



**Make backend code typed**

**Hardik Katehara**

**Google Summer of Code**

# About You

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

Oppia's aim is to provide students with quality education irrespective of the resources they have access to. I want to help Oppia achieve its aim. I believe that everyone should have access to quality education.

Moreover, I was always fascinated by the idea of open source. It is very fascinating to see how people from different backgrounds come together and create things that make a major impact. I had a great experience contributing to Oppia. I have learned so much in just 2-3 months that would not have been possible without the help of the community.

I chose this project because I have the required skill set to complete this project. Moreover, I like the idea of making codebase typed as it makes it easy for new developers and makes it easy to debug programs.

## Prior experience

I have been contributing to Oppia for the past 2-3 months and have a decent idea of the codebase.

I have been working with Python for the past 1.5 years. I also have a decent understanding of Javascript. I am a backend developer. I have developed backends for 2 websites as a freelancer. I used Django for both of them and also helped with the Javascript part of the frontend.

My previous works include:

1. Backend of an E-commerce website like amazon in which different sellers and buyers come together. It also had Razorpay API integration.
2. Backend of a website on which a Diamond Showroom owner sells his diamonds online.

Links to PRs:

1. Fix part of #10049: Remove deprecated fields from ExplorationRightsModel ([link](#))
2. Fix part of #10049: Remove deprecated fields from ExplorationModel ([link](#))

3. Fix part of #10415: Refactor activity validators ([link](#))
4. Fix part of #11693: Adds validation for MAX\_COMMIT\_MESSAGE\_LENGTH ([link](#))
5. Fix part of #7450: Modify private methods in SuggestionIntegrationTests ([link](#))

You can find all my PRs [here](#).

## Contact info and timezone(s)

### Communication

1. **Email:** [hardikkatehara@gmail.com](mailto:hardikkatehara@gmail.com)
2. **GitHub:** [@hardikkat24](#)

I will be in India throughout this summer.

The time zone will be Indian Standard Time(GMT +5:30).

## Time commitment

I will be able to dedicate 6-7 hours/day for 5 days a week for 10 weeks. This can be increased if there is any need to do so.

## Essential Prerequisites

Answer the following questions:

- *I am able to run a single backend test target on my machine. (Show a screenshot of a successful test.)*

```
13:21:27 FINISHED core.controllers.base_test: 40.7 secs

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

SUCCESS core.controllers.base_test: 52 tests (19.9 secs)

Ran 52 tests in 1 test class.
All tests passed.

Done!
```

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

```

Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4348 of 4351 SUCCESS (0 secs / 1 min 3.222 secs)
LOG: 'Spec: Story Viewer Page component should get complete exploration url when clicking on svg elemChrome Headless 89.0.4
389.90 (Linux x86_64): Executed 4349 of 4351 SUCCESS (0 secs / 1 min 3.231 secs)LOG: 'Spec: Story Viewer Page component shou
ld not show story's chapters when story data is not loaded has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4349 of 4351 SUCCESS (0 secs / 1 min 3.231 secs)
LOG: 'Spec: Story Viewer Page component should not show story's chapters when story data is not loadedChrome Headless 89.0.4
389.90 (Linux x86_64): Executed 4350 of 4351 SUCCESS (0 secs / 1 min 3.239 secs)LOG: 'Spec: Story Viewer Page component shou
ld show warning when fetching story data fails has passed'
Chrome Headless 89.0.4389.90 (Linux x86_64): Executed 4350 of 4351 SUCCESS (0 secs / 1 min 3.239 secs)
LOG: 'Spec: Story Viewer Page component should show warning when fetching story data fails has passedChrome Headless 89.0.4
389.90 (Linux x86_64): Executed 4351 of 4351 SUCCESS (0 secs / 1 min 3.247 secs)Chrome Headless 89.0.4389.90 (Linux x86_64):
Executed 4351 of 4351 SUCCESS (1 min 21.326 secs / 1 min 3.247 secs)
TOTAL: 4351 SUCCESS
TOTAL: 4351 SUCCESS
04 04 2021 19:00:20.281:WARN [launcher]: ChromeHeadless was not killed in 2000 ms, sending SIGKILL.
Done!

