Google Summer of Code 2022

# Make Backend Code Typed

Sahil jhangar

# Section 1: About You

### What project are you applying for?

Make backend code typed

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

I'm really fascinated by the spirit of oppia organization, which brings different contributors from different countries to build a platform that provides quality education to all the students irrespective of their status and language. I think every student has the right to get basic education and oppia is really heading towards this goal at an exponential rate.

Also, I had a great experience of learning and contributing to oppia. While I was in the lace quality team (LaCE quality) I learned a lot from the mentors and fellow contributors, which is not possible to learn without the help of this community.

I chose this project because I wanted to make the developer's experience even more smoother and make code even more self-documented for new contributors. Also after completion of this project, developers have a good idea of the flow of code which helps developers to make even more error-prone contributions. After working with the LaCE quality team for 6-7 months, I think I have a pretty good understating of oppia codebase which helps me during this project when I'm debugging the code for typing.

### Prior experience

- I have been contributing to oppia for 6+ months as a member of the LaCE quality team and LaCE android team, and I also have a decent knowledge of oppia's codebase.
- I'm also continuously participating in release testing from November 2021.
- I have currently 26 merged PRs and I also did one debugging doc. Which you can find here.
- I have been working with python for 1.5+ years. I also did a backend project.
    - CRICKET-API-PROJECT - technologies used for this project includes Django, Django-rest-framework, beautiful soup, and python's typing module. Every functionality in this project is tested with TDD (test-driven development).

I strongly believe that this project along with my experience with Oppia for the past six months should help me continue my GSoC journey smoothly.

**Links to PRs:**

- Added Mypy type annotations to user_domain.py (#15057)
- Added Mypy type annotations to activity_services.py (#14986)
- Removing all annotated files from core/jobs. (#15228)
- Added Mypy type annotations to collection_domain.py (#14958)
- Fixes topic editor tab when wrong URL fragment is entered in Topic URL Fragment (#14903)

***Link to all my contributions to oppia is HERE***

*I have also raised some issues:*

- #*15224*
- #*14404*
- #*14787*

## Project size

Large (~350 hours).

## Project timeframe

From 13th June 2022 to 3rd October 2022.

I want to increase the project's timeframe because this project includes a huge number of files (**467 files**). So, to accommodate all these files in reasonable sized PR I want extra 3 weeks. However, this timeframe can be decreased if some files are already annotated before the 13th of June ( starting of coding period ).

## Contact info and timezone(s)


- ***Contact***
  - **Email***: souravjangar91@gmail.com*
  - **Phone No**: (+91) 9958791453
  - **Github**: @sahiljoster32
- **Preferred mode of Communication:** Hangout(google chat), Discord, Email, and WhatsApp.
- **TimeZone**: Indian Standard Time (GMT+5:30)

## Time commitment

- I'm sure about my vacation period, which is from 5th June to 25h July. So, I can put in some extra hours in this vacation period.
- I'm decreasing the number of hours for the 10th week, because of my mid-term exams.

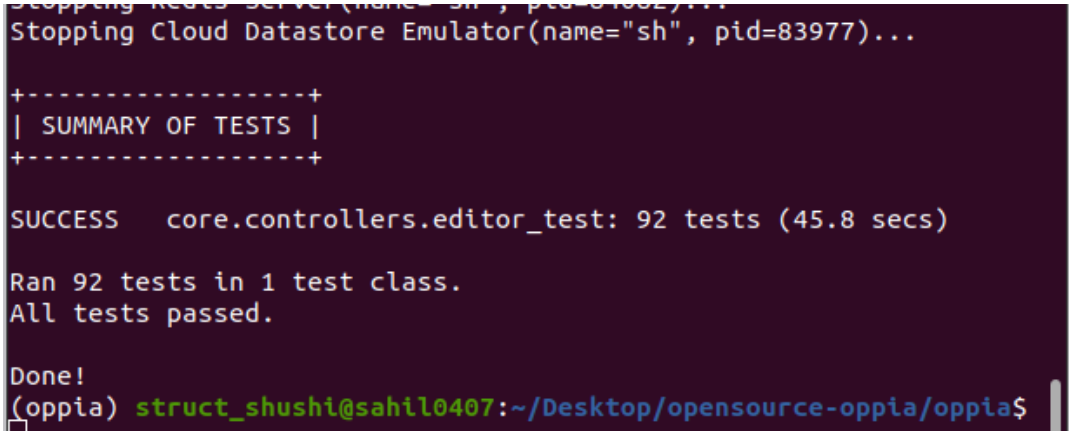
| S. No | Dates | Days(Total) | Time Commitment |
|---|---|---|---|
| 1. | 13th June - 19th June | Mon - Sat (6) | 4-5hr/day - 28h/week |
| 2. | 20th June - 26th June | Mon - Sat (6) | 4-5hr/day - 28h/week |
| 3. | 27th June - 3rd July | Mon - Sun (7) | 4-5hr/day - 28h/week |
| 4. | 4th July - 10th July | Mon - Sat (6) | 4-5hr/day - 28h/week |
| 5. | 11th July - 17th July | Mon - Sat (6) | 4-5hr/day - 28h/week |
| 6. | 18th July - 24th July | Mon - Sat (6) | 3-4hr/day - 20h/week |
| 7. | 25th July - 31st July | Tue - Sat (5) | 2-3hr/day - 15h/week * |
| 8. | 1st Aug - 7th Aug | Mon - Sat (5) | 3-4hr/day - 20h/week |
| 9. | 8th Aug - 14th Aug | Mon - Sat (5) | 4-5hr/day - 20h/week |
| 10. | 15th Aug - 21st Aug | Mon - Sat (6) | 2-3hr/day - 15h/week * |
| 11. | 22nd Aug - 28th Aug | Mon - Sat (6) | 3-4hr/day - 20h/week |
| 12. | 29th Aug - 3rd Sept | Mon - Fri (5) | 3-4hr/day - 20h/week |
| 13. | 4th Sept - 12th Sept | Mon - Sat (6) | 3-4hr/day - 20h/week |
| 14. | 13th Sept - 19th Sept | Mon - Fri (5) | 3-4hr/day - 20h/week |
| 15. | 20th Sept - 26th Sept | Mon - Sat (6) | 3-4hr/day - 20h/week |
| 16. | 27th Sept - 3rd Oct | Mon - Sat (6) | 3-4hr/day - 20h/week |

***Estimated Total Time Commitment**: 350 hours (This can be subject to change according to progress and need).*

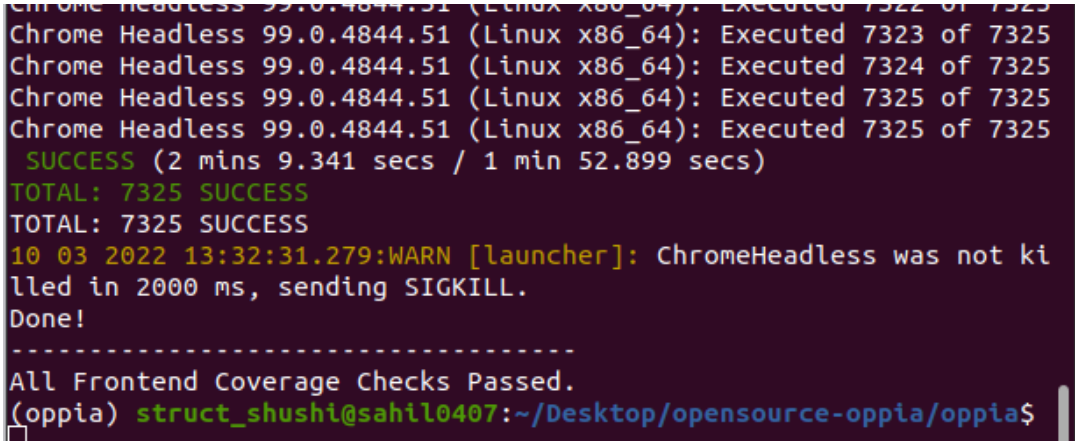
(The above-mentioned information regarding the time commitment is best in my knowledge, at the time of writing this proposal.)

## Essential Prerequisites

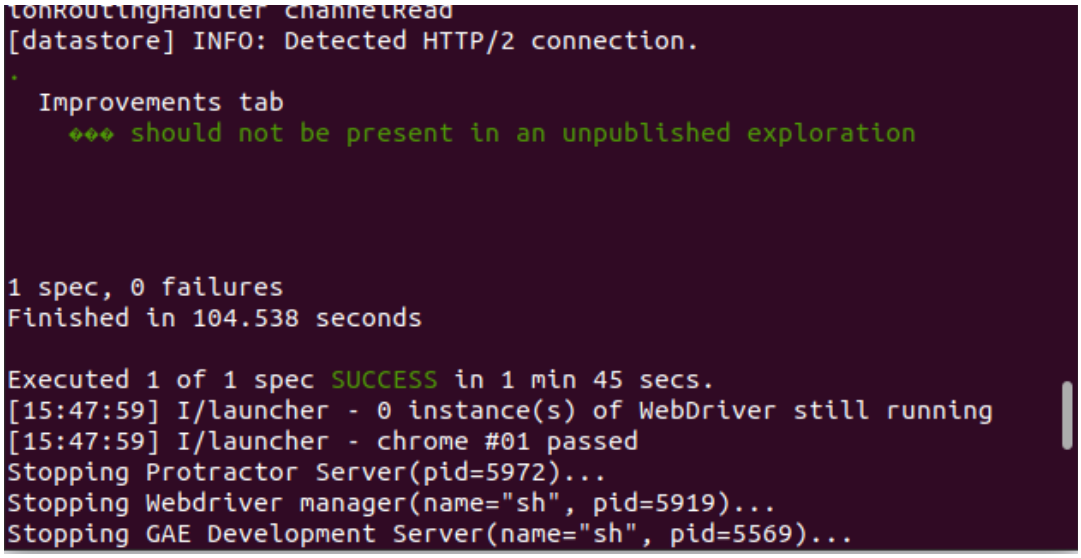
- *I am able to run a single backend test target on my machine.*




- *I am able to run all the frontend tests at once on my machine.*




- *I am able to run one suite of e2e tests on my machine.*



### Other summer obligations

The Schedule of my end-semester exams is currently tentative, but probably exams are going to be held in late May month. So, I do not have any summer obligations. If there is a change in schedule I will mention it in advance.

### Communication channels

I am planning to update my mentors twice a week by google meets, also I'm always available through google chat (hangouts). I am also comfortable with any other channel that the mentors prefers.

