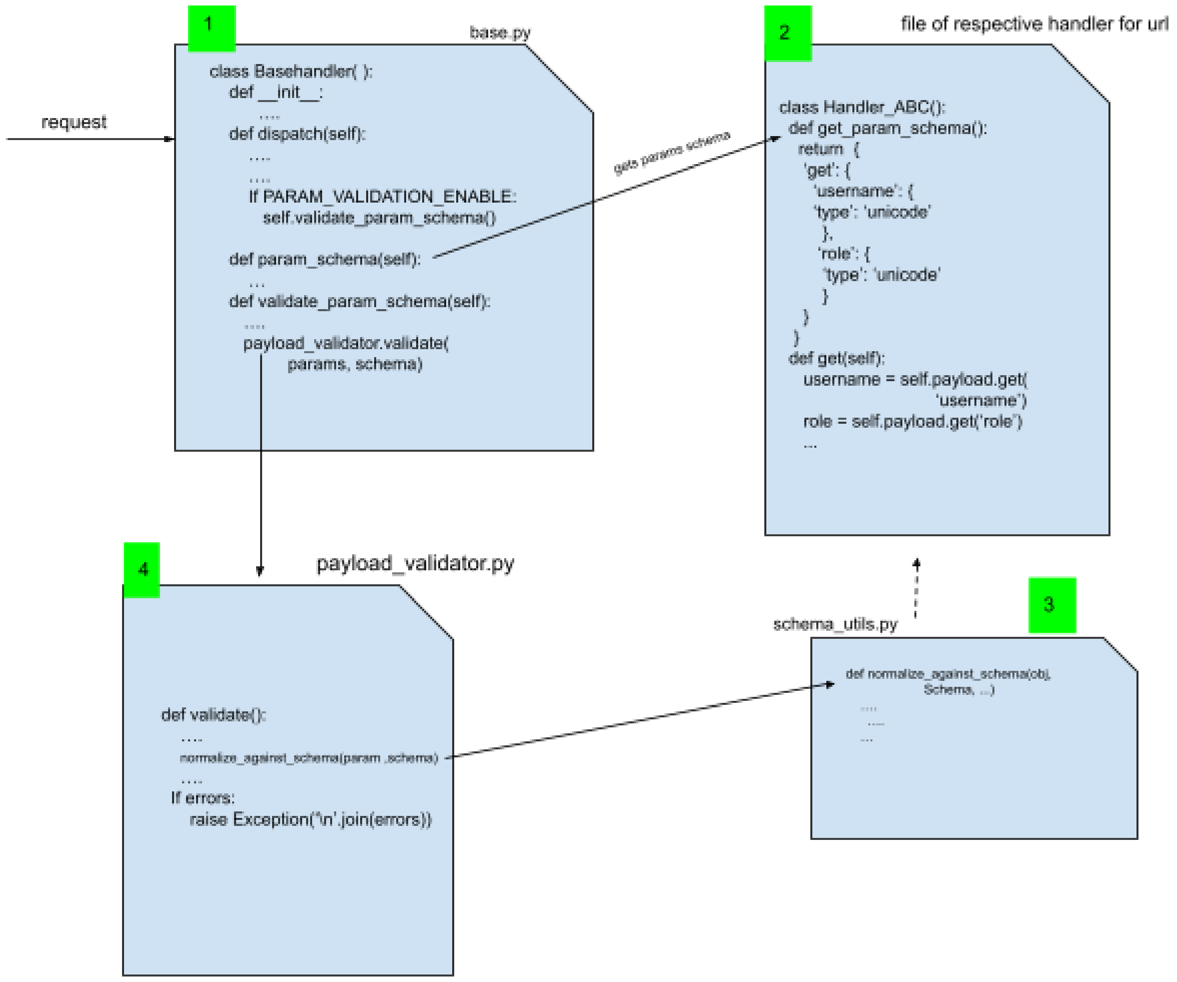
Expected behaviours from the param validation architecture:

1. Validates params schema before handling the request i.e, before going through acl and before calling request method i.e, get/post etc. methods defined in the handler class.
    a. This helps us to ensure that all the data reaching acls and handler's request-method are valid and are in expected pattern.
    b. Benefits for validating before acl:
        i. Not allow reaching acl layer with invalid/malicious data in the payload/param.
        ii. Acl layer needs to make datastore calls, if we validate the schema first then we will have a blocker which won't have to make any datastore call. and can avoid datastore calls for malicious requests.
        iii. This will help us follow the fail fast rule i.e, fail asap without waiting for any datastore calls.
2. Does-not validate any value using datastore data.

    a. As this validation is going to happen as soon as the request reaches the server and before reaching the acl decorators, we will avoid touching the datastore to validate these data.

3. Only validates the schema of the request params and their values.
    a. The validation for the values can include type, expected-values, expected pattern checks.

Architecture



```
class Basehandler( ):
   def __init__:
       ....
   def dispatch(self):
      ....
      ....
      If PARAM_VALIDATION_ENABLE:
        self.validate_param_schema()

   def param_schema(self):
      ...
   def validate_param_schema(self):
     ....
     payload_validator.validate(
            params, schema)
```

```
class Handler_ABC():
  def get_param_schema():
   return  {
    'get': {
     'username': {
     'type': 'unicode'
      },
     'role': {
      'type': 'unicode'
      }
    }
  }
  def get(self):
    username = self.payload.get(
                     'username')
    role = self.payload.get('role')
    ...
```

```
def validate():
   ....
   normalize_against_schema(param ,schema)
   ....
  If errors:
    raise Exception('\n'.join(errors))
```

```
def normalize_against_schema(obj,
                Schema, ...)
  ....
   .....
  ...
```

*Alternative considered section includes another architecture (**[bookmark]()**)*

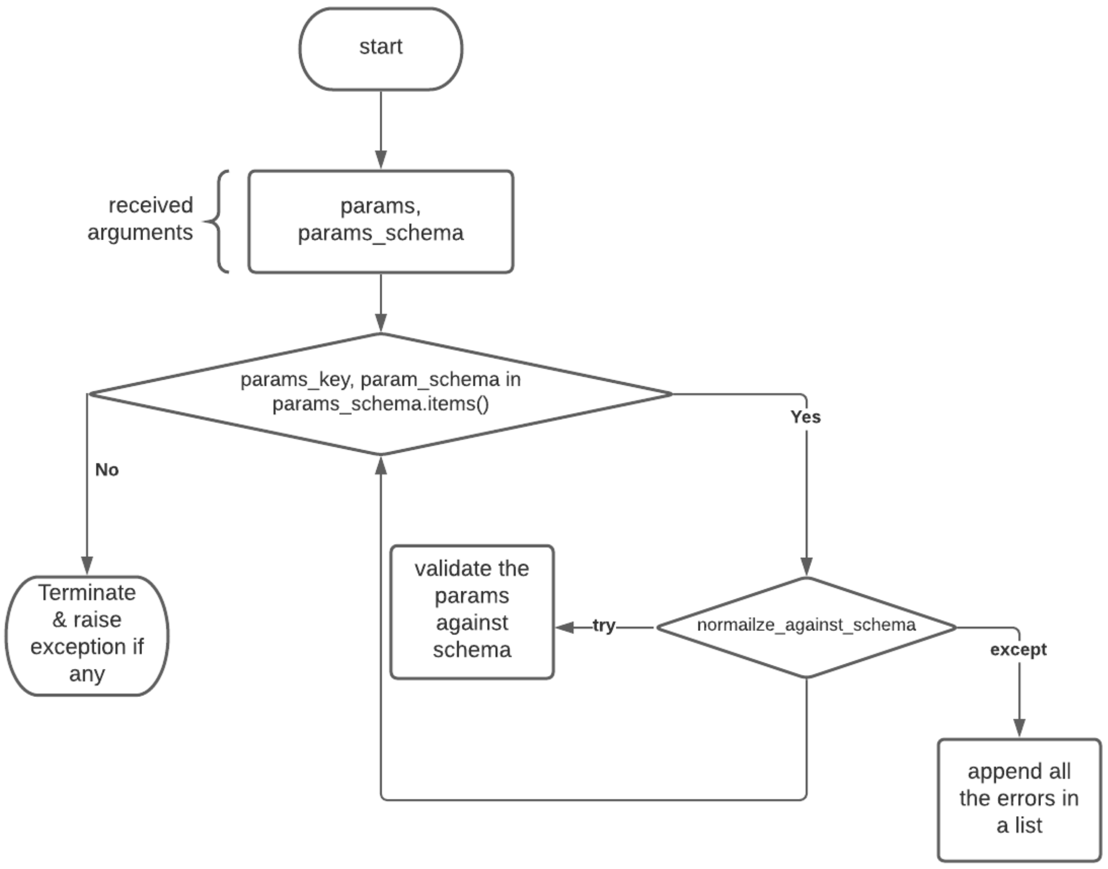
Glossary:

1**base.py**: Represents the BaseHandler class of base,py. Here extraction of params, schema and calling of validate method from payload_validator is done.
2 Represents the file where the **handler class** is present and inside that, get_param_schema( ) method should be present from which the schema of the corresponding url and method is received.
3 **schema_utils**: Represents a file, which contains classes of all the param types and a method to validate param against its schema.
4 **payload_validator.py** a newly added file for validation of handler params.