```

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

```

Executed 4 of 4 specs SUCCESS in 10 mins 24 secs.
[01:58:01] I/launcher - 0 instance(s) of WebDriver still running
[01:58:01] I/launcher - chrome #01 passed
ERROR 2021-04-04 20:28:03,626 instance.py:284] Cannot connect to the instance on localhost:24875
i emulators: Received SIGTERM for the first time. Starting a clean shutdown.
i emulators: Please wait for a clean shutdown or send the SIGTERM signal again to stop right now.
i emulators: Shutting down emulators.
i ui: Stopping Emulator UI
⚠ Emulator UI has exited upon receiving signal: SIGTERM
i auth: Stopping Authentication Emulator
i hub: Stopping emulator hub
i logging: Stopping Logging Emulator

```

## Other summer obligations

I will be doing a project this summer which is compulsory in my university. It will basically be an internship for industrial experience. I will be required to dedicate only 10-12hours/weeks to this project.

## Communication channels

I will be in continuous touch with mentors through mail and hangouts (or as preferred by mentors). There could be biweekly meetings with the mentors through Google Meet (or as preferred by mentors).

---

# Project Details

## Product Design

Python is a dynamically typed language. It means that python types are checked at runtime.

The aims of this project are:

- Pre-push type checks
- CI type checks
- Add documentation about adding types
- assets/constants.ts converted to protobuf
- Implement types to some part of the codebase

The users of this feature are the **developers** on the Oppia team.

## Why type checking?

1. Help catch certain errors and increase runtime efficiency.
2. Help document code in form of type annotations.
3. Make code easier to understand and maintain.
4. Can help IDEs with code completion.

## What developers must do?

Developers must add type annotations to newly added code.

Suppose their code is:

```
def power(x, y):  
    ans = x**y  
    return ans
```

Type annotated code:

```
def power(x: float, y: float) -> float:  
    ans = x**y  
    return ans
```

Now, these type annotations will be verified by **mypy**. Mypy is a static type checker. Mypy is a lint-like tool and these annotations are hints for mypy, they don't interfere while running the program.

## Checks to ensure the newly added code is typed

There will be mypy pre-push tests to ensure that the developer has added correct type annotations to the newly added code. In these tests, mypy will check all the changed files.

Moreover there will be pre-push and CI tests to check all the type annotations (existing and newly added) are correct.

There will be two types of files: Type annotated and not type annotated. We will be storing the list of non type annotated files and will be annotating them.

If someone adds code to type-annotated files, added code must also be type-annotated or else there will be a check fail.

If someone adds code to not type-annotated files, there won't be any check fails if he doesn't add type-annotations.

```
schema_utils.py:328: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:345: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:360: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:375: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:387: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:399: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:412: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:425: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:438: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:453: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:465: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:491: error: Function is missing a type annotation [no-untyped-def]
schema_utils.py:501: error: Call to untyped function "get_validator" in typed context [no-untyped-call]
schema_utils.py:504: error: Function is missing a type annotation [no-untyped-def]
```

Here is an example of how the mypy errors will look like. It has the filename, line number, the error message and the error code(Example: no-untyped-def). If we have certain unavoidable errors in our codebase, we can silent a particular error-code.

## Protobuf

Protobuf (or Protocol Buffers) is a way of serializing structured data. It was developed by Google.

## Why Protobuf for our assets/constants.ts?

We are currently making our back-end codebase typed. It is widely adopted and has been used for the past 15 years. Protobuf will help us introduce typing in our constants too. It is simple to write the structure of data using Protobuf. We are already using protobuf in some part of our code. We will be using proto3 in this project. It is the latest version of protobuf.

## After the project is complete

1. There will be pre-push checks to ensure type hints are present and are correct.
2. There will be CI checks to ensure type hints are present and are correct.
3. There will be documentation to show about adding types.
4. There will be a .proto file defining the structure of file which will store the data of assets/constants.ts
5. There will be a file which assigns data which is present in assets/constants.ts to Protobuf.