# Section 2: Proposal Details

# Problem Statement

| | |
|---|---|
| **Link to PRD (or N/A if there isn't one)** | **Make backend code typed** |
| **Target Audience** | The targeted audience of this feature is mainly **developers** of oppia organization. |
| **Core User Need** | The developers of oppia codebase currently follows the Docstring's type annotations for python files. The docstring type annotations are not sufficient for checking the type errors. Changing the type of parameters/functions/methods in docstring without correcting them in other places can lead to untrackable errors and also put other developers in ambiguity.<br><br>So, as a **contributor** to oppia, I want a seamless workflow without caring about type errors. Also, it would be better if errors could be caught earlier in the development before pushing it to the main repository, in that way we have more robust code against type errors. To achieve these tasks we need to fully type the backend codebase. |
| **What goals do we want the solution to achieve?** | • Introduce backend typing in all python files in oppia codebase.<br>• Remove all the docstring typeinfo from python files.<br>• Update docstring lint checks to support the new docstring style.<br>• Updating the wiki page for common errors that occurred in the codebase. |

# Section 2.1: WHAT

## Key User Stories and Tasks

| # | Title | User Story Description (role, goal, motivation) | Priority | List of tasks needed to achieve the goal (this is the "User Journey") | Links to mocks/prototypes, and/or PRD sections that spec out additional requirements. |
|---|---|---|---|---|---|
| 1 | developer workflow | As a contributor I want to be sure about the types of parameters and return value of a function that I'm going to use. So that, proper workflow is maintained and types of incoming values from other functions is predicted easily. | Must have | Introduce type annotations to all python files. | N/A. Because most of the changes is to be done in the existing codebase. |
| | | | | Document all exceptional cases while introducing MyPy type annotations to python files. | |
| | | | | Enable schema validation for all the handler classes in "core/controllers" folder. | |
| 2 | Duplicate parameter typeinfo. | As a contributor, I don't want to see duplicate typeinfo in docstring. There might be cases where docstring and function signature have different typeinfos. | Must have | Remove the typeinfos from existing docstring. | Current docstring style with type info<br><br>Mock docstring style without typeinfo. |
| | | | | Update the python docstring lint checks, to check the existence of typeinfo in docstring. | |
| 3 | Wiki about common errors. | As a new contributor, I would prefer to read docs for solutions to some frequent errors rather than searching for similar cases in the codebase or asking a mentor. | Should have | Add information about common errors on the existing wiki page. | N/A. Because the existing wiki page is going to be updated. |

## Technical Requirements

No additional requirements are needed for this project.

## Folders that are going to be covered under backend typing:

- core/domain
- core/jobs
- core/controllers
- extensions/
- scripts/
- Other selected files which are mentioned in **NOT_FULLY_COVERED_FILES** deny list of MyPy.

## Other accomplishments that are covered under this project:

- Covering schema validation for all files that come under the `core/controllers` folder.
- Updating docstring lint checkers for python files.
    - Files that are going to be updated.

```
└── scripts/
    ├── docstrings_checker.py
    ├── docstrings_checker_test.py
    └── linters/
        ├── pylint_extensions.py
        └── pylint_extensions_test.py
```

- Updating the existing Wiki for common MyPy errors encountered in the codebase.

---

# Section 2.2: HOW

## Existing Status Quo

Python is a dynamically typed language and oppia uses it in mostly the backend part of the codebase. Previous year oppia introduced backend typing in the codebase to convert dynamically typed code to static typed code and this task is supported by the MyPy type checker ( a static type checker in python ). In the previous year, core/storage, core/platform, and root folder files were annotated. But currently more than half of the codebase is still pending with the static type annotations.

There are a total of **467 files** that are still pending with static type annotations.

All these files are mentioned in the **NOT_FULLY_COVERED_FILES** list **(** also known as the mypy deny list ) and this list is defined in `scripts/run_mypy_checks.py`.

Apart from the backend type annotations, There are **67 handler classes** present in the files of 'core/controllers' that are still pending from schema validation being enabled.

Also, the oppia codebase currently using an additional custom pylint checker to check the python file's docstring. The name of the checker is **DocstringParameterChecker** and it is defined as a class in **pylint_extensions.py.** The **DocstringParameterChecker** usually checks docstrings for styling, missing sections, missing definitions, and their respective typeinfos. So, this class contains the **visit_functiondef()** method which is mainly responsible for checking the docstring of functions/methods by calling other appropriate methods of the class mentioned above**.**

Following methods are called by **visit_functiondef()**:-

- check_functiondef_params()
- check_functiondef_returns()*
- check_functiondef_yields()*
- check_docstring_style()*
- check_docstring_section_indentation()
- check_typeinfo()

*(methods marked with * are not responsible for checking typeinfo in the docstrings, so we are not going to alter these checks.)*

This custom pylint checker (DocstringParameterChecker) uses **scripts/docstrings_checker.py** as a utility module because, this module defines a class like **GoogleDocstring** (class for checking whether docstrings follow the custom Google Python Style Guide or not.) and functions like which help in fetching arguments name from functions, converting docstrings into list of strings and etc.

Currently following are the formats used by oppia to document the parameters definitions in the docstrings:

- **For the "Args" section -**
  'variable_name: typeinfo. Description.' format is used to document parameters in docstrings.

- **For the "Returns/Yields" section -**
  'Typeinfo. Description.' format is used to define return/yield documentation in docstrings.

- **For the "Raises" section -**
  'Exception_name. Description.' format is used to document raised exceptions in docstrings.

**Problem with current docstrings -**
Once all the python files are annotated, this typeinfo in the docstrings are considered duplicate information. So to avoid this we have to remove all the typeinfos from existing docstrings.

Also, oppia's current **DocstringParameterChecker** does not check docstrings for the missing "Args:" section. Whereas, missing "Returns:" and missing "Raises:" sections are checked by **visit_return()** and **visit_raise()** methods respectively.

## Solution Overview

I have divided the project's solution into two phases:

1. Updating the existing python docstring lint checker.
    1.1. How existing docstring lint checks are updated?
2. Introducing MyPy static type annotations to all Python files.
    2.1. How are static type annotations introduced to the chosen files?
    2.2. Order in which folders are going to be annotated.

## Task 1: Updating the existing python docstring lint checker.

### 1.1 How existing docstring lint checks are updated?

This solution is proposed for the following docstring pattern:

```
def create_profile_user_auth_details(
    user_id: str, parent_user_id: str
) -> auth_domain.UserAuthDetails:
    """Returns a domain object for a new profile user.

    Args:
        user_id: A user ID produced by Oppia for the new profile user.
        parent_user_id: The user ID of the full user account which will own
            the new profile account.

    Returns:
        Auth details for the new user.

    Raises:
        ValueError. The new user's parent is itself.
    """
    if user_id == parent_user_id:
        raise ValueError('user cannot be its own parent')
    return auth_domain.UserAuthDetails(user_id, None, None, parent_user_id)
```

The above docstring follows the following format to document parameter definitions in docstring:

- **For the “Args” section -**
  ‘variable_name: Description.’ format is used to document parameters in docstrings.

- **For the “Returns/Yields” section -**
  ‘Description.’ format is used to define return/yield documentation in docstrings.

- **For the “Raises” section -**
  ‘Exception_name. Description.’ format is used to document raised exceptions in docstrings.

To achieve the new docstring pattern. We need to perform the following tasks:

- Update the existing methods/functions of the lint checker to follow the new docstring style.
- Add a check for the missing “Args:” section.
- Add a lint check to forbid `# type pragmas` if a proper comment is not present.
- Add a lint check to forbid exceptional types(Any, cast and object) in the backend codebase if a proper comment is not present.

**Update the existing methods/functions of the lint checker to follow the new docstring style:**

**check_typeinfo():**

This method checks whether all parameters in a function definition are documented in the proper format or not. It splits docstring into different sections like Args, Returns, Raises, and Yields. Then it checks for appropriate formatting of parameters by matching against a Reg-ex expression.

**Args: section -**

Currently, we are following the **‘variable_name: typeinfo. Description.’** formatting to document a parameter in the “Args:” section. To remove the typeinfo from the existing format we have to replace “**: typeinfo.”** with a new delimiter.

The delimiter that I’m going to use is ‘**:**’ ( colon ). So, the new format to define parameters is **‘variable_name: Description.’**

To achieve this format, I’m replacing the current reg-ex pattern in **check_typeinfo()** with:

```
re_param_line = re.compile
    r"""
        \s*  \*{0,2}(\w+)             # identifier potentially with asterisks
        \s*  ([:])                    # separator for identifier and description
        \s*  [A-Z0-9](.*)[.\]}\)]+$  # beginning of optional description
    """, flags=re.X | re.S | re.M)
```

**Returns: and Yields: section -**

Currently, we are following the **'typeinfo. Description**.' format to define return/yield docstring. To remove typeinfo, the format is going to be replaced with only **'Description**.'

To achieve this format, I'm replacing the current reg-ex pattern in **check_typeinfo()** with:

```
re_returns_line = re.compile(
        r"""
            \s* [A-Z0-9](.*)[.\]}\)]+$         # beginning of description
        """, flags=re.X | re.S | re.M)
```

*(Note: return and yield both uses same reg-ex pattern.)*

**Raises: section -**

The raises section is fine with its current state and we are not changing it in this project because:

- MyPy only checks typing information but it does not include the type of exceptions raised by a function. Example of typed function raising valid exceptions.

```
    def validate(self) -> None:
        """Checks that the user_id, email, pin and display_alias
        fields of this UserSettings domain object are valid.

        Raises:
            ValidationError. The user_id is not str.
            ValidationError. The email is not str.
            ValidationError. The pin is not str.
            ValidationError. The display alias is not str.
        """
        if not isinstance(self.user_id, str):
            raise utils.ValidationError(
```

- If we remove exception_name from here we have no other place in the function definition to cover this exception_name.

**check_functiondef_params():**

This method checks whether all parameters in a function definition are documented or not. So, this task is achieved by calling the **check_arguments_in_docstring()** method. The check_functiondef_params() also makes sure that __init__ or class docstrings can have no parameters documented as long as the other documents them and this task is accomplished by **check_single_constructor_params()** helper function.

As stated above, this method uses 2 helper functions:

**check_single_constructor_params() -**
This method only checks the existence of “Args:” section in __init__ or class docstring. But this method has nothing to do with typeinfo so it doesn’t require any alterations.

**check_arguments_in_docstring() -**
In this method, a set of parameters name defined in function definition (*expected_argument_names*) is compared with the set of parameters name documented in Args: section (*params_with_doc*).

Here, **expected_argument_names** is fetched from argument_node of a function/method.

```
        expected_argument_names = set(
            arg.name for arg in arguments_node.args)
        expected_argument_names.update(
            arg.name for arg in arguments_node.kwonlyargs)
```

Whereas **params_with_doc** is fetched using the inbuilt function **( match_param_docs() )** but as we are not following the current format that we have already defined in the the class **GoogleDocstring** under the file docstrings_checker.py**.** This inbuilt function is going to return an empty set and due to an empty set, this whole check is going to be skipped.