Flowchart Diagram of validate method

Validate method needs to be defined in payload_validator file (a new file introduced). validate method takes params and its schema for validation and after validation, the method raises an exception (if any).
The flowchart implementation of validate method is given below:

.


start
received arguments
params, params_schema
params_key, param_schema in params_schema.items()
Yes
No
Terminate & raise exception if any
validate the params against schema
try
normailze_against_schema
except
append all the errors in a list

*For Pseudocode Implementation kindly refer here: (**bookmark**)*

Note: *The dotted arrow from file of handler to schema utils represents there is an indirect relation between these two files i.e., the types of the params are defined in schema utils.*

API request journey:

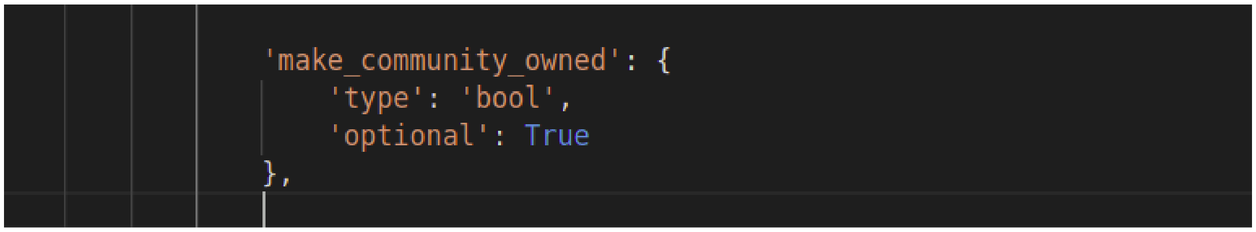
1. Client makes a request to an API.
2. The handler class gets initialized (the initialization method written in **BaseHandler class**.)
3. Check whether PARAM_VALIDATION_ENABLE flag is true, otherwise skip the validation step.
4. For Validation params, schema is extracted for a particular request method, i,e., PUT/ POST/ DELETE.
5. By taking schema, validate_param_schema method calls validate method from a newly added file i.e., payload_validator.py
6. Validate method validates the params and raises errors if any.

7. After validation the corresponding method (get / post / delete) gets called.

   Note: *Kindly refer here for description and pseudocode Implementation.*

Handling cases when a param is optional

Optional cases can be handled by setting 'optional' key to True or false in the schema, and by default every param is treated as a required param.



```
'make_community_owned': {
    'type': 'bool',
    'optional': True
},
```

Here **make_community_owned** is optional and in schema its optional key value is set to True.

Adding/Updating new/existing handler using the param validator architecture:

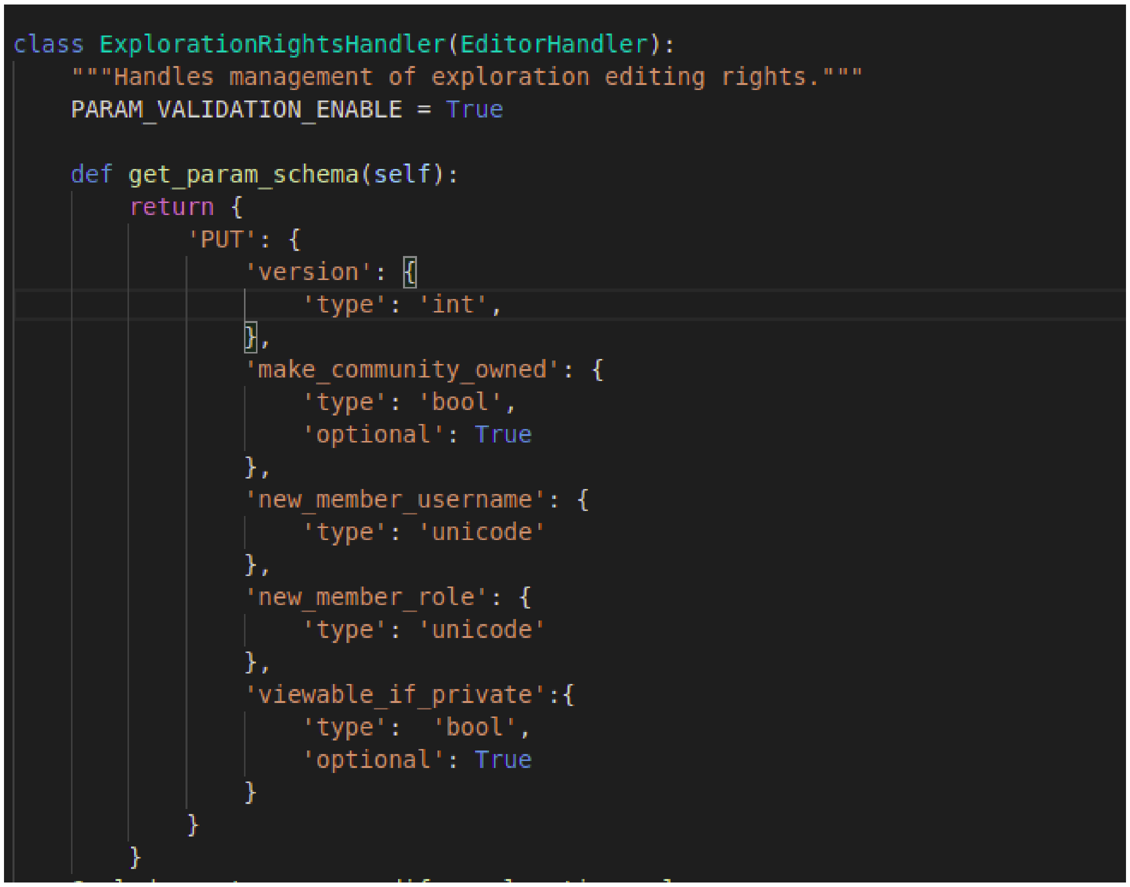
Updating ExplorationHandler(put method) in editor.py to use the param schema validation architecture. Steps to follow :

1. Look for all the params used in the *put* function [fetched using self.payload.get(..)]
   a. Found the following:
      i. version
      ii. commit _message
      iii. change_list
2. Look for corresponding param object type in objects.py
   a. Found the following:
      i. version → objects.Int
      ii. Commit_message → objects.UnicodeString
      iii. Change_list → *nothing found*
3. Define new param objects if needed
   a. Define ExplorationChangeListParam in objects.py. (Details here)
4. Define get_param_schema in ExplorationHandler class.
5. Now the schema validation architecture will ensure the schema validation for the put method.

Testing of above steps for ExplorationRightsHandler in editor.py for giving rights to users in exploration.

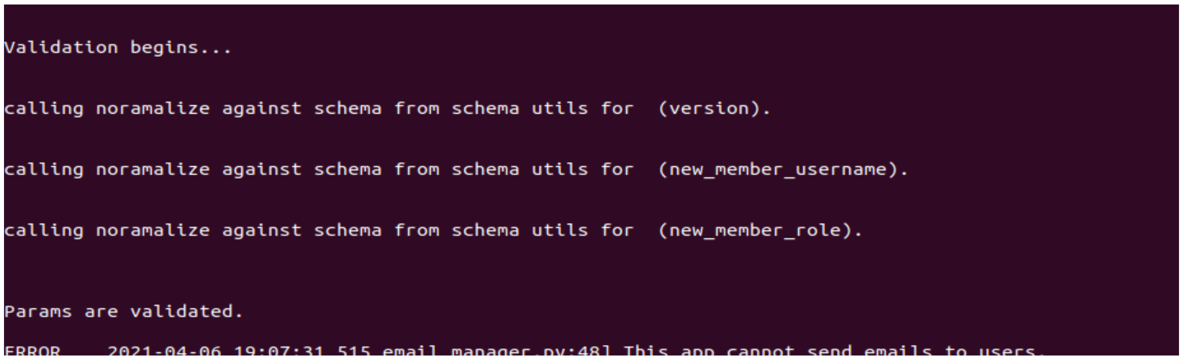
Steps for validating params in a handler.