6. Type hints will be added to core/storage , core/platform and root folder files.

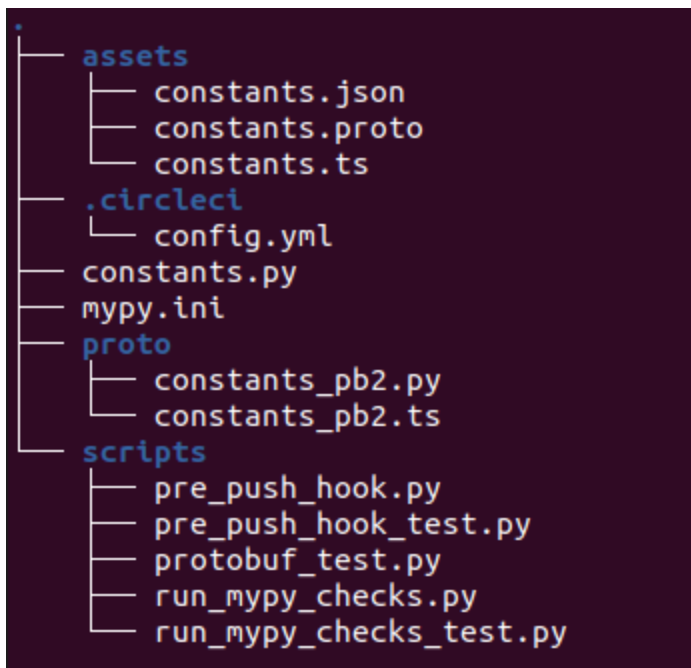
**Note** - Another organisation, [Zulip](#) also uses mypy for implementing strict type checking.

**Note** - A github repo [pgoapi](#) uses protobuf.

## Technical Design

### Architectural Overview

Structure of files to be edited or added in this project



```
├── assets
│   ├── constants.json
│   ├── constants.proto
│   └── constants.ts
├── .circleci
│   └── config.yml
├── constants.py
├── mypy.ini
├── proto
│   ├── constants_pb2.py
│   └── constants_pb2.ts
├── scripts
│   ├── pre_push_hook.py
│   ├── pre_push_hook_test.py
│   ├── protobuf_test.py
│   ├── run_mypy_checks.py
│   └── run_mypy_checks_test.py
```

Following files will be added or edited for this project.

Mypy type checker script

The scripts are written in the directory `scripts/`

The runner script will be `scripts/run_mypy_checks.py`

Mypy configuration file

The configuration file `mypy.ini` will be kept in the root directory as `/mypy.ini`



## Mypy pre-push check

A function to run mypy checks before push will be added in `scripts/pre_push_hook.py`

## Mypy CI check

The CI checks are added in `.circleci/config.yml`

## Mypy tests

Mypy script tests will be written in `scripts/run_mypy_checks_test.py`

Mypy pre\_push\_hook tests will be added in `scripts/pre_push_hook_test.py`

## Protobuf constants.proto file

This file goes to the `assets/` directory. Its path will be `assets/constants.proto`

## Protobuf auto-generated files

Protobuf generated files are stored in the `proto/` directory.

1. Python: `proto/constants_pb2.py`
2. Typescript: `proto/constants_pb2.ts`

## JSON file which stores constants

This file will store the values of our constants. This file will be placed in `assets/constants.json`.

## `constants.py`

This file was used to access `assets/constants.ts`. But now it will use protobuf to parse `assets/constants.json`

## `assets/constants.ts`

This file stored the value of constants and exported these values. Now it will parse `assets/constants.json` and will export the protobuf message.

## Protobuf Tests

Protobuf tests will be written in `scripts/protobuf_test.py`

This project aims at making the complete backend code typed. The final aim(beyond the GSoC period too) of this project is to add type annotations to all the backend python files.

The files to be migrated(adding type annotations) during the GSoC period are the files of `core/storage`, `core/platform` and the root files.

Order of files to be migrated during GSoC period is:

1. `core/storage` files in alphabetic order of folder names
2. `core/platform` files in alphabetic order of folder names
3. Root folder files

## Implementation Approach

This project is divided in 5 major sections:

1. Adding mypy check script, pre-push checks and CI checks

The mypy checks will be done through a script.