To avoid the above scenario we have to define a new method **( match_new_param_docs() )**, which returns a set of parameters name documented in “Args:” section according to the new docstring style.

This method is going to be defined under the class **GoogleDocstring,** present in **scripts/docstrings_checker.py**.

Code implementation of above the method is as follows:

```
re_new_param_line = re.compile(
    r"""
        \s*  \*{0,2}(\w+)             # identifier potentially with asterisks
```

```
        \s*  ([:])                    # delimiter for definition
        \s*  ([A-Z0-9](.*)[.\]}\)]+$)     # beginning of optional description
    """, flags=re.X | re.S | re.M)

def match_new_param_docs(self):
    """Returns the set parameter names which are properly documented

     Returns:
         set(str). A set of parameter names which are properly defined
         in docstring.
     """
     params_with_doc = set()

     entries = self._parse_section(self.re_param_section)
     entries.extend(self._parse_section(self.re_keyword_param_section))
     for entry in entries:
         match = self.re_new_param_line.match(entry)
         if not match:
            continue

         param_name = match.group(1)
         param_desc = match.group(3)

         if param_desc:
             params_with_doc.add(param_name)

     return params_with_doc
```

In updated docstring lint checks, this **match_new_param_docs()** is going to be called in place of the **match_param_docs()** method (old docstring style function).

★ Also, the functionality of checking parameters which are documented but typeinfo is not mentioned will also be removed.

**check_docstring_section_indentation():**

This method checks whether the function argument definitions ("Args": section, "Returns": section, "Yield": section, "Raises: section) are indented properly. Also, Parameters/Return documentation should be indented by 4 relative to the heading and any wrap-around descriptions should be indented by 8. If the description line ends with a colon, then we can assume the rest of the section is free form. Example of free_form_section, in the free form section we're ignoring all indentation.

**Returns: and Yields: section -**

This method checks indentation section-wise. For getting into the section it checks for the heading like “Returns:”, “Args:” and etc.

```
elif stripped_line.startswith('Returns:'):
```

After getting into one of the sections, it checks for the head of the parameter definition. The head of the parameter definition is used to identify the start of new parameter/return documentation and all other indentations (description indentation and free_form_indetation) are checked relative to this head.

In the current scenario, the head of the parameter definition is **‘typeinfo.’** which is a subpart of **‘typeinfo. Description.’**

Once head of the parameter definition is matched, then this method checks for description indentation until no new head of the parameter definition is encountered or there is no colon(:) at the end of the line.

So reg-ex which oppia is using for head of the parameter definition in Returns/Yields section is:

```
if (re.search(r'^[a-zA-Z_() -:,\*]+\.',
                      stripped_line) and not in_description):
```

To follow the new docstring style, head of the parameter definition should be changed to ‘**Description’.** This means the first letter is always a capital letter or a number followed by any character.

```
if (re.search(r'^[A-Z0-9](.*)',
                      stripped_line) and not in_description):
```

**Args: -**

In the args section, the head of the parameter definition is **‘variable_name:’** which is a subpart of **‘variable_name: typeinfo. Description.’**. But after changing the formatting style to **‘variable_name: Description.’,** the head of the parameter definition still remains the same. Because **‘variable_name:’** is independent of type info.

So, in this project “Args:” section’s head of the parameter definition is not going to be altered.

**Add a check for the missing “Args:” section.**

The aim of this check is to throw an error, whenever we define a function with parameters (not including self & cls) but its corresponding “Args:” section is not documented in docstring. For example:

```
    def __init__(self, linter=None):
        super(RestrictedImportChecker, self).__init__(linter=linter)
        self._module_to_forbidden_imports = []
```

So, I'm going to build this check under the name **check_functiondef_args().** This check is going to be called by **visit_functiondef()**.

Why **visit_functiondef()** is used to call **check_functiondef_args()**?
Because **visit_functiondef()** is called for every function/method internally and it invokes all the functions listed in its function body.

The implementation of **check_functiondef_args()** is:

```
def check_functiondef_args(self, node, node_doc):
    """Checks whether a function having arguments "except self and cls"
    also documents Args: section in its docstring.

    Args:
        node: astroid.scoped_nodes.Function. Node for a function or
            method definition in the AST.
        node_doc: Docstring. Pylint Docstring class instance representing
            a node's docstring.
    """

    args_in_function_node = set(
        arg.name for arg in node.args.args)

    expected_argument_names = (
        args_in_function_node - self.not_needed_param_in_docstring)

    if (not node_doc.has_params()) and (expected_argument_names):
        self.add_message(
                    'missing-arg-doc', node=node)
```

Firstly, all the parameters from the function node (**args_in_function_node**) are fetched. After that self and cls (defined in **self.not_needed_param_in_docstring**) are removed.

By adding errors to the **self.add_message()** an error message is displayed in the console if docstring does not contain Args: section (**node_doc.has_params()**) but it has parameters in its function definition apart from self and cls.

**Add a lint check to forbid `# type pragmas` if a proper comment is not present:**

The aim of this check is throw an error whenever a `type pragma` is used without an explanatory comment.

```
class TypeIgnoreCommentChecker(checkers.BaseChecker):
    """Custom pylint checker which checks if MyPy's type ignores are
    properly documented or not.
    """

    __implements__ = interfaces.ITokenChecker

    name = 'type_ignore_documentation'
    priority = -1
    msgs = {
        'C0045': (
            'MyPy type ignore is used. Add a proper comment if
             \'type: ignore\' is needed.',
            'mypy-ignore-used',
            'MyPy ignores should be used with proper comments. Except’
            ‘for no-untyped-call MyPy ignore.'
        )
    }

    def process_tokens(self, tokens):
        """Custom pylint checker which allows only those MyPy type ignores
        that are properly documented.

        Args:
            tokens: Token. Object to access all tokens of a module.
        """
        type_ignore_comment_regex = r'^# Here we used MyPy ignore because'
        type_ignore_comment_present = False

        for (token_type, _, (line_num, _), _, line) in tokens:
            if token_type == tokenize.COMMENT:
                line = line.lstrip()

                if re.search(type_ignore_comment_regex, line):
                    type_ignore_comment_present = True

                if re.search(r'(#\s*type:)', line):
                    if '# type: ignore[no-untyped-call]' in line:
```

```
                    continue
                if type_ignore_comment_present:
                    type_ignore_comment_present = False
                else:
                    self.add_message(
                        'mypy-ignore-used', line=line_num)
```

Here, firstly we are checking that a comment starting with `Here we used MyPy ignore because` is present or not. After that we are checking for the following cases:

1. If `# type:` is present after the comment then its an ok case and we can continue with our check.
2. If `# type:` is present but it's corresponding comment is not present then this check will throw an error `MyPy type ignore is used. Add a proper comment if 'type: ignore' is needed.`

**Add a lint check to forbid exceptional types(Any, cast and object) in the backend codebase if a proper comment is not present:**

```
class ExceptionalTypesCommentChecker(checkers.BaseChecker):
    """Custom pylint checker which checks that there is always a comment
    for exceptional types in backend type annotations.
    """

    __implements__ = interfaces.ITokenChecker

    name = 'comment-for-exceptional-types'
    priority = -1
    msgs = {
        'C0047': (
            'Any type is used. Please add a proper comment if'
            ' Any type is needed',
            'any-type-used',
            'Annotations with Any type should be done with'
            ' a proper comments.'
        ),
        'C0048': (
            'cast function is used. Please add a proper comment if'
            ' cast is needed',
            'cast-func-used',
            'Casting of any value should be done with a proper comment.'
        ),
```

```
        'C0049': (
            'object class is used. Please add a proper comment if'
            ' object is needed',
            'object-class-used',
            'Annotating any value with object should be done with a proper'
            ' comment.'
        )
    }

    def process_tokens(self, tokens):
        """Custom pylint checker which makes sure that every exceptional type
        should be documented properly.
        Args:
            tokens: Token. Object to access all tokens of a module.
        """
        any_type_regex = r'^# Here we used type Any because'
        cast_type_regex = r'^# Here we used cast because'
        object_type_regex = r'# Here we used object because'

        # Variables to keep count of exceptional types in the same line.
        any_already_encountered_line_num = 0
        object_already_encountered_line_num = 0

        # Variables to keep track of comments for exceptional types.
        any_type_comment_present = False
        cast_comment_present = False
        object_comment_present = False

        for (token_type, token, (line_num, _), _, line) in tokens:
            line = line.strip()

            # Checking if comment for exceptional type is encountered or not.
            if token_type == tokenize.COMMENT:
                if re.search(any_type_regex, line):
                    any_type_comment_present = True
                if re.search(cast_type_regex, line):
                    cast_comment_present = True
                if re.search(object_type_regex, line):
                    object_comment_present = True

            if token_type == tokenize.NAME:

                if token == 'import':
```

```
                    import_token_line_num = line_num

                if token == 'Any':
                    # Excluding the case when Any is present in an import.
                    # Eg: from typing import Any.
                    if re.search(r'typing', line):
                        continue

                    # Excluding the case when Any is present with too many
                    # other types in an import.
                    # Eg: from typing import (
                    #   Any, Callable, Dict, FrozenSet, Iterator, List,
                    #   Set, Tuple, Type, cast)
                    if line_num in (
                        import_token_line_num + 1,
                        import_token_line_num + 2
                    ):
                        continue

                    # Excluding the case when two or more Any types are present
                    # in a single line.
                    # Eg: Dict[Any, Any]
                    if any_already_encountered_line_num == line_num:
                        continue
                    any_already_encountered_line_num = line_num

                    # Throwing an error when Any is encountered but there is no
                    # corresponding comments exist.
                    if any_type_comment_present:
                        any_type_comment_present = False
                    else:
                        self.add_message(
                            'any-type-used', line=line_num)

                if token == 'cast':
                    # Excluding the case when cast is present in an import.
                    # Eg: from typing import cast.
                    if re.search(r'typing', line):
                        continue

                    # Excluding the case when cast is present with too many
                    # other types in an import.
                    # Eg: from typing import (
```

```
                    #   Any, Callable, Dict, FrozenSet, Iterator, List, Set,
                    #   Tuple, Type, cast)
                    if line_num in (
                        import_token_line_num + 1,
                        import_token_line_num + 2
                    ):
                        continue

                    # Throwing an error when cast is encountered but there is no
                    # corresponding comment exist.
                    if cast_comment_present:
                        cast_comment_present = False
                    else:
                        self.add_message(
                            'cast-func-used', line=line_num)

                if token == 'object':
                    if object_already_encountered_line_num == line_num:
                        continue
                    object_already_encountered_line_num = line_num
                    # Excluding the case when object is called:
                    # Eg: var = object()
                    if 'object()' in line:
                        continue
                    if object_comment_present:
                        object_comment_present = False
                    else:
                        self.add_message(
                            'object-class-used', line=line_num)
```

This linter checks for the following behaviors:

1. If a comment starting with `Here we used type Any because` is present before `Any` type then It's an ok case and if the comment is not present then the linter throws an error.

```
# Here we used type Any because function int can take any type of value
# and convert it to the integer value.
def int(value: Any) -> int: ...
```

2. If a comment starting with `Here we used cast because` is present before `cast` method then It's an ok case and if the comment is not present then the linter throws an error.

```
# Here we used cast because the return value of getattr() method is
# dynamic and mypy will assume it to be Any otherwise.
```

```
return cast(Callable[..., str], getattr(cls, normalizer_id))
```

3. Linter check the similar behavior for object, If a comment starting with `Here we used object because` is present before `object` class then It's an ok case and if the comment is not present then the linter throws an error.