1. The PARAM_VALIDATION_ENABLE flag in ExplorationRightsHandler is set to True.
2. get_param_schema method should be written, which contains the schema for PUT method.



```
class ExplorationRightsHandler(EditorHandler):
    """Handles management of exploration editing rights."""
    PARAM_VALIDATION_ENABLE = True

    def get_param_schema(self):
        return {
            'PUT': {
                'version': {
                    'type': 'int',
                },
                'make_community_owned': {
                    'type': 'bool',
                    'optional': True
                },
                'new_member_username': {
                    'type': 'unicode'
                },
                'new_member_role': {
                    'type': 'unicode'
                },
                'viewable_if_private':{
                    'type':  'bool',
                    'optional': True
                }
            }
        }
```

3. Now the architecture will validate the params.
4. Proof of validation



```
Validation begins...

calling noramalize against schema from schema utils for  (version).

calling noramalize against schema from schema utils for  (new_member_username).

calling noramalize against schema from schema utils for  (new_member_role).


Params are validated.
ERROR    2021-04-06 19:07:31,515 email_manager.py:48] This app cannot send emails to users.
```

Here temporary print statements are added for understanding. All the required params are validated individually, and logic for validating optional params will be added in the implementation section.

Architecture for incrementally adding schema validators for all handlers

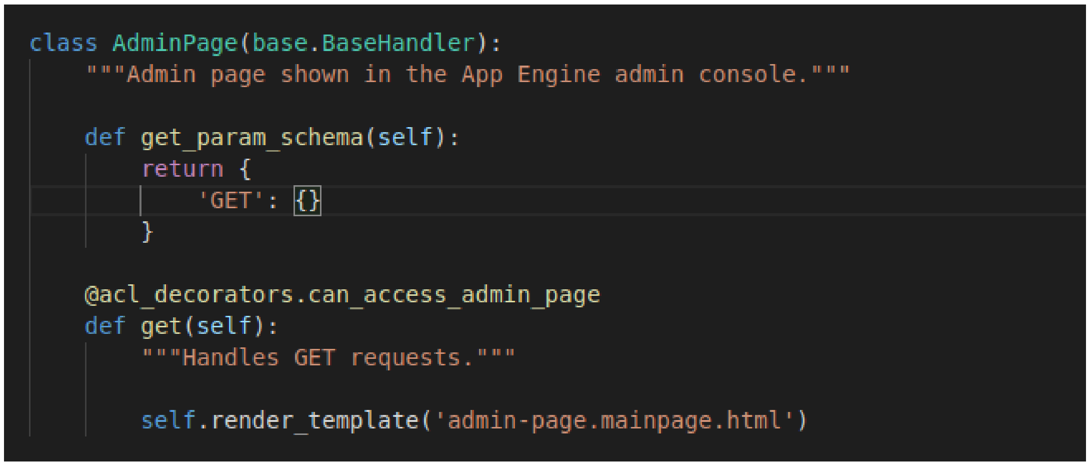
This project is going to implement a new system to validate params in handlers. But there are already 100+ handlers in the codebase. There needs to be a process for incrementally

integrating all the handlers with new params validators, without affecting the development of the existing codebase.
The steps for incrementally adding validators are as follows:

- Define PARAM_VALIDATION_ENABLE in the BaseHandler and set it to False.
- Implement the param schema validation architecture and disable the validation call using the PARAM_VALIDATION_ENABLE flag.
- Enable the PARAM_VALIDATION_ENABLE in handler classes of three files and add schema for all those handlers.
- Disallow writing new handlers without schema:
    - Write a wiki page explaining how to add schema for handler params and announce on oppia-dev@
      For complete reference kindly follow this bookmark.
    - Writing a backend test and lint tests ensures that only the “expected list of handler class” have PARAM_VALIDATION_ENABLE set to False.
- Incrementally add schema for old handlers and remove handler name from “expected list of handler class”
- Once all old handlers have schema and the “expected list of handler class” is empty change the PARAM_VALIDATION_ENABLE to True in BaseHandler and remove PARAM_VALIDATION_ENABLE from all child/main handlers.

Handling param validation for different types of handlers.

1. Param validation for the handler is enabled but schema for the params is not defined.
    a. The backend tests will ensure that if the PARAM_VALIDATION_ENABLE flag is true then get_param_schema() method should be present, which contains the schema for the handler’s param.
2. Handler with some request method doesn’t need any params
    a. When a handler’s request method does not need any params in its request then get_param_schema should return an empty dict as its schema.
    Example:



```
class AdminPage(base.BaseHandler):
    """Admin page shown in the App Engine admin console."""

    def get_param_schema(self):
        return {
            'GET': {}
        }

    @acl_decorators.can_access_admin_page
    def get(self):
        """Handles GET requests."""

        self.render_template('admin-page.mainpage.html')
```

3. Incoming request with extra or missing params.

   a. Validate method in payload_validate file should ensure that any required data should not be missed in params and any extra data should not be passed to params. And if the validate method receives this kind of data then it will raise an exception.

4. Handlers containing value in their URL.
   a. **_get_url_param()**
   b. _get_url_param() should be defined in every handler class.
      i. This method returns the schema for url_params
      ii. The handler class which does not contain url_params, then this method returns an empty dict.
      iii. The handler class which contains the url_params but they do not have any specific schema to validate, then the value of the key in the schema is empty.

```
class ExplorationRightsHandler(EditorHandler):
   """Handles management of exploration editing rights."""

   PARAM_VALIDATION_ENABLE = True
   def _get_url_param(self):
       return {
           'exploration_id': {}
       }
```

      iv. The handler class which contains the url_params and the params require a specific schema for validation, then the schema contains additional keys and values like, validators, choices, etc.

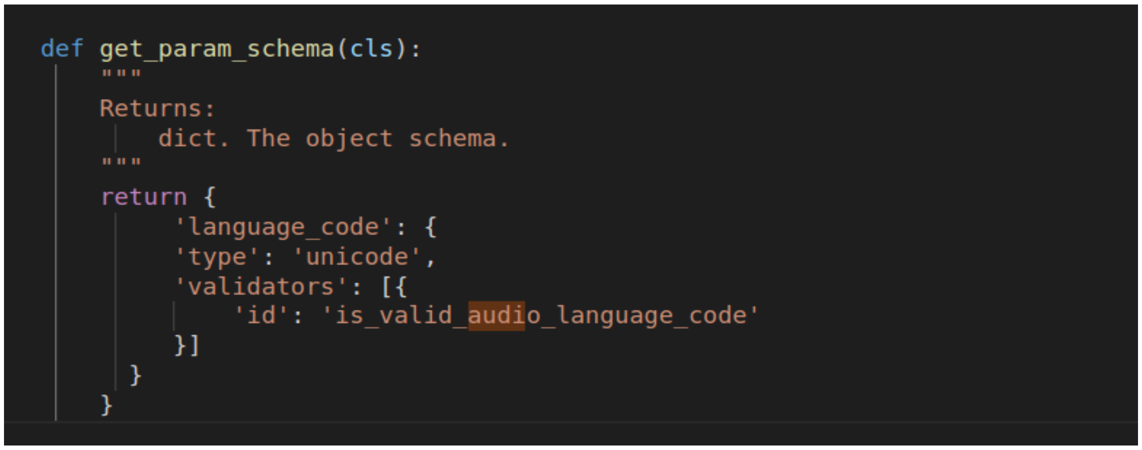
5. Handlers in which SVS is not ready
   a. In order to disable param validation check, the PARAM_VALIDATION_ENABLE flag needs to be set to False explicitly, in the handler class.
      i. By disabling the flag, helps me to incrementally add SVS to all existing handlers.
   b. Once all old handlers have schema and the “expected list of handler class” is empty change the PARAM_VALIDATION_ENABLE to True in BaseHandler and remove PARAM_VALIDATION_ENABLE from all child/main handlers.

Adding new validator for a param: [3 examples below]

Assume a developer wants to add a param validator for audio language_code.