The code for this mypy check script is:

```
# scripts/run_mypy_checks.py
import argparse
import os
import subprocess

# import python_utils

# list of non-strictly typed files
not_strict_typed_files = [
    'core/controllers/base.py',
    'core/controllers/admin.py',
    'core/controllers/base_test.py',
    'core/controllers/admin_test.py',
    'scripts',
    'third_party'
```

```

]

CONFIG_FILE_PATH = os.path.join('.', 'mypy-strict.ini')

_PARSER = argparse.ArgumentParser(
    description="""
Type checking script for Oppia codebase.
""")

_PARSER.add_argument(
    '--files',
    help='Files to type-check',
    action='extend',
    nargs='+')

def main(args=None):
    parsed_args = _PARSER.parse_args(args=args)
    # python_utils.PRINT('Type checking the oppia codebase')

    if parsed_args.files:
        process = subprocess.Popen(
            ['mypy', '--config-file', CONFIG_FILE_PATH] +
parsed_args.files,
            stdin=subprocess.PIPE)

        # take files from config-file
    else:
        cmd = ['mypy', '--exclude', '|'.join(not_strict_typed_files),
            '--config-file', CONFIG_FILE_PATH, '.']

        process = subprocess.Popen(cmd, stdin=subprocess.PIPE)

if __name__ == '__main__': # pragma: no cover
    main()

```

I will be improving this code frequently as I explore more about mypy checks.

Explanation of the above code:

1. There will be a configuration file for MyPy. Path for it is stored in `CONFIG_FILE_PATH`.
2. Argparse is used to parse the arguments passed through the command-line.
3. If files are specified after `--files`, the test runs on those files. Else it runs on the whole codebase except the paths specified in `'not_strict_typed_files'`.
4. The subprocess module calls the mypy command to be run.

Why denylist approach?

I think starting off with the denylist approach will be great. In our denylist, we can mention the directories too. For example:

```
not_strict_typed_files = [  
    'core/controllers/base.py',  
    'core/controllers/admin.py',  
    'scripts',  
    'core/storage',  
    'core/domain'  
]
```

This list contains the files with directories. This list will exclude all files of directories `scripts`, `core/storage`, `core/domain` along with files `base.py` and `admin.py` of `core/controllers`.

Here, I am using the `'exclude'` configuration of mypy. It accepts the regex of files to be excluded from the mypy checks. So we can enter the filename and directories both. Therefore using denylist, we won't be required to write down all the files and we can simply add the directory. Moreover this will ensure that any new added file will be strictly typed.

For developers, a basic documentation will be added which will show how to add type annotations. This will prevent them from facing issues relating to adding type annotations to newly created files.

There will be mypy pre-push checks which will check all the strictly typed files and will notify if an already strictly typed file is not strictly typed after the changes.

This check will use mypy check script.

Mypy pre-push check code:

```
# scripts/pre_push_hook.py
def start_mypy_checks():
    task = subprocess.Popen(
        [PYTHON_CMD, '-m', MYPY_MODULE])
    task.communicate()
    return task.returncode
```

Mypy CI check code:

This type of code will be added in .circleci/config.yml and similar in .github/workflows/

```
# .circleci/config.yml
python_type_checks:
  <<: *job_defaults
  steps:
    - checkout
    - merge_target_branch
    - attach_workspace:
      at: /home/circleci/
    - run:
      name: Run python type checks on strictly implemented files
      command: |
        python -m scripts.run_mypy_checks
    - upload_screenshots
```

## 2. Adding mypy configuration to config-file mypy.ini

It won't allow untyped functions and variables and will give type checking error whenever untyped functions, classes and variables are written.

mypy.ini :

```
[mypy]
python_version = 3.7
```

```
ignore_missing_imports = True

show_error_codes = True

follow_imports = skip

strict = True
```

Explanation of mypy.ini file:

1. `python_version`: takes the python version used.
2. `ignore_missing_imports`: If true, tells mypy to ignore all unresolved imports.
3. `show_error_codes`: If true, adds error code to error messages.
4. `follow_imports`: Tells mypy how it should follow imports. 'skip' will not follow imports and will silently replace module with an object of type 'Any'.