***( Mock implementation of new python docstring lint checks can be found [here.](#) )***

## Task 2: Introducing MyPy static type annotations to all Python files.

### 2.1 How are static type annotations introduced to the chosen files?

When converting the current oppia's codebase files from dynamic typing to static typing, I encountered some of the following recurring cases along with their solutions. I also documented few codebase-specific cases.

**Case 1:**
Sometimes functions have different types of return values based on different types of input arguments.

```
def test_func(x: Union[str, int]) -> Union[str, int]:
    if isinstance(x, str):
        return 'hi'
    if isinstance(x, int):
        return 1
```

Looking at the function signature in this case we can see that the function returns Union of string and int on every string and int argument x. To make this definition even more clear we can add ***overload*** decorator as follows:

```
@overload
def test_func(x: str) -> str: ...

@overload
def test_func(x: int) -> int: ...

def test_func(x: Union[str, int]) -> Union[str, int]:
    if x is isinstance(x, str):
```

**Case 2:**

When we inherit a class and change it's one of the method's signature in child class.

```
class test1:

    def mock_func(x: str) -> str:
        return x

class test2(test1):

    def mock_func(y: int) -> int:
        return y
```

Taking the above implementation, MyPy throws an error with the statement "error: Signature of "mock_func" incompatible with supertype "test1"  [override]". To avoid this we should refactor the code, if possible. Otherwise we have to add an  ignore statement "# type: ignore[override]" with an explanatory comment.

**Case 3:**

When we are assigning values of different types to the variables of different types. MyPy throws an assignment error with the statement "error: Incompatible types in assignment (expression has type "*{{expression's type}}*", variable has type "*{{variable's type}}*")  [assignment]".

```
string_var: str = other_type_var
```

In this case we have 2 scenarios:

- If **other_type_var** is of Optional[str] type (means it can contain both **str** value and **None**) and we are sure that at this assignment **other_type_var** is containing a string value. Then we can use "assert other_type_var is not None" with an explanatory code comment to rule out the possibility of **None** for the MyPy type checker.

  ```
  # Ruling out the possibility of None for mypy type checking.
  assert other_type_var is not None
  string_var: str = other_type_var
  ```

- If **other_type_var** is of Union[str, int] type (means it can contain both **str** type and **int** type values). And at this point of the assignment, we know **other_type_var** is containing a str type value. Then we can use "assert isinstance(other_type_var, str)" with an explanatory code comment to rule out the possibility of int of  **other_type_var** for the MyPy type checker.

```
# Ruling out the possibility of int of other_type_var for mypy type
# checking.
assert isinstance(other_type_var, str)
string_var: str = other_type_var
```

**Case 4:**

When an untyped decorator is applied to a particular typed function. Then that typed function is also considered as an untyped function by the MyPy type checker.

Please take a look at the following code snippet taken from the codebase:

```
    # Using type ignore[misc] here because untyped decorator makes function
    # "get" also untyped.
    @acl_decorators.open_access # type: ignore[misc]
    def get(self, username: str) -> None:
        """Validates access to profile page."""

        user_settings = user_services.get_user_settings_from_username( #
type: ignore[no-untyped-call]
            username)

        if not user_settings:
            raise self.PageNotFoundException
```

( Note: decorator given in this example is not annotated )

The ideal way to tackle this error is that we have to annotate the decorator first then only we should annotate the function. If the decorator belongs to any third-party library, then we have to add its typing in stubs.

**Case 5:**

When we import modules and services indirectly in a python file and MyPy is not able to recognize those modules and services. In that case, MyPy throws an error if we try to access that module or service, with the error statement “error: Module has no attribute "ActivityReferencesModel"  [attr-defined]”.

```
(activity_models,) = models.Registry.import_models([models.NAMES.activity])
```

To handle this error during MyPy type checking we have to add modules/services as follows:

```
MYPY = False
if MYPY: # pragma: no cover
    from mypy_imports import activity_models

(activity_models,) = models.Registry.import_models([models.NAMES.activity])
```

( Above example is taken from the codebase. )

By doing this we are allowing MyPy to import module directly, but this module is imported only for type checking. At runtime this import is ignored.

**Case 6:**

**Use cases of cast keyword.**

There are some cases in the codebase where we know the return type of function but mypy is not able to fetch that return type. Because of some situations like function is not annotated yet or function belongs to some third-party library whose stubs are not available in the typeshed yet. Then In that case, mypy assumes the return value to be **Any** type.

To narrow down the return type from **Any** of a function that belongs to any third-party library and not type annotated yet we have to add stubs for it in the stubs folder, so that MyPy can refer to that stub's function signature while checking the type annotations. While if a function is present in the codebase and not type annotated yet then there we can cast the value to suppress the error until the function is annotated, an example of casting the value is mentioned below :

Please take a look at the code snippet taken from codebase:

```
SERIALIZATION_FUNCTIONS: SerializationFunctionsDict = {
    CACHE_NAMESPACE_COLLECTION: lambda x: x.serialize(),
    CACHE_NAMESPACE_EXPLORATION: lambda x: cast(str, x.serialize()), # type:
ignore[no-untyped-call]
    CACHE_NAMESPACE_SKILL: lambda x: cast(str, x.serialize()), # type:
ignore[no-untyped-call]
    CACHE_NAMESPACE_STORY: lambda x: cast(str, x.serialize()), # type:
ignore[no-untyped-call]
    CACHE_NAMESPACE_TOPIC: lambda x: x.serialize(),
    CACHE_NAMESPACE_PLATFORM_PARAMETER: lambda x: cast(str, x.serialize()), # type:
ignore[no-untyped-call]
    CACHE_NAMESPACE_CONFIG: json.dumps,
    CACHE_NAMESPACE_DEFAULT: json.dumps
}
```

Taking the above implementation, CACHE_NAMESPACE_SKILL's function is not type annotated yet so mypy assumes Its return value to be Any type. But we know the return type so

we casted it to **str** type. Whereas, other functions are annotated so we don't need **cast** for them.

Other cases where **cast** can be used:

- When the return value of a function is too dynamic, then MyPy assumes its return value to be Any type. But we can use cast if we are sure about the function's return value based on its usage.
  Please take a look at the code snippet taken from the codebase:

```
        # Using a cast here because the return value of getattr() method is
        # dynamic and mypy will assume it to be Any otherwise.
        return cast(Callable[..., str], getattr(cls, normalizer_id))
```

**Case 7:**

When we are returning a dictionary that has different types of values from a function, then in that case we can use Dict[str, Any] type.

```
def dict_mock() -> Dict[str, Any]:

    return {
        'name': 'XYZ',
        'ID_number': 1,
        'IDs': ['1','2', '3'],
        'created_on': datetime.utcnow()
    }
```

But the above annotations are not precise if we look carefully. So, to fully annotate this we have to define a TypedDict class and assign it as a return annotation.

```
class MockDict(TypedDict):

    name: str
    ID_number: int
    IDs: List[str]
    created_on: datetime.datetime

def dict_mock() -> MockDict:
...
```

**Case 8:**

Mypy also performs numerous other, less commonly failing checks that don't have specific error codes. These checks use the [misc] error code. Some of the commonly encountered [misc] errors in the oppia codebase are:

- "error: Class cannot subclass 'PTransform' (has type 'Any')  [misc]". The suspected cause for this error is that the typing of 'PTransform' class is still not available for the MyPy type checker.
  **Solution:**
  we need to write stubs only for the part of the library we are using, and place those stubs inside the stubs/ folder.

- Common errors related to the keys of dictionary:
  - "error: Key 'id' of TypedDict "CollectionDict" cannot be deleted  [misc]". This error mostly occurs when we try to delete a key from a well defined TypedDict type.
  - "error: TypedDict "CollectionDict" has no key 'next_skill_id'  [misc]". The reason for this error is that we try to access the key which is not defined in the TypedDict class of the dictionary.

  **Solution:**
  we should try to refactor some code so that we can minimize the use of accessing undefined keys and deletion of keys.

  However, in some cases it is not feasible to refactor the code. So there we can use "# type: ignore statements wth an explanatory comment.

- "error: 'classmethod' used with a non-method  [misc]". This error mostly occurs when we try to define a **classmethod** inside a method of the class.

  Please take a look at the code snippet taken from codebase:

```
class RegistryTests(test_utils.TestBase):

    def test_get_all_jobs_returns_value_from_job_metaclass(self) -> None:
        unique_obj = object()

        @classmethod # type: ignore[misc]
        def get_all_jobs_mock(
            unused_cls: Type[base_jobs.JobMetaclass]
        ) -> object:
            """Returns the unique_obj."""
            return unique_obj
```

**Solution:**
This case mostly occurs in test files, so there we can use `# type pragmas` to silent the error but the preferred solution is to refactor the code and an example of this can be found HERE.

**Case 9:**
When we define a TypedDict class to provide annotations for a dictionary, and we are accessing the dictionary's key using a constant that has a different name than that defined in TypedDict class. But the value of the constant is correct according to the TypedDict class.

Please see the code snippet taken from the codebase:

```
CACHE_NAMESPACE_STORY: Final = 'story'

CACHE_NAMESPACE_TOPIC: Final = 'topic'


class DeserializationFunctionsDict(TypedDict):
    """Type for the DESERIALIZATION_FUNCTIONS."""

    story: Callable[[str], story_domain.Story]
    topic: Callable[[str], topic_domain.Topic]


DESERIALIZATION_FUNCTIONS: DeserializationFunctionsDict = {
    CACHE_NAMESPACE_STORY: story_domain.Story.deserialize,
    CACHE_NAMESPACE_TOPIC: topic_domain.Topic.deserialize,
}
```

Then in that case MyPy throws `"error: TypedDict "DeserializationFunctionsDict" has no key CACHE_NAMESPACE_STORY [misc]"`. The suspected cause for this error is that MyPy is not able to recognize useful constants unless their type is declared explicitly.