Steps:

1. Define get_param_schema method in the handler class, which returns the schema of the language_code param. Here type of the language_code is unicode and is_valid_language_code is passed as validator.



```
def get_param_schema(cls):
    """
    Returns:
        dict. The object schema.
    """
    return {
        'language_code': {
        'type': 'unicode',
        'validators': [{
            'id': 'is_valid_audio_language_code'
        }]
      }
    }
```

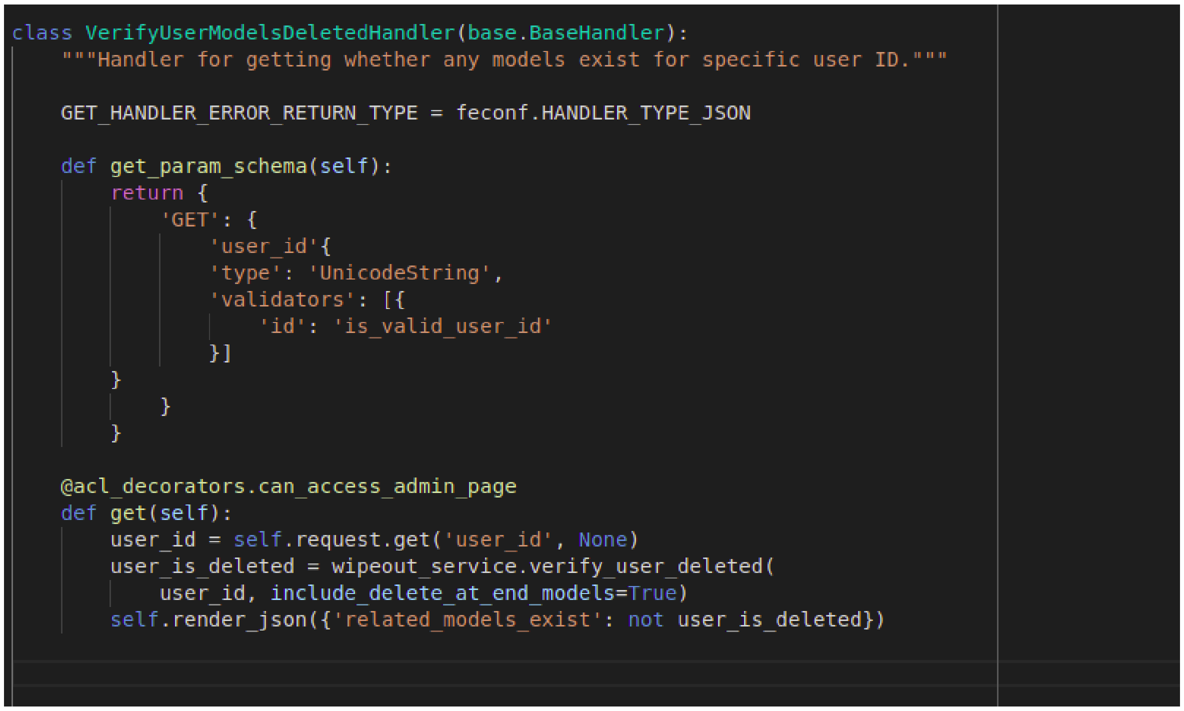
2. Found is_supported_audio_language_code in schema_utils.py, thus simply used in the schema.

```
    @staticmethod
    def is_supported_audio_language_code(obj):
        """Checks if the given obj (a string) represents a valid language
code.

        Args:
            obj: str. A string.

        Returns:
            bool. Whether the given object is a valid audio language code.
        """
        return utils.is_supported_audio_language_code(obj)
```

3. objects.AudioLangaugeCodeParam is ready for use in any handler.

Assume a developer wants to add param validator for user_id:

steps:
1. Define the schema in the handler class and provide validator as is_valid_user_id.
2. is_valid_user_id is already defined in schema utils.
3. Thus VerifyUserModelsDeletedHandler is ready for use.

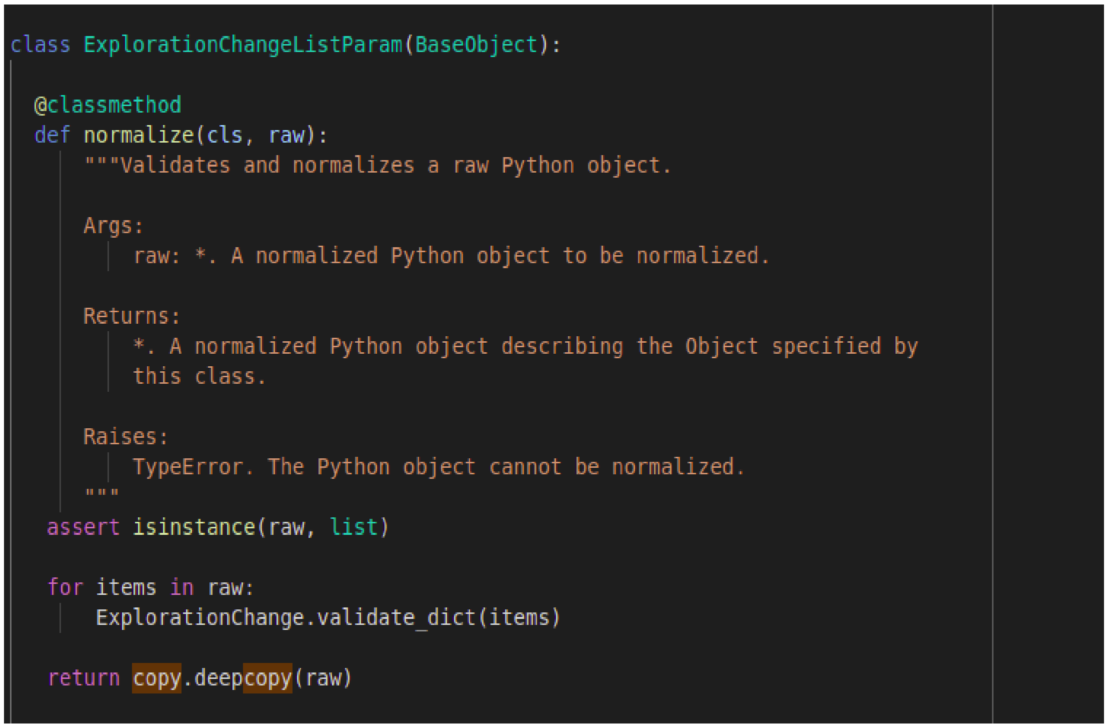


```
class VerifyUserModelsDeletedHandler(base.BaseHandler):
    """Handler for getting whether any models exist for specific user ID."""

    GET_HANDLER_ERROR_RETURN_TYPE = feconf.HANDLER_TYPE_JSON

    def get_param_schema(self):
        return {
            'GET': {
                'user_id'{
                'type': 'UnicodeString',
                'validators': [{
                    'id': 'is_valid_user_id'
                }]
        }
            }
        }

    @acl_decorators.can_access_admin_page
    def get(self):
        user_id = self.request.get('user_id', None)
        user_is_deleted = wipeout_service.verify_user_deleted(
            user_id, include_delete_at_end_models=True)
        self.render_json({'related_models_exist': not user_is_deleted})
```

Assume a developer wants to add a param validator for ExplorationChangeList

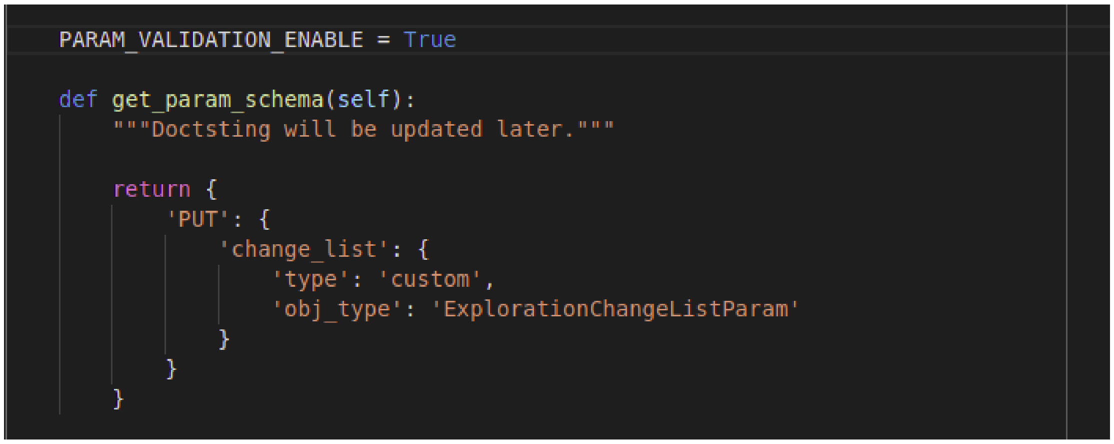
Steps:

1. Define **ExplorationChangeListParam** in objects.py



```
class ExplorationChangeListParam(BaseObject):

  @classmethod
  def normalize(cls, raw):
      """Validates and normalizes a raw Python object.

      Args:
          raw: *. A normalized Python object to be normalized.

      Returns:
          *. A normalized Python object describing the Object specified by
          this class.

      Raises:
          TypeError. The Python object cannot be normalized.
      """
    assert isinstance(raw, list)

    for items in raw:
        ExplorationChange.validate_dict(items)

    return copy.deepcopy(raw)
```

2. Write get_schema_method in handler class



```
PARAM_VALIDATION_ENABLE = True

def get_param_schema(self):
    """Doctsting will be updated later."""

    return {
        'PUT': {
            'change_list': {
                'type': 'custom',
                'obj_type': 'ExplorationChangeListParam'
            }
        }
    }
```

3. Now the change_list param is ready for use.

When to write object classes and when to write validator methods.

Steps for making a decision, when to add new validators in schema_utils.py and when to add new object class in objects.py.

1. Observe the params type if it is simple and just needs little modification in property, then writing validators is the correct choice.
    a. Example 2 (here) mentions that when the param type is simple and as per requirement it's schema can be modified by adding validators from schema utils.
2. When the type of the param is a bit complex like dict and its validate method is written in domain class then writing its object class should be the most optimum way.
    a. Example 3 (here) mentions that when the param type is dict and the validator method is written in domain class then the schema should be of type custom and obj_type key in the schema should contain the name of the newly added object class

Implementation Approach

The project implementation part can be divided into three parts:
1. Implementation of the param schema validation architecture
2. Test/check to enforce developers to use schema validators for new handlers
3. Incrementally integrating schema validators for existing handlers.

Description of methods and variables

1. PARAM_VALIADTION_ENABLE
    a. Variable with default value, False and set to True in the handler, when a handler class is ready for schema validation.
2. validate_param_schema()
    a. Method defined is the BaseHandler class. Used in order to validate params against schema by calling validate method form payload_validator.py file (new file written for validation of schema)
3. validate()
    a. Method defined in the payload_validator file validates the params against the schema.

Implementation of code for validating handler params

[A] : Add new class/function.
[U] : Update existing class/function/variable.

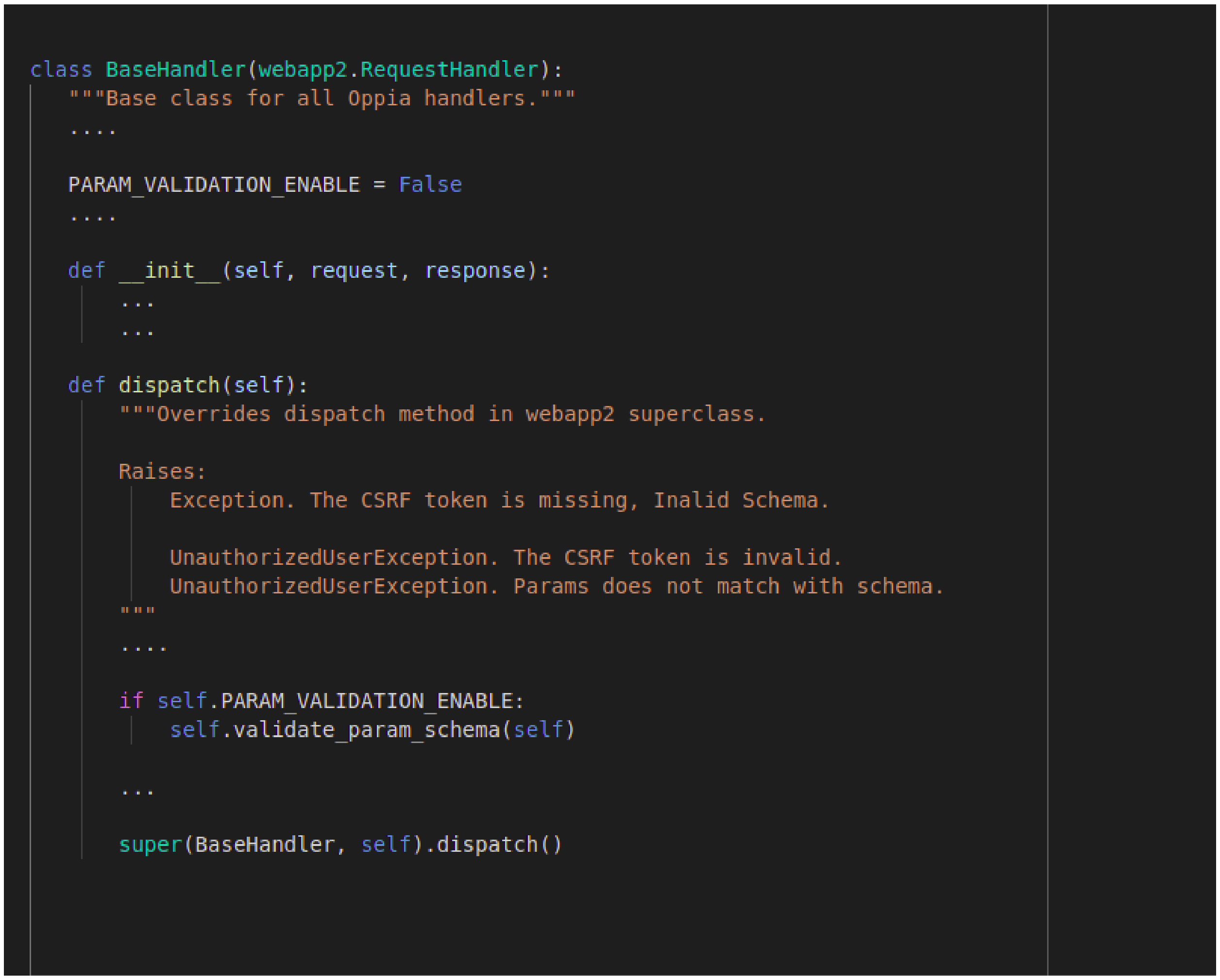
- Controllers
    - [U] BaseHandler class in base.py
        - [U] dispatch method
            - If PARAM_VALIDATION_ENABLE, call validate_param_schema() method.
        - [A] validate_param_schema()
            - Extract the schema
                - Take all the schema from the handler class for a request method.
            - Extract all the params.
                - Extract all types of params that need validation.
                - request_params
                    - Use of **self.request.get(payload)**
                - url_params
                    - Use of **self.request.route_kwargs**
                - query_string_params
                    - Use of **self.request.query_string**
            - Call validate() method,
                - Call validate method from payload validator.

            - Raise all the exceptions if any.
    - [A] payload_validator.py
        - [A] validate()
            - Iterate over all the items passed through schema.
            - Checks whether a particular key is optional or not.
            - If optional and not passed in params then don't raise any exception.
            - If schema and params both are present.
                - Within the try block call normalized_against_schema from schema utils by passing the params individually and its corresponding schema.
                - Within the except block collect all the error messages and raise all the errors, once all the params validated.
            - Check for any extra or missing params in param dict and append to the list of errors.
            - Return the list of errors back to the validate_param_schema() method.