So, the solution for this error is that we have to annotate the constants also. But we can do that in 2 ways:

- By using Literal keyword from typing.
- By using Final keyword from typing. ( as shown in the above example )

I prefer to use the latter one because by using Final we don't have to mention the value of constant. Whereas in Literal we have to mention the constant's values.

```
CACHE_NAMESPACE_TOPIC: Literal['topic'] = 'topic'
```

**Case 10:**

For external libraries, MyPy obtains the type information from the type stubs defined in the typeshed package. But currently, there are some libraries that are not supported by the typeshed yet. So, to overcome this issue we need to define the stubs ourselves only for the part of the library we are using, and place those stubs inside the "stubs/" folder.

There might be a situation where we don't have enough information to write stubs, Then in that case we can create a TODO issue to add stubs later when information is available and add that TODO issue in the codebase wherever is applicable.

**Practices that I'm going to follow during introducing type annotations:**

- Annotations will be introduced in the codebase, by keeping in mind that strict types will be on high priority and try to use Any type as minimum as possible.
- For any kind # ignore: other than [no-untyped-call], an explanatory comment will be added with proper reasoning.
- Some # ignore: can be fixed when the whole codebase is annotated. So, for them, I'll create a TODO issue and mention it in the file wherever applicable.
- While adding annotations I will follow oppia's coding style by following the coding_style_doc wiki page.

**What if a new case is encountered in the codebase?**

Due to huge community support for python typing, there are high chances that error is already caught by the community and documented with reasoning and its solution.

However, to tackle the new cases I'm going to follow some documentation and wikis for this project but are not limited to:

- PEP 484 type hints.
- Error_code_lists of the MyPy type checker.
- Wiki of oppia for backend type annotations.
- Type_hint cheat sheet of python3.

If the error is not resolved by following any of the above resources then I'll let my mentors know about the error and try to find a generalized solution for this new error.

**What if a proper typeinfo is not available for function, method, or class?**

Oppia's codebase has 100% test coverage. This will help us a lot in adding type annotations. If for some reason tests are not sufficient, then I'll try to infer the type from the usage of function/method/class in the whole oppia's codebase.

However, in some cases we can use **reveal_type()** function (does not exist on runtime) to fetch the return type of function and the types of arguments of function.

As mentioned earlier, mentors would be the last option if typeinfo is not available in any way.

### 2.2 Order in which folders are going to be annotated.

To introduce the backend type annotations in the whole codebase, I will annotate all the files/folders which are mentioned in the **NOT_FULLY_COVERED_FILES** list.

This list is defined in `scripts/run_mypy_checks.py`.

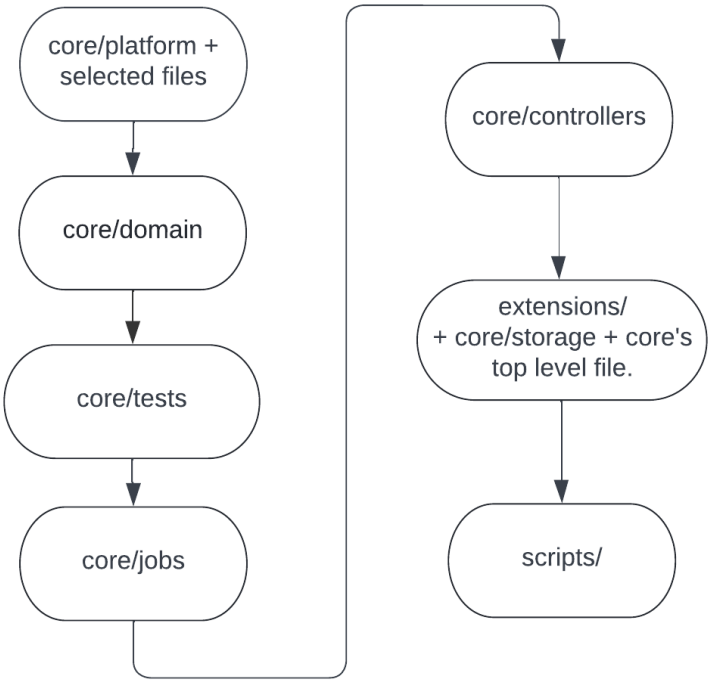
The list of folders that I'm going to cover:

| Folders | No. of Files |
|---|---|
| core/controllers<br>core/domain<br>core/platform ( some new files )<br>core/storage ( some new files )<br>core/jobs<br>extensions/<br>scripts/<br>core/tests<br>core's root folder files. | 108<br>136<br>02<br>05<br>74<br>43<br>88<br>07<br>04 |
| Total files ( test files are included ) | 467 |

**Dependency order:**

After inspecting all the folders where untyped files are present, I think that it would be efficient to annotate those folders first where most of the dependencies are present. So that we can minimize the use of `# ignore:` statements while introducing the backend type annotations and annotate the folders according to the flow of code.

The following flow diagram clearly shows the order of folders in which they are going to be annotated.




**The reasoning and current status of each folder are as follows:**

**core/platform + some selected files:**

These files are frequently used in the domain, controllers, and other folders too. So to follow the dependency order I propose to annotate these files first than any other folder.

**Order of covering files:**

These files will be annotated in the following manner as mentioned in the table.

| No. | File | Dependencies yet to be annotated. |
|---|---|---|
| 1. | core/platform/storage/cloud_storage_emulator.py | **N/A** |
| 2. | core/python_utils.py | **N/A** |
| 3. | storage/blog/gae_models.py | **N/A** |
| 4. | storage/beam_job/gae_models.py | **N/A** |

**core/domain folder:**

The dependency of the domain folder is very less on the files outside of the domain folder and the few files which are imported from outside of the domain folder are somewhat already annotated. The only file which is most frequently imported in the domain's test files and not annotated yet is **`test_utils.py`.**

Also, folders like core/test and core/controllers import too many modules from core/domain. So to maintain the dependency order I chose core/domain over core/controllers and core/test to annotate first.

**Order of covering files within the core/domain folder is:**

Files that have fewer dependencies which are yet to be annotated are going to be covered first. For example:

- Files like **auth_services.py** and **change_domain.py** are going to be covered before **blog_services.py** and **user_services.py.** ( please refer to the following table )

On further inspection of the files in **domain** folder together with the issue tracking the files of this folder. I found the current state of the files during writing of this proposal is as follows:

**Status of core/domain folder:**

| No. | File | Dependencies yet to be annotated. |
|---|---|---|
| 1. | auth_services.py | **N/A** |
| 2. | change_domain.py **(No test file)** | **N/A** |
| 3. | classroom_services.py | **N/A** |
| 4. | cron_services.py **(No test file)** | **N/A** |
| 5. | customization_args_util.py | **N/A** |
| 6. | expression_parser.py | **N/A** |
| 7, | fs_domain.py | **N/A** |
| 8. | improvements_services.py | **N/A** |
| 9. | interaction_registry.py | **N/A** |
| 10. | learner_goals_services.py | **N/A** |
| 11. | playthrough_issue_registry.py | **N/A** |
| 12. | recommendations_services.py | **N/A** |
| 13. | role_services.py | **N/A** |
| 14. | subscription_services.py | **N/A** |

| | | |
|---|---|---|
| 15. | taskqueue_services.py | **N/A** |
| 16. | fs_services.py | core.domain.fs_domain |
| 17. | html_cleaner.py | core.domain.rte_component_registry |
| 18. | image_validation_services.py | core.domain.html_validation_service |
| 19. | learner_playlist_services.py | core.domain.subscription_services |
| 20. | moderator_services.py | core.domain.taskqueue_services |
| 21. | param_domain.py | core.domain.object_registry |
| 22. | platform_parameter_domain.py | core.domain.change_domain |
| 23. | platform_parameter_registry.py | core.domain.platform_parameter_domain |
| 24. | rte_component_registry.py | core.python_utils |
| 25. | rules_registry.py | core.python_utils |
| 26. | search_services.py | core.domain.rights_manager |
| 27. | skill_fetchers.py | core.domain.skill_domain |
| 28. | story_fetchers.py | core.domain.story_domain |
| 29. | subtopic_page_services.py | core.domain.subtopic_page_domain |
| 30. | takeout_service.py | core.domain.user_services |
| 31. | user_query_services.py | core.domain.email_manager |
| 32. | visualization_registry.py | extensions.visualizations.models |
| 33. | blog_services.py | core.domain.html_cleaner<br>core.domain.role_services |
| 34. | classifier_services.py | core.domain.exp_fetchers<br>core.domain.fs_services |
| 35. | exp_fetchers.py | core.domain.exp_domain<br>core.domain.subscription_services |
| 36. | object_registry.py | core.python_utils<br>extensions.objects.models.objects |
| 37. | platform_parameter_list.py | core.domain.platform_parameter_domain<br>core.domain.platform_parameter_registry |
| 38. | question_fetchers.py | core.domain.question_domain<br>core.domain.state_domain |
| 39. | subtopic_page_domain.py | core.domain.change_domain |

| | | |
|---|---|---|
| | | core.domain.state_domain |
| 40. | topic_fetchers.py | core.domain.classroom_services<br>core.domain.story_fetchers |
| 41. | user_services.py | core.domain.auth_services<br>core.domain.role_services |
| 42. | email_manager.py | core.domain.html_cleaner<br>core.domain.subscription_services<br>core.domain.user_services |
| 43. | platform_feature_services.py | core.platform_feature_list<br>core.domain.platform_parameter_domain<br>core.domain.platform_parameter_registry |
| 44. | rating_services.py | core.domain.event_services<br>core.domain.exp_fetchers<br>core.domain.exp_services |
| 45. | stats_Service.py | core.domain.exp_fetchers<br>core.domain.question_services<br>core.domain.stats_domain |
| 46. | draft_upgrade_services.py | core.domain.exp_domain<br>core.domain.html_validation_services<br>core.domain.rules_registry<br>core.domain.state_domain |
| 47. | rights_manager.py | core.domain.role_services<br>core.domain.subscription_services<br>core.domain.taskqueue_services<br>core.domain.user_services |
| 48. | skill_domain.py | core.domain.change_domain<br>core.domain.state_domain<br>core.domain.html_cleaner<br>core.domain.html_validation_service |
| 49. | stats_domain.py | core.domain.customization_args_util<br>core.domain.exp_domain<br>core.domain.interaction_registry<br>core.domain.playthrough_issue_registry |
| 50. | exp_domain.py | core.domain.change_domain<br>core.domain.param_domain<br>core.domain.state_domain<br>core.domain.html_cleaner<br>core.domain.html_validation_service |
| 51. | feedback_services.py | core.domain.email_manager<br>core.domain.rights_manager<br>core.domain.subscription_services<br>core.domain.taskqueue_services<br>core.domain.user_services |