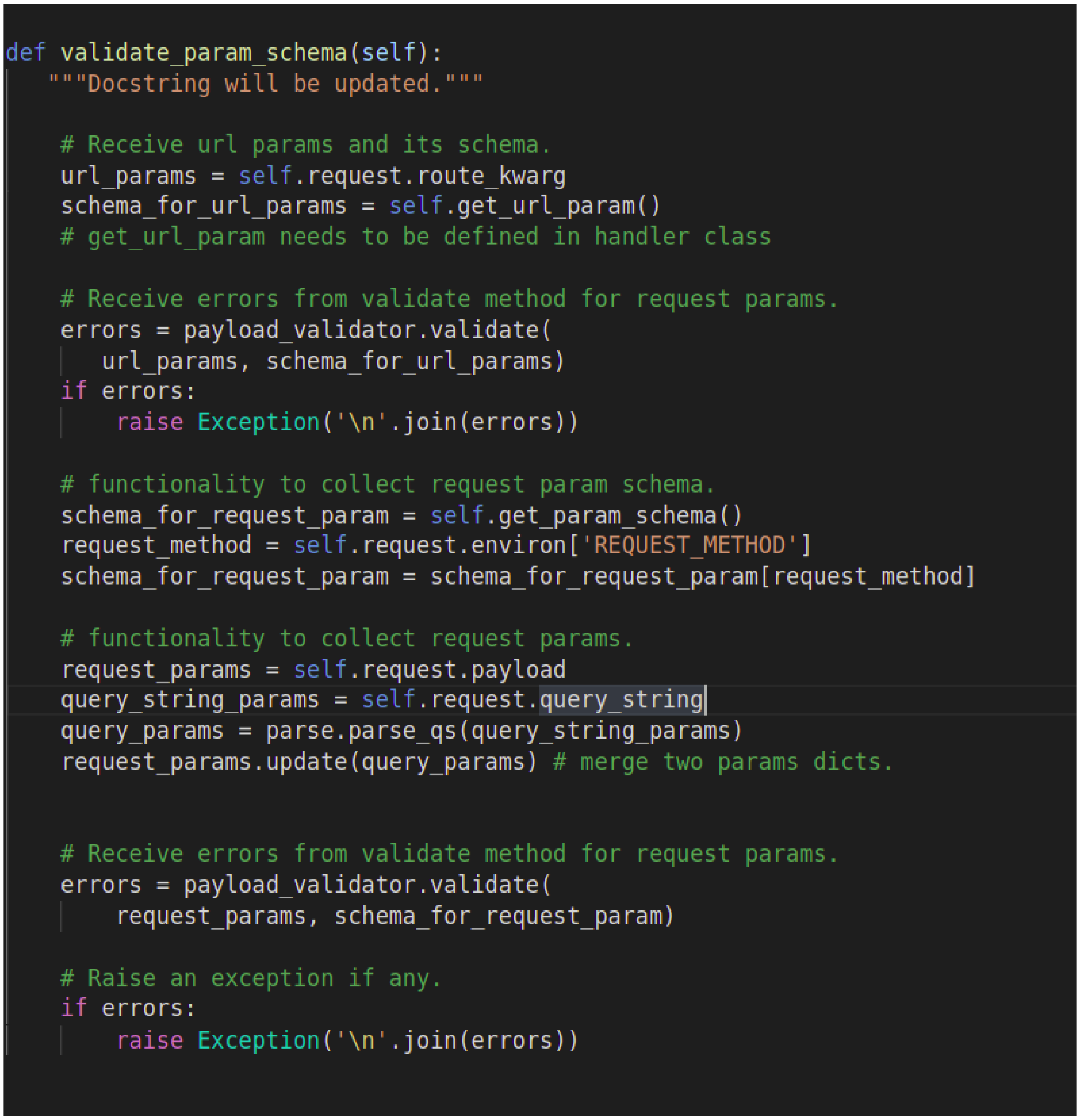
Pseudocode Implementation

Here (…) in the code signifies the existing code, which is untouched and removed temporarily for clear vision of newly added lines.

base.py

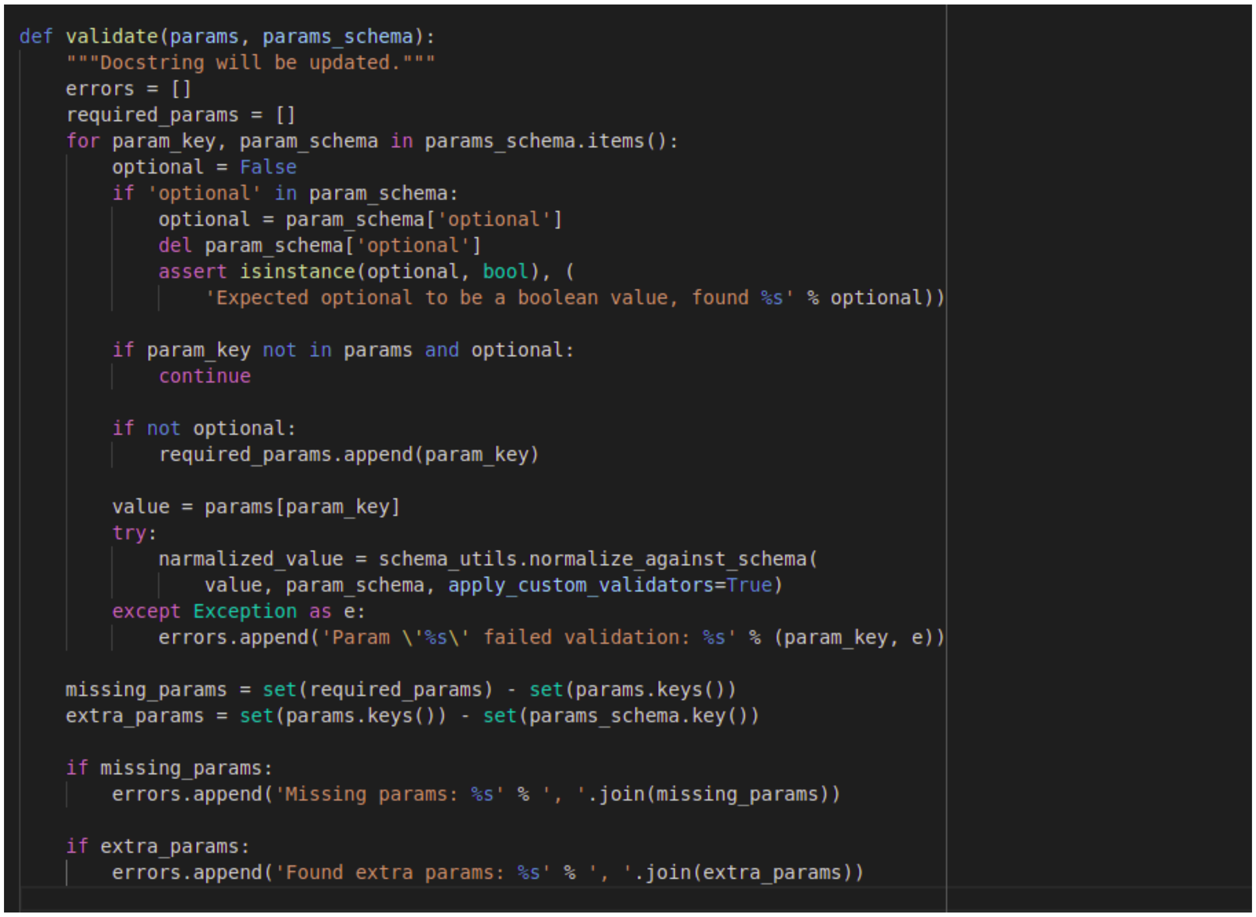

```
class BaseHandler(webapp2.RequestHandler):
    """Base class for all Oppia handlers."""
    ....

    PARAM_VALIDATION_ENABLE = False
    ....

    def __init__(self, request, response):
        ...
        ...

    def dispatch(self):
        """Overrides dispatch method in webapp2 superclass.

        Raises:
            Exception. The CSRF token is missing, Inalid Schema.

            UnauthorizedUserException. The CSRF token is invalid.
            UnauthorizedUserException. Params does not match with schema.
        """
        ....

        if self.PARAM_VALIDATION_ENABLE:
            self.validate_param_schema(self)

        ...

        super(BaseHandler, self).dispatch()
```

validate_param_schema()



```python
def validate_param_schema(self):
    """Docstring will be updated."""

    # Receive url params and its schema.
    url_params = self.request.route_kwarg
    schema_for_url_params = self.get_url_param()
    # get_url_param needs to be defined in handler class

    # Receive errors from validate method for request params.
    errors = payload_validator.validate(
        url_params, schema_for_url_params)
    if errors:
        raise Exception('\n'.join(errors))

    # functionality to collect request param schema.
    schema_for_request_param = self.get_param_schema()
    request_method = self.request.environ['REQUEST_METHOD']
    schema_for_request_param = schema_for_request_param[request_method]

    # functionality to collect request params.
    request_params = self.request.payload
    query_string_params = self.request.query_string
    query_params = parse.parse_qs(query_string_params)
    request_params.update(query_params) # merge two params dicts.


    # Receive errors from validate method for request params.
    errors = payload_validator.validate(
        request_params, schema_for_request_param)

    # Raise an exception if any.
    if errors:
        raise Exception('\n'.join(errors))
```

payload_validator.py



```python
def validate(params, params_schema):
    """Docstring will be updated."""
    errors = []
    required_params = []
    for param_key, param_schema in params_schema.items():
        optional = False
        if 'optional' in param_schema:
            optional = param_schema['optional']
            del param_schema['optional']
            assert isinstance(optional, bool), (
                'Expected optional to be a boolean value, found %s' % optional))

        if param_key not in params and optional:
            continue

        if not optional:
            required_params.append(param_key)

        value = params[param_key]
        try:
            narmalized_value = schema_utils.normalize_against_schema(
                value, param_schema, apply_custom_validators=True)
        except Exception as e:
            errors.append('Param \'%s\' failed validation: %s' % (param_key, e))

    missing_params = set(required_params) - set(params.keys())
    extra_params = set(params.keys()) - set(params_schema.key())

    if missing_params:
        errors.append('Missing params: %s' % ', '.join(missing_params))

    if extra_params:
        errors.append('Found extra params: %s' % ', '.join(extra_params))
```

return errors

Implementation of Lint checks.

Pseudo algo:

```
1. For invalid case with new handler:
    a. Swap OLD_HANDLER_NAMES with [‘SomeOldHandler’]
    b. Create dummy_code:
        i. Class NewHandler(base.BaseHandler)
                Pass
    c. Create nodeastroid.extract_node(dummy_code)
    d. Run a new lint check on the ast and expect to raise a message.
2. For valid case with new handler:
    a. Swap OLD_HANDLER_NAMES with [‘SomeOldHandler’]
    b. Create dummy_code:
        i. Class NewHandler(base.BaseHandler)
              PARAM_VALIDATION_ENABLED = true

    c. Create nodeastroid.extract_node(dummy_code)
    d. Run the new lint check on the ast and expect to not raise any message.
3. For old handler not enabled validation:
    a. Swap OLD_HANDLER_NAMES with [‘SomeOldHandler’]
    b. Create dummy_code:
        i. Class ‘SomeOldHandler’(base.BaseHandler)
```

```
                    pass
```

c. Create nodeastroid.extract_node(dummy_code)
d. Run the new lint check on the ast and expect to not raise any message.

Incrementally integrating schema validators for existing handlers

Work estimation

Name of the files

The name of all the files which are under the scope of this project are:

- Milestone 1: Name of the files for milestone 1.
    - *admin.py*
    - *classifier.py*
    - *classroom.py*
- Milestone 2: Name of the files for milestone 2
    - *collection_editor.py*
    - *collection_viewer.py*
    - *concept_card_viewer.py*
    - *contributor_dashboard.py*
    - *creator_dashboard.py*
    - *cron.py*
    - *custom_landing_pages.py*
    - *editor.py*
    - *email_dashboard.py*
    - *features.py*
    - *feedback.py*
    - *improvements.py*
    - *incoming_emails.py*
    - *learner_dashboard.py*
    - *learner_playlist.py*
    - *library.py*
    - *moderator.py*
    - *pages.py*
    - *platform_feature.py*
    - *practice_sessions.py*

Note: Took a deeper look into each of the files/handlers and wrote all the detailed requirements and the schema for each param in a spreadsheet.

The extracted and compressed details from the spreadsheet are as follows:

Data	Frequency
Number of files	23
Number of handler classes	126
Number of Get methods	100
Number of Put methods	11
Number of Post methods	26
Number of delete methods	6
Number of params	124

Number of new object classes to be implemented

Name of new class	Description	Usage
ExplorationChangeParam	A new class with this name should be created which contains a schema of type dict. It should be implemented, as shown here.	This class is expected to be used for validating **3** params .
ExplorationTaskEntriesParam	A new class with this name should be created which contains a schema of type dict.	This class is expected to be used for validating **1** param.
NewRulesParam	A new class with this name should be created which contains a schema of type dict.	This class is expected to be used for validating **1** param.

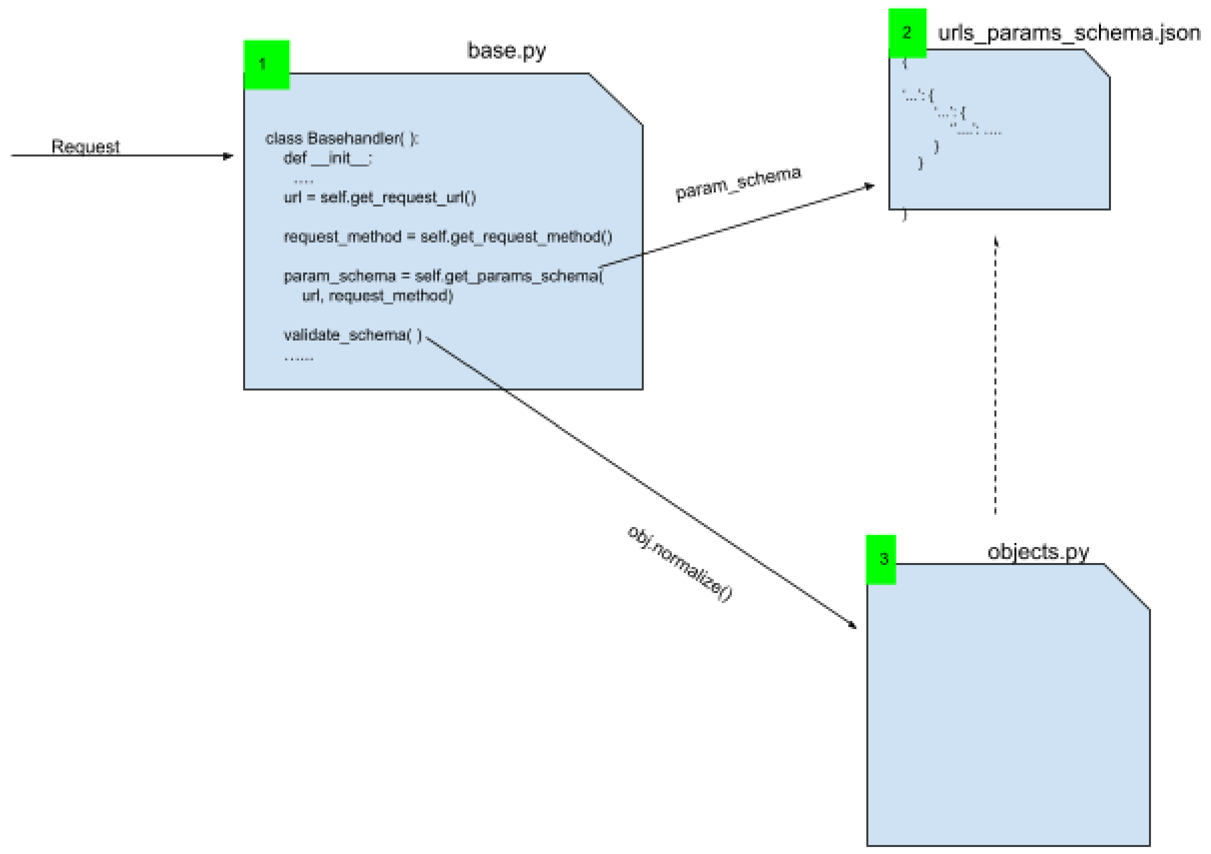
Occurrence of the each type of param under scope of this project

Type of param	Frequency
int	21
unicode	91
bool	5

list	5
dict	7

Alternative Considered*

Architecture 1



```
class Basehandler( ):
   def __init__:
     ....
   url = self.get_request_url()

   request_method = self.get_request_method()

   param_schema = self.get_params_schema(
      url, request_method)

   validate_schema( )
   ......
```

```
{

'...': {
        '...': {
            "....': ....
        }
    }

}
```

Glossary:

1 **base.py**: Represents the __init__ constructor of Basehandler class in base.py. This constructor will hold the extraction and validation of params.
2 **urls_params_schema.json**: Represents a file from which we get the schema of the corresponding url and method. This file should contain urls and their corresponding schema.
Eg:
```
{
	'/library': {
		'GET': {
			'Subject': 'UnicodeStringObject',
			'Language_code': 'ContentLanguageCodeObject'
		},
		'POST': {...}
	}
}
```

3 **objects.py**: Represents a file from which we call normalize method for validating the params. Typed objects are initialized from a raw python object. They are validated and normalized to basic python objects (primitive types combined via lists and dicts.)

Note: The dotted arrow from file urls.json to **objects.py** represents there is an indirect relation between these two files. As for validation of objects formed with help of schema (form urls.json),
should be normalized by calling methods from objects.py

Handling cases when a param is optional
When a param is optional, then param name should be prefixed with '*' in param name.
Eg:
```
{
	'/library': {
		'GET': {
			'subject': 'UnicodeStringObject',
			'*language_code': 'ContentLanguageCodeObject'
		},
		'POST': {...}
	}
}
```
Here Language_code is optional i.e., users may or maynot give language code.

Handling cases when multiple type for param is allowed:
When a param accepts multiple types then the schema should contain both of the types.
Eg:

```
{
        '/library': {
                'GET': {
                        'Subject': 'UnicodeStringObject',
                        'Language_code': ['ContentLanguageCodeObject' , 'StringObject']
                },
                'POST': {...}
        }
}
```
Here Language_code requires any of the two data types i.e, ContentLanguageCodeObject or StringObject.

API request journey:

1. Client makes a request to an API.
2. The handler class get initialized (the initialization method written in BaseHandler class.)
3. While Initializing the handler class, validate the params passed in the request. Validation algorithm in brief:
    a. Extract all params from self.request.get_all() and store it in received_params.
        i. received_params = self.request.get_all( )
    b. Find the request url using the request object.
    c. Find the request method i.e., either PUT/ POST/ DELETE
        i. method = self.request.environ['REQUEST_METHOD']
    d. We take the schema of the corresponding url and method from urls.json
        i. params_schema = extract param schema using url and method

    e. Now we have to validate_all_required_params_exist( )
4. After validation the corresponding method (get / post / delete) gets called.