| | | |
|---|---|---|
| 52. | html_validation_service.py | core.domain.fs_domain<br>core.domain.fs_services<br>core.domain.rte_component_registry<br>extensions.objects.models.objects<br>extensions.rich_text_components.components |
| 53. | opportunity_services.py | core.domain.exp_fetchers<br>core.domain.question_fetchers<br>core.domain.story_fetchers<br>core.domain.suggestion_services<br>core.domain.topic_fetchers |
| 54. | question_services.py | core.domain.opportunity_services<br>core.domain.question_domain<br>core.domain.question_fetchers<br>core.domain.skill_fetchers<br>core.domain.state_domain |
| 55. | story_domain.py | core.domain.change_domain<br>core.domain.fs_domain<br>core.domain.fs_services<br>core.domain.html_cleaner<br>core.domain.html_validation_service |
| 56. | collection_services.py | core.domain.exp_fetchers<br>core.domain.exp_services<br>core.domain.rights_manager<br>core.domain.search_services<br>core.domain.subscription_services<br>core.domain.user_services |
| 57. | event_services.py | core.domain.exp_domain<br>core.domain.exp_fetchers<br>core.domain.feedback_services<br>core.domain.stats_domain<br>core.domain.stats_services<br>core.domain.taskqueue_services |
| 58. | state_domain.py | core.domain.customization_args_util<br>core.domain.param_domain<br>extensions.objects.models.objects<br>core.domain.html_cleaner<br>core.domain.interaction_registry<br>core.domain.rules_registry |
| 59. | voiceover_services.py | core.domain.email_manager<br>core.domain.exp_fetchers<br>core.domain.opportunity_services<br>core.domain.rights_manager<br>core.domain.suggestion_registry<br>core.domain.user_services |
| 60. | story_services.py | core.domain.exp_fetchers<br>core.domain.exp_services<br>core.domain.opportunity_services |

| | | |
|---|---|---|
| | | core.domain.rights_manager<br>core.domain.story_domain<br>core.domain.story_fetchers<br>core.domain.suggestion_services<br>core.domain.topic_fetchers |
| 61. | suggestion_services.py | core.domain.email_manager<br>core.domain.exp_fetchers<br>core.domain.feedback_services<br>core.domain.html_cleaner<br>core.domain.html_validation_service<br>core.domain.question_domain<br>core.domain.suggestion_registry<br>core.domain.user_services |
| 62. | summary_services.py | core.domain.collection_services<br>core.domain.exp_domain<br>core.domain.exp_fetchers<br>core.domain.exp_services<br>core.domain.rights_manager<br>core.domain.search_services<br>core.domain.stats_services<br>core.domain.user_services |
| 63. | question_domain.py | core.domain.html_cleaner<br>core.domain.html_validation_service<br>core.domain.interaction_registry<br>core.domain.change_domain<br>core.domain.customization_args_util<br>core.domain.exp_domain<br>core.domain.expression_parser<br>core.domain.state_domain<br>extensions.domain |
| 64. | wipeout_service.py | core.domain.auth_services<br>core.domain.collection_services<br>core.domain.email_manager<br>core.domain.exp_fetchers<br>core.domain.exp_services<br>core.domain.rights_manager<br>core.domain.taskqueue_services<br>core.domain.topic_services<br>core.domain.user_services |
| 65. | learner_progress_services.py | core.domain.classroom_services<br>core.domain.collection_services<br>core.domain.exp_fetchers<br>core.domain.learner_goals_services<br>core.domain.learner_playlist_services<br>core.domain.skill_services<br>core.domain.story_fetchers<br>core.domain.story_services<br>core.domain.subscription_services<br>core.domain.topic_fetchers<br>core.domain.topic_services |

| | | |
|---|---|---|
| 66. | skill_services.py | core.domain.html_cleaner<br>core.domain.opportunity_services<br>core.domain.role_services<br>core.domain.skill_domain<br>core.domain.skill_fetchers<br>core.domain.state_domain<br>core.domain.suggestion_services<br>core.domain.taskqueue_services<br>core.domain.topic_fetchers<br>core.domain.topic_services<br>core.domain.user_services |
| 67. | suggestion_registry.py | core.domain.exp_domain<br>core.domain.exp_fetchers<br>core.domain.exp_services<br>core.domain.fs_services<br>core.domain.html_cleaner<br>core.domain.question_domain<br>core.domain.question_services<br>core.domain.skill_domain<br>core.domain.skill_fetchers<br>core.domain.state_domain<br>core.domain.user_services |
| 68. | topic_services.py | core.domain.feedback_services<br>core.domain.opportunity_services<br>core.domain.role_services<br>core.domain.state_domain<br>core.domain.story_fetchers<br>core.domain.story_services<br>core.domain.subtopic_page_domain<br>core.domain.subtopic_page_services<br>core.domain.suggestion_services<br>core.domain.topic_fetchers<br>core.domain.user_services |
| 69. | exp_services.py | core.domain.classifier_services<br>core.domain.draft_upgrade_services<br>core.domain.email_manager<br>core.domain.exp_domain<br>core.domain.exp_fetchers<br>core.domain.feedback_services<br>core.domain.fs_domain<br>core.domain.html_cleaner<br>core.domain.html_validation_service<br>core.domain.opportunity_services<br>core.domain.param_domain<br>core.domain.recommendations_services<br>core.domain.rights_manager<br>core.domain.search_services<br>core.domain.state_domain<br>core.domain.stats_services<br>core.domain.taskqueue_services<br>core.domain.user_services |

*( This table contains files and their dependencies that are yet to be annotated. )*

**core/tests folder:**

Out of all the files in the core/tests folder, **test_utils.py** is the most important one. Because this file is imported in almost every backend test of oppia's codebase and this file is annotated after the domain folder because it imports 23 domain modules.

Once **test_utils.py** is annotated, then we can get rid of `# type: ignore[no-untyped-call]` statements from all the backend tests of oppia codebase.

**Order of covering files within this folder is:**

- **test_utils.py** is going to be annotated first.
- All other files are annotated in dependency order.

**Status of core/tests folder:**

| No. | File | Dependencies yet to be annotated. |
|---|---|---|
| 1. | test_utils.py | core.controllers.base<br>*20+ domain files* |
| 2. | gae_suite.py | **N/A** |
| 3. | load_tests/feedback_thread_summaries_test.py* | core.domain.feedback_services<br>core.tests.test_utils |
| 4. | build_sources/extensions/base.py | core.python_utils<br>core.domain.object_registry<br>core.domain.visualization_registry<br>extensions.domain<br>extensions.objects.models.objects |

*( Files marked with * does not have test file. )*

**core/jobs folder:**

**Order of covering sub-folders within this folder is:**

- Order of covering folders are shown in the following table.
    - Folder **jobs/types** is going to be covered before folder **jobs/decorators.**

- Files in **jobs/batch_jobs** and **jobs/transforms/validation** are going to be covered in alphabetical order. Because files in these folders are mostly independent of each other.

**Status of core/jobs folder:**

| No. | Files/Folder | Total untyped files. |
|---|---|---|
| 1. | jobs/types | 16 |
| 2. | jobs/decorators | 02 |

| | | |
|---|---|---|
| 3. | jobs/transforms/* ( root-level files in transforms folder ) | 02 |
| 4. | jobs/transforms/validation | 30 |
| 5. | jobs/batch_jobs | 24 |

**core/controllers folder:**

The files of core/controllers are highly dependent on the files of core/domain folder and some files are dependent on the files of core/jobs folder. So that's why i propose to cover this folder after core/domain and core/jobs.

Also, There are **67 handler classes** present in the files of core/controllers that are still pending from schema validation being enabled. So I propose to enable schema validation for these handler classes in this project as well. To enable schema validation, I'm going to follow the instructions as mentioned in the issue (Write schemas for handler class arguments).

**Order of covering files in core/controllers folder is:**

- **acl_decorators.py, base.py** and **domain_objects_validator.py** are going to be annotated first because these 3 files are imported into every other file of controllers.
- All other files will be annotated in alphabetical order because, at this point all dependencies from core/controllers/* ( core/domain and core/jobs ) will already be annotated.

**Status of core/controllers folder:**

| No. | File | Dependencies yet to be annotated. |
|---|---|---|
| 1. | acl_decorators.py | core.controllers.base<br>10+ domain files. |
| 2. | base.py | core.controllers.payload_validator<br>core.domain.auth_services<br>core.domain.user_services |
| 3. | domain_objects_validator.py | core.controllers.base<br>core.domain.exp_domain<br>core.domain.image_validation_services<br>core.domain.question_domain<br>core.domain.state_domain |
| 4. | android_e2e_config.py | core.controllers.acl_decorators<br>core.controllers.base<br>15+ domain files. |
| 5. | admin.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.controllers.domain_objects_validator |

| | | |
|---|---|---|
| | | 20+ domain files. |
| 6. | blog_admin.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.controllers.domain_objects_validator<br>core.domain.blog_services<br>core.domain.role_services<br>core.domain.user_services |
| 7. | blog_dashboard.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.controllers.domain_objects_validator<br>core.domain.blog_services<br>core.domain.fs_services<br>core.domain.image_validation_services<br>core.domain.user_services |
| 8. | blog_homepage.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.blog_services<br>core.domain.user_services |
| 9. | classifier.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.classifier_services<br>core.domain.email_manager<br>core.domain.exp_fetchers |
| 10. | classroom.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.classroom_services<br>core.domain.topic_fetchers |
| 11. | collection_editor.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.collection_services<br>core.domain.rights_manager<br>core.domain.search_services<br>core.domain.summary_services |
| 12. | collection_viewer.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.rights_manager<br>core.domain.summary_services |
| 13. | concept_card_viewer.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.skill_fetchers |
| 14. | contributor_dashboard_admin.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.email_manager<br>core.domain.suggestion_services<br>core.domain.topic_fetchers<br>core.domain.user_services |

| | | |
|---|---|---|
| 15. | contributor_dashboard.py | core.controllers.acl_decorators<br>core.controllers.base<br>6 domain files |
| 16. | creator_dashboard.py | core.controllers.acl_decorators<br>core.controllers.base<br>10+ domain files. |
| 17. | cron.py | core.controllers.acl_decorators<br>core.controllers.base<br>5 domain files<br>4 jobs files |
| 18. | custom_landing_pages.py | core.controllers.acl_decorators<br>core.controllers.base |
| 19. | editor.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.controllers.domain_objects_validator<br>14+ domain files |
| 20. | email_dashboard.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.controllers.domain_objects_validator<br>core.domain.email_manager<br>core.domain.user_query_services<br>core.domain.user_services |
| 21. | features.py | core.controllers.acl_decorators<br>core.controllers.base |
| 22. | feedback.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.feedback_services<br>core.domain.suggestion_services<br>core.domain.user_services |
| 23. | improvements.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.controllers.domain_objects_validator<br>core.domain.exp_fetchers<br>core.domain.improvements_services |
| 24. | incoming_app_feedback_report.py | core.controllers.acl_decorators<br>core.controllers.base |
| 25. | learner_dashboard.py | core.controllers.acl_decorators<br>core.controllers.base<br>7 domain files |
| 26. | learner_goals.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.learner_goals_services<br>core.domain.learner_progress_services |