Architecture : here

Pros for architectures :

Architecture 1	Architecture 2
Unified place for all the urls and params, in future urls_param_schema.json can be used in the frontend to define the type of request params. [In typescript]	The schemas are defined in the handler itself makes it easier to update/maintain/read.

Cons for architectures:

Architecture 1	Architecture 2
The schemas are not defined in the handler class will make an isolated and independent handler class depend on other files/json.	The schemas are defined in the handler class and won't allow us to re-use them in other places to avoid duplicate structure/code like defining types of params in the frontend.
The chances of error in this architecture is greater because json files are not dynamically connected to python files.	

As per discussion with mentor (Vojtech) Architecture 2 got approval because of the reasons as mentioned in the cons of architecture 1.

Third-party Libraries*

*Implementation of this project does not include any third party libraries. As per requirements of the proposal for this project, integrate all the handlers with new params validators, without affecting the development of existing codebase. Proposal must include comparison between oppia's SVS and third party libraries. The comparison is given above (**Bookmark**) which mentions that oppia's SVS is favourable for validating the handlers params.*

Testing Approach

The backend tests and linter tests will ensure that all the handlers should have schema in it, so the handlers param is validated by SVS.

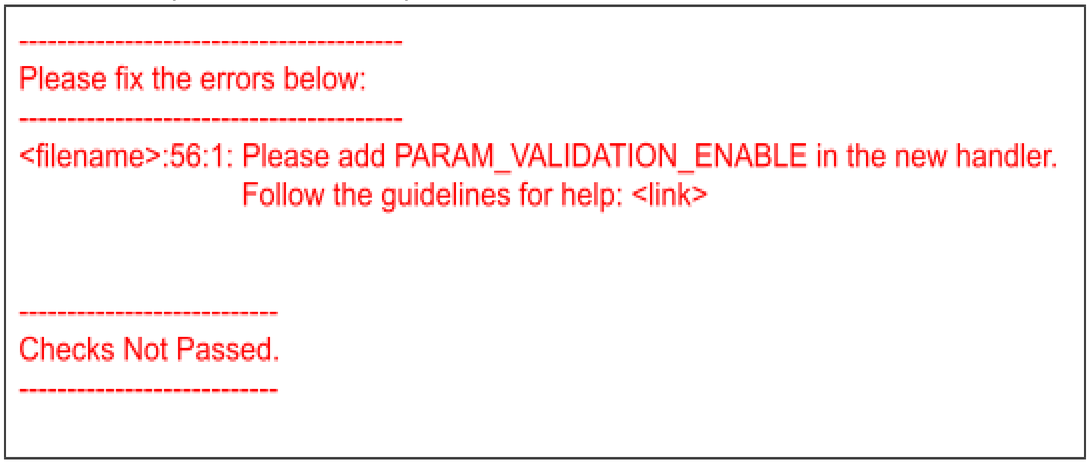
Pseudocode Implementation for lint checks:

```
Declare all the handler class which doesn't need verification in
handlers_without_param_schema.py

handlers_without_param_schema.py:
```

```
OLD_HANDLER_NAMES = [
   'HandlerName1',
   'HandlerName2',
    ....
]


In pylint_extensions.py:

import OLD_HANDLER_NAMES from handlers_without_param_schema

class DisallowNewHandlerWithoutParamSchema():

   __implements__ = interfaces.IAstroidChecker
   name = 'disallow-new-handler-without-param'
   priority = -1
   msgs = {
       'C0036': (
           'Please add PARAM_VALIDATION_ENABLE in the new handler.'
           'Follow the guidelines for help: <link>,
           'no-flag-in-new-handler'
       ),
       'C0037': (
           'Please add get_param_schema function in the new handler.'
           'Follow the guidelines for help: <link>,
           'no-schema-function-in-new-handler'
       ),
   }

   def visit_classdef(node):
       if not "base.BaseHandler" in node.mro():
           return

       if node.name in OLD_HANDLER_NAMES:
           return

       if  not node.local_attr('PARAM_VALIDATION_ENABLE'):
           self.add_message('no-flag-in-new-handler', node=node)

       if 'get_param_schema' not in node.mymethods():
           self.add_message('no-schema-function-in-new-handler', node=node)
```

Mocks for errors generated by linters.\

1. When developers don't enable prama validation in a new handler.



```
-----------------------------------------
Please fix the errors below:
-----------------------------------------
<filename>:56:1: Please add PARAM_VALIDATION_ENABLE in the new handler.
                     Follow the guidelines for help: <link>


----------------------------
Checks Not Passed.
----------------------------
```

If the param validation is enabled for a handler then it won't work if the handler doesn't have the correct schema for the params/url. So we don't need any other check for the new handler

Milestones

The details of work that need to be done under the scope of this project is given below:

Milestone 1

Key Objective:

The key objective in milestone 1 is given as following as:

- Implementation of architecture for schema validation of handler params.
- Implements backend and linter tests for architecture.
- Documenting schema validation for adding to wiki.
- Implement schema schema in files:
  - admin.py
  - classifier.py
  - classroom.py
- Create starter issue for new contributors on how to add handlers params.
  - List of all the files which needs validation and are not in the scope of this project.
    - profile.py
    - question_editor.py
    - question_list.py
    - reader.py
    - recent_commits.py
    - resource.py
    - review_tests.py

- skill_editor.py
- story_viewer.py
- subscription.py
- subtopic_viewer.py
- suggestion.py
- tasks.py
- topic_ediotr.py
- topic_viewer.py
- topic_and_skill_dashboard.py
- voice_artist.py

Summary

Data	Frequency
Number of files	3
Number of handler classes	22
Number of params	48

Number of pull requests.

Total number of pull requests for milestone1 should be **3**

No.	**Description of PR**	**Prereq PR numbers**	**Target date for PR submission**	**Target date for PR to be merged**
1.1	Implements Architecture and its testing for schema validation.	*None*	15/06/21	21/06/21
1.2	Write Documentation for schema validation	1.1	22/06/21	26/06/21
1.3	Implement schema validation for files (classroom, classifier, admin)	1.1	29/06/21	07/07/21
1.4	Create starter issues for new contributors for remaining files.	*none*	03/07/21	*none*

Milestone 2

Key Objective:
The key objective in milestone 1 is given as following as:

- Implement schema schema in files:
  - collection_editor.py
  - collection_viewer.py
  - concept_card_viewer.py
  - contributor_dashboard.py
  - creator_dashboard.py
  - cron.py
  - custom_landing_pages.py
  - editor.py
  - email_dashboard.py
  - features.py
  - feedback.py
  - improvements.py
  - incoming_emails.py
  - learner_dashboard.py
  - learner_playlist.py
  - library.py
  - moderator.py
  - pages.py
  - platform_feature.py
  - practice_sessions.py

Summary

Data	Frequency
Number of files	20
Number of handler classes	104
Number of params	76

Number of pull requests.

Total number of pull requests for milestone2 should be **4**

Table for PR description and target planning

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Implement schema validation for files (collection_editor, collection_viewer, concept_card_viewer, contributor_dashboard, email_dashboard)	1.1	06/07/21	13/07/21
2.2	Implement schema validation for files (creator_dashboard, cron, custom_landing_pages, editor, features)	1.1	15/07/21	23/07/21
2.3	Implement schema validation for files (improvements, incoming_emails, learner_dashboard, learner_playlist, library)	1.1	25/07/21	01/07/21
2.4	Implement schema validation for files (feedback, moderator, pages, platform_feature, practice_session)	1.1	03/08/21	09/08/21

Here every pull request is planned in a way that the effort for working on each pr is more or less equal. As because there are some files which do not have any params in their handlers, some files which have 5-6 handler classes while some of the files are a bit large like editor.py. Thus working on very large and very small files simultaneously, reduces the effort over the entire pr.

Optional Sections

Additional Project-Specific Considerations

Privacy

This feature does not collect any user data as it only validates the data of the handler params by help of an architecture which helps in non data store link validation.

Security

This feature does not have any security consideration as it only validates the data of the handler params by help of an architecture which helps in non data store link validation..

Accessibility

The project adds a backend structure for validating handler's params. It doesn't add/remove/update any user-facing UI features.
Thus this feature does not have any accessibility consideration.

Documentation Changes

Documentation needs to be added in oppia wiki on How to add schema in handler class.

- Introduction
- How to add schema for params in the handler
- How to write tests for the handler class.
- Steps to add new objects class .
- Details on when to object class and when to write validator method
- Example on how to write custom validators for some complex example like list of dicts.

Ethics

This feature does not have any ethical consideration.

Future Work

Future work can be summarised as:

1. Starter Issue created in milestone 1 for adding schema to remaining handlers.
2. If any new handler is introduced in the codebase then SVS will ensure that the handler must have a schema.
    a. Thus addition of schema for new handlers is under the scope of future work.