| | | |
|---|---|---|
| 27. | learner_playlist.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.learner_playlist_services<br>core.domain.learner_progress_services |
| 28. | library.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.collection_services<br>core.domain.exp_services<br>core.domain.summary_services<br>core.domain.user_services |
| 29. | moderator.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.email_manager<br>core.domain.summary_services |
| 30. | oppia_root.py | core.controllers.acl_decorators<br>core.controllers.base |
| 31. | pages.py | core.controllers.acl_decorators<br>core.controllers.base |
| 32. | payload_validator.py | **N/A** |
| 33. | platform_feature.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.platform_feature_services |
| 34. | practice_sessions.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.skill_fetchers<br>core.domain.topic_fetchers |
| 35. | profile.py | core.controllers.acl_decorators<br>core.controllers.base<br>7 domain files |
| 36. | question_editor.py | core.controllers.acl_decorators<br>core.controllers.base<br>7 domain files. |
| 37. | questions_list.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.question_services<br>core.domain.skill_domain<br>core.domain.skill_fetchers |
| 38. | reader.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.controllers.domain_objects_validator<br>core.controllers.editor<br>15+ domain files. |
| 39. | recent_commits.py | core.controllers.acl_decorators |

| | | |
|---|---|---|
| | | core.controllers.base<br>core.domain.exp_services<br>core.domain.user_services |
| 40. | release_coordinator.py | core.controllers.acl_decorators<br>core.controllers.base |
| 41. | resources.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.fs_domain |
| 42. | review_tests.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.skill_fetchers<br>core.domain.story_fetchers |
| 43. | skill_editor.py | core.controllers.acl_decorators<br>core.controllers.base<br>6 domain files |
| 44. | skill_mastery.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.skill_domain<br>core.domain.skill_fetchers<br>core.domain.skill_services<br>core.domain.topic_fetchers |
| 45. | story_editor.py | core.controllers.acl_decorators<br>core.controllers.base<br>7 domain files. |
| 46. | story_viewer.py | core.controllers.acl_decorators<br>core.controllers.base<br>8 domain files. |
| 47. | subscriptions.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.subscription_services<br>core.domain.user_services |
| 48. | subtopic_viewer.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.subtopic_page_services<br>core.domain.topic_fetchers |
| 49. | suggestion.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.controllers.domain_objects_validator<br>9 domain files. |
| 50. | tasks.py | core.controllers.acl_decorators<br>core.controllers.base<br>10 domain files. |
| 51. | topic_editor.py | core.controllers.acl_decorators |

| | | |
|---|---|---|
| | | core.controllers.base<br>15 domain files. |
| 52. | topic_viewer.py | core.controllers.acl_decorators<br>core.controllers.base<br>core.domain.email_manager<br>core.domain.skill_services<br>core.domain.story_fetchers<br>core.domain.topic_fetchers |
| 53. | topics_and_skills_dashboard.py | core.controllers.acl_decorators<br>core.controllers.base<br>10 domain files. |
| 54. | voice_artist.py | core.controllers.acl_decorators<br>core.controllers.base<br>5 domain files |

*( This table contains files and their dependencies that are yet to be annotated. )*

**extensions + core/storage + pending core's root level files:**

**Order of covering files in these folders are:**

- For extensions folder, files and sub-folders are going to be covered in directory-wise order.
    - Root-level files are annotated first.
    - Then all folders are annotated in alphabetical order. Because these folders are mostly independent of each other.

- For core/storage files and core's root files, order will be in alphabetical order.

**Status of extensions folder**

| No. | Files/Folder | Dependencies yet to be annotated. |
|---|---|---|
| 1. | domain.py | **N/A** |
| 2. | actions/* (5) | extensions.domain |
| 3. | answer_summarizers/models.py | core.domain.exp_domain<br>core.domain.stats_domain |
| 4. | interactions/* (22) | extensions.interactions.base (majorly) |
| 5. | issues/* (5) | extensions.domain ( only for base.py )<br>extensions.interactions.base |
| 6. | objects/* (2) | **N/A** |
| 7. | rich_text_components/* (2) | core.python_utils |

| | | |
|---|---|---|
| 8. | value_generators/* (2) | core.domain.value_generators_domain |
| 9. | extensions/visualizations/models.py | core.domain.calculation_registry |

*( No of files in a folder specified as folder_name/* (no of files) )*

**Status of core/storage + core's root level files.**

| No. | File | Dependencies yet to be annotated. |
|---|---|---|
| 1. | core/platform_feature_list.py | core.domain.platform_parameter_list |
| 2. | storage/storage_models_test.py | core.domain.takeout_service<br>core.tests.test_utils |

**scripts/ folder:**
Status of the current scripts/ folder can be found in the issue tracking the files of this folder.

**Order of covering files in scripts/ folder are:**

- **common.py** and **servers.py** are the scripts/ folder's root level files, so these are going to be annotated first. Because these 2 are imported in almost every file of scripts/ folder.
- All other files/folders are annotated in alphabetical order.
- Files in sub-folder **scripts/linters** and **scripts/release_scripts** will also be annotated in alphabetical order.

***( The order in which static type annotations will be introduced to all the python files is listed down in Milestones. )***

## Impact on Other Oppia Teams

After updating all the python files with the new docstring style, we need to inform all the contributors about the new docstring styles. So that whenever they push a new python file they don't get lint errors and are already aware of the new style.

To inform we can take the help of oppia-dev group, by sending an email a week before completing all the python files.

**Developer workflow team:**

There might be a possibility that we face a clash between our PRs, Consider a case where I'm working on an ABC.py file to introduce MyPy type annotations and someone also working on an ABC.py file to cover backend test coverage.

Then, in that case, we have to mutually agree on a solution. So that everything goes as planned.

## Risks and mitigations

- While introducing backend type annotations, there might be chances that the backend test fails and impact the coverage. In that case, I'm going to cover that backend test and coverage as well.
- The intention of this project is to cover the whole codebase. So files that are assigned to other contributors are going to be covered as well, Iff there is no PR already opened for the assigned files.

## Implementation Approach

### Documentation changes

➔ **Backend Type Annotations documentation:**
All the common cases which are encountered in the oppia codebase plus all the new cases which I'm going to encounter are going to be well-documented on the wiki page. A draft of the wiki page can be found here.

➔ **Coding_style documentation:**
Once all the python files are updated with new docstring style, then the docstrings section of this documentation will also be updated.

## Testing Plan

In this project, we are introducing new python docstrings lint checks and there will be tests for this lint checks also.

```
class DocstringParameterCheckerTests(unittest.TestCase):

    def setUp(self):
        super(DocstringParameterCheckerTests, self).setUp()
        self.checker_test_object = testutils.CheckerTestCase()
        self.checker_test_object.CHECKER_CLASS = (
            pylint_extensions.DocstringParameterChecker)
        self.checker_test_object.setup_method()

    def test_well_formated_returns_section_old_docstring_style(self):
        node_with_no_error_message = astroid.extract_node(
            u"""def func(): #@
                \"\"\"Does nothing.

                Returns:
                    int. Argument escription.
```

```
            \"\"\"
            return args
        """)

        with self.checker_test_object.assertAddsMessages():
            self.checker_test_object.checker.visit_functiondef(
                node_with_no_error_message)

    def test_correct_args_formatting_in_new_docstring_style(self):
        incorrect_args_format_in_new_style = astroid.extract_node(
            """
        def func(test_var_one, test_var_two): #@
            \"\"\"Function to test docstring parameters.

            Args:
                test_var_one: int. First test variable.
                test_var_two: str. Second test variable.
            \"\"\"
            result = test_var_one + test_var_two
        """)
        malformed_args_section = testutils.Message(
            msg_id='malformed-args-section',
            node=incorrect_args_format_in_new_style)
        with self.checker_test_object.assertAddsMessages(
            malformed_args_section,
            malformed_args_section
        ):
            self.checker_test_object.checker.visit_functiondef(
                incorrect_args_format_in_new_style)

        correct_args_format_in_new_style = astroid.extract_node(
            """
        def func(test_var_one, test_var_two): #@
            \"\"\"Function to test docstring parameters.

            Args:
                test_var_one: First test variable.
                test_var_two: Second test variable.
            \"\"\"
            result = test_var_one + test_var_two
        """)
        with self.checker_test_object.assertNoMessages():
            self.checker_test_object.checker.visit_functiondef(
                correct_args_format_in_new_style)

    def test_correct_returns_formatting_in_new_docstring_style(self):
        invalid_return_documentation = astroid.extract_node(
```

```
            """
        def func(test_var_one, test_var_two): #@
            \"\"\"Function to test docstring parameters.

            Args:
                test_var_one: First test variable.
                test_var_two: Second test variable.

            Returns:
                int. The test result.
            \"\"\"
            result = test_var_one + test_var_two
            return result
        """)

        malformed_returns_section = testutils.Message(
            msg_id='malformed-returns-section',
            node=invalid_return_documentation)
        with self.checker_test_object.assertAddsMessages(
            malformed_returns_section
        ):
            self.checker_test_object.checker.visit_functiondef(
                invalid_return_documentation)

        valid_return_documentation = astroid.extract_node(
            """
        def func(test_var_one, test_var_two): #@
            \"\"\"Function to test docstring parameters.

            Args:
                test_var_one: First test variable.
                test_var_two: Second test variable.

            Returns:
                The test result.
            \"\"\"
            result = test_var_one + test_var_two
            return result
        """)
        with self.checker_test_object.assertNoMessages():
            self.checker_test_object.checker.visit_functiondef(
                valid_return_documentation)
```

( Note: these tests are just to elaborate the approach, this is not the full test suite. )

**MyPy static type annotations checks:**

Every python file mentioned in this project is tested with the MyPy type checker and I'll make sure that every python file passes all MyPy checks.

**Backend test and backend coverage test:**

Oppia already has 100 percent backend test coverage. However, in case while introducing backend type annotations any test fails then I'll also fix those test cases and I'll make sure that every test passes without any failing coverage checks.

## Feature testing

Does this feature include non-trivial user-facing changes? **NO**

# Implementation Plan:

## Milestone 1:

**Key Objective:**
Fully type core/domain, core/tests, and core/jobs. Put measures in place to ensure that the files in these folders have full backend typing in perpetuity, to Oppia's standards (i.e. not using "Any", casts, and objects, and only using a narrow, fully-documented subset of type-ignore pragmas).

| PR No. | Description of PR / action | Prereq PR numbers | Target date for PR creation | Target date for PR to be merged |
|---|---|---|---|---|
| 1.1 | Adding MyPy static type annotations to<br>● core/platform/storage/cloud_storage_emulator.py<br>● core/python_utils.py<br>● storage/blog/gae_models.py<br>● storage/beam_job/gae_models.py | | 13th June | 17th June |
| 1.2 | Adding MyPy static type annotations to<br>1. auth_services.py<br>2. change_domain.py **(No test file)**<br>3. classroom_services.py<br>4. Cron_services.py **(No test file)**<br>5. customization_args_util.py<br>6. expression_parser.py<br>7. fs_domain.py | | 16th June | 22nd June |

| | | | | |
|---|---|---|---|---|
| | 8. improvements_services.py<br>9. interaction_registry.py<br>10. learner_goals_services.py<br>11. playthrough_issue_registry.py<br>12. recommendations_services.py<br>13. role_services.py<br>14. subscription_services.py<br>15. taskqueue_services.py | | | |
| 1.3 | Adding MyPy static type annotations to<br>1. fs_services.py<br>2. html_cleaner.py<br>3. image_validation_services.py<br>4. learner_playlist_services.py<br>5. moderator_services.py<br>6. param_domain.py<br>7. platform_parameter_domain.py<br>8. platform_parameter_registry.py<br>9. rte_component_registry.py<br>10. rules_registry.py | 1.1 | 19th June | 25th June |
| 1.4 | Adding MyPy static type annotations to<br>1. search_services.py<br>2. skill_fetchers.py<br>3. story_fetchers.py<br>4. subtopic_page_services.py<br>5. takeout_service.py<br>6. user_query_services.py<br>7. visualization_registry.py<br>8. blog_services.py | | 22nd June | 28th June |
| 1.5 | Adding MyPy static type annotations to<br>● classifier_services.py<br>● exp_fetchers.py<br>● object_registry.py<br>● platform_parameter_list.py<br>● question_fetchers.py<br>● subtopic_page_domain.py | 1.1 | 25th June | 1st July |
| 1.6 | Adding MyPy static type annotations to<br>● topic_fetchers.py<br>● user_services.py<br>● email_manager.py<br>● platform_feature_services.py<br>● rating_services.py<br>● stats_Service.py | | 28th June | 4th July |
| 1.7 | Adding MyPy static type annotations to<br>● draft_upgrade_services.py<br>● rights_manager.py<br>● skill_domain.py<br>● stats_domain.py<br>● exp_domain.py | | 1st July | 7th July |

| | | | | |
|---|---|---|---|---|
| | ● feedback_services.py | | | |
| 1.8 | Adding MyPy static type annotations to<br>● html_validation_service.py<br>● opportunity_services.py<br>● question_services.py<br>● story_domain.py<br>● collection_services.py<br>● event_services.py | | 4th July | 10th July |
| 1.9 | Adding MyPy static type annotations to<br>● state_domain.py<br>● voiceover_services.py<br>● story_services.py<br>● suggestion_services.py<br>● summary_services.py<br>● question_domain.py | | 7th July | 13th July |
| 1.10 | Adding MyPy static type annotations to<br>● wipeout_service.py<br>● learner_progress_services.py<br>● skill_services.py<br>● suggestion_registry.py<br>● topic_services.py<br>● exp_services.py | | 10th July | 16th July |
| 1.11 | Adding MyPy static type annotations to<br>● test_utils.py<br>● gae_suite.py<br>● tests/load_tests/feedback_thread_summaries_test.py<br>● tests/build_sources/extensions/base.py | 1.1,<br>1.6,<br>1.7,<br>1.8,<br>1.9,<br>1.10 | 11th July | 15th July |
| 1.12 | Adding MyPy static type annotations to<br>● jobs/types<br>● jobs/decorators<br>● jobs/transforms/* (root-level files) | 1.11 | 14th July | 18th July |
| 1.13 | Adding MyPy static type annotations to<br>● jobs/transforms/validation | 1.12 | 17th July | 21st July |
| 1.14 | Adding MyPy static type annotations to<br>● jobs/batch_jobs | 1.12,<br>1.13 | 20th July | 24th July |
| 1.15 | Adding lint checks that forbids '# type: ignore' and exceptional types(Any, cast and object) if no proper comment is present for them. | | 23rd July | 27th July |

## Milestone 2:

**Key Objective:**
Fully type the entire backend codebase, including schema validation for all handlers. Drop typeinfo from all docstrings and add new docstring lint checks to ensure that docstrings adhere to the new format in perpetuity. Also, ensure that measures are in place to prevent backend typing coverage from regressing in the codebase.

| No. | Description of PR / action | Prereq PR numbers | Target date for PR creation | Target date for PR to be merged |
|---|---|---|---|---|
| 2.1 | Adding MyPy static type annotations to<br>• acl_decorators.py<br>• base.py<br>• domain_objects_validator.py<br>• android_e2e_config.py<br>• admin.py<br>• blog_admin.py<br>• blog_dashboard.py<br>• blog_homepage.py | 1.11 | 30th July | 3rd August |
| 2.2 | Adding MyPy static type annotations to<br>• classifier.py<br>• classroom.py<br>• collection_editor.py<br>• collection_viewer.py<br>• concept_card_viewer.py<br>• contributor_dashboard_admin.py<br>• contributor_dashboard.py<br>• creator_dashboard.py | 1.11,<br>2.1 | 1st August | 7th August |
| 2.3 | Adding MyPy static type annotations to<br>• cron.py<br>• custom_landing_pages.py<br>• editor.py<br>• email_dashboard.py<br>• features.py<br>• feedback.py<br>• improvements.py<br>• incoming_app_feedback_report.py | 1.11,<br>2.1 | 5th August | 11th August |
| 2.4 | Adding MyPy static type annotations to<br>• learner_dashboard.py<br>• learner_goals.py<br>• learner_playlist.py<br>• library.py<br>• moderator.py<br>• oppia_root.py<br>• pages.py<br>• payload_validator.py | 1.11,<br>2.1 | 8th August | 14th August |
| 2.5 | Adding MyPy static type annotations to<br>• platform_feature.py | 1.11,<br>2.1 | 11th August | 17th August |

| | | | | |
|---|---|---|---|---|
| | • practice_sessions.py<br>• profile.py<br>• question_editor.py<br>• questions_list.py<br>• reader.py<br>• recent_commits.py<br>• release_coordinator.py | | | |
| 2.6 | Adding MyPy static type annotations to<br>• resources.py<br>• review_tests.py<br>• skill_editor.py<br>• skill_mastery.py<br>• story_editor.py<br>• story_viewer.py<br>• subscriptions.py<br>• subtopic_viewer.py | 1.11,<br>2.1 | 14th August | 20th August |
| 2.7 | Adding MyPy static type annotations to<br>• suggestion.py<br>• tasks.py<br>• topic_editor.py<br>• topic_viewer.py<br>• topics_and_skills_dashboard.py<br>• voice_artist.py | 1.11,<br>2.1 | 17th August | 23rd August |
| 2.8 | Adding MyPy static type annotations to<br>• extensions/domain.py<br>• extensions/actions/*<br>• extensions/answer_summarizers/models.py | 1.7 | 20th August | 26th August |
| 2.9 | Adding MyPy static type annotations to<br>• extensions/interactions/*<br>• extensions/issues/*<br>• extensions/objects/* | 2.8 | 23rd August | 29th August |
| 2.10 | Adding MyPy static type annotations to<br>• extensions/rich_text_components/*<br>• extensions/value_generators/* | 1.1 | 28th August | 3rd September |
| 2.11 | Adding MyPy static type annotations to<br>• extensions/visualizations/models.py<br>• core/platform_feature_list.py<br>• storage/storage_models_test.py | 1.4,<br>1.5,<br>1.11 | 31st August | 6th September |
| 2.12 | Adding MyPy static type annotations to<br>• scripts/common.py<br>• scripts/servers.py<br>• scripts/build.py<br>• scripts/check_e2e_tests_are_captured_in_ci.py<br>• scripts/check_frontend_test_coverage.py<br>• scripts/check_if_pr_is_low_risk.py | 1.11 | 3rd September | 9th September |
| 2.13 | Adding MyPy static type annotations to<br>• scripts/concurrent_task_utils.py<br>• scripts/docstrings_checker.py<br>• scripts/extend_index_yaml.py | 2.12,<br>1.11 | 6th September | 12th September |

| | | | | |
|---|---|---|---|---|
| | ● scripts/flake_checker.py<br>● scripts/install_backend_python_libs.py<br>● scripts/install_third_party_libs.py<br>● scripts/install_third_party.py | | | |
| 2.14 | Adding MyPy static type annotations to<br>● scripts/pre_commit_hook.py<br>● scripts/pre_push_hook.py<br>● scripts/regenerate_requirements.py<br>● scripts/rtl_css.py<br>● scripts/run_backend_tests.py<br>● scripts/run_custom_eslint_tests.py<br>● scripts/run_e2e_tests.py | 2.12,<br>1.11 | 9th September | 15th September |
| 2.15 | Adding MyPy static type annotations to<br>● scripts/run_frontend_tests.py<br>● scripts/run_lighthouse_tests.py<br>● scripts/run_mypy_checks.py<br>● scripts/run_portserver.py<br>● scripts/run_presubmit_checks.py<br>● scripts/setup.py<br>● scripts/typescript_checks.py | 2.12,<br>1.11 | 12th September | 18th September |
| 2.16 | Adding MyPy static type annotations to<br>● scripts/linters<br>● scripts/release_scripts | 2.12,<br>1.11 | 15th September | 21st September |
| 2.17 | Updating existing python docstring lint checkers:<br>● Updating custom docstring pylint checker.<br>● Removing type Infos from all docstrings of the codebase. | | 20th September | 25th September |
| 2.18 | ● Updating existing oppia's backend type annotations Doc.<br>● Updating documentation of oppia's coding_style doc. | 2.17 | 22nd September | 24th September |

## Future Work:

After completion of this project, we can introduce some more strict rules from mypy and remove ‘follow_imports’ flag (introduced previous year) to make the codebase even more strictly typed. Also, I will regularly maintain the stubs of third party libraries.

Once we changed all the docstrings, we can introduce some more lint checks to make the codebase even more robust against loose Code styling.

Apart from the project I will continue my contributions with the LaCE quality team and LaCE android team.

---

***Note:*** *I have taken references from Eesha Arif's proposal, Mridul Setia's proposal, Hardik Katehara's proposal and oppia’s backend_type_annotations doc while making this proposal.*