Following are the configurations set by `strict = True`

5. `warn_unused_configs`: This flag makes mypy warn about unused [mypy-<pattern>] config file sections.
6. `disallow_any_generics`: This flag disallows usage of generic types that do not specify explicit type parameters.
7. `disallow_subclassing_any`: This flag reports an error whenever a class subclasses a value of type 'Any'.
8. `disallow_untyped_calls`: This flag reports an error whenever a function with type annotations calls a function defined without annotations.
9. `disallow_untyped_defs`: This flag reports an error whenever it encounters a function definition without type annotations.
10. `disallow_incomplete_defs`: This flag reports an error whenever it encounters a partly annotated function definition.
11. `check_untyped_defs`: This flag checks the body of every function, regardless of whether it has type annotations (By default the bodies of functions without annotations are not type checked).

12. `disallow_untyped_decorators`: This flag reports an error whenever a function with type annotations is decorated with a decorator without annotations.
13. `no_implicit_optional`: This flag causes mypy to stop treating arguments with a `None` default value as having an implicit 'Optional' type.
14. `warn_redundant_casts`: This flag will make mypy report an error whenever your code uses an unnecessary cast that can safely be removed.
15. `warn_unused_ignores`: This flag will make mypy report an error whenever your code uses a `# type: ignore` comment on a line that is not actually generating an error message.
16. `warn_return_any`: This flag causes mypy to generate a warning when returning a value with type `Any` from a function declared with a non-`Any` return type.
17. `no_implicit_reexport`: By default, imported values to a module are treated as exported and mypy allows other modules to import them. This flag changes the behavior to not re-export unless the item is imported using `from-as` or is included in `__all__`.
18. `strict_equality`: This flag prohibits comparisons of non-overlapping types.

### 3. Adding Documentation

A wiki page will be added to demonstrate how to add type annotations. And it will also show how to run mypy checks script.

### 4. Converting constants.ts to Protobuf

#### 4.1 Writing constants.proto

The constants.ts file has a lot of data. Here I am showing a .proto file for a part of the constants.ts file. This proto file defines message formats.

Code snippet of constants.proto:

```
syntax = "proto3";  
  
message constants {
```

```

bool CAN_SEND_ANALYTICS_EVENTS = 1;
string CLASSROOM_URL_FRAGMENT_FOR_UNATTACHED_TOPICS = 2;
string DEFAULT_CLASSROOM_URL_FRAGMENT = 3;
repeated string ALL_CATEGORIES = 4;
int32 MAX_COMMIT_MESSAGE_LENGTH = 5;
message svg_attrs_whitelist {
    repeated string a = 1;
    repeated string altglyph = 2;
    repeated string altglyphdef = 3;
}
svg_attrs_whitelist SVG_ATTRS_WHITELIST = 6;

message supported_content_languages {
    string code = 1;
    string description = 2;
    string direction = 3;
}
repeated supported_content_languages SUPPORTED_CONTENT_LANGUAGES = 7;
}

```

#### 4.2 Compiling constants.proto to python

The constants.proto file is compiled using the protocol buffer compiler from the protocol definition. Code for running the compiler:

```
protoc -I=assets/ --python_out=assets/ assets/constants.proto
```

This creates a python file that has the automatically generated code.

The compilation of constants.proto will be done in scripts/install\_third\_party\_libs.py . This is because compilation of other .proto files is also done there.

#### 4.3 Compiling constants.proto to typescript

For this, we use third-party library [ts-proto](#). It generates .ts file from .proto files.

Code for running the compiler:

```
protoc --plugin=./node_modules/.bin/protoc-gen-ts_proto
--ts_proto_out=../.. --ts_proto_opt=snakeToCamel=false
assets/constants-p.proto
```



snakeToCamel=false keeps the names in snake case. By default it is converted to camel case. The compilation of constants.proto will be done in scripts/install\_third\_party\_libs.py . This is because compilation of other .proto files is also done there.

#### 4.4 Storing constants in assets/constants.json

This JSON file will store values of constants.

```
{
  "MAX_COMMIT_MESSAGE_LENGTH": 375,
  "CLASSROOM_URL_FRAGMENT_FOR_UNATTACHED_TOPICS": "staging",
  "ALL_CATEGORIES": [
    "Algebra",
    "Algorithms",
    "Architecture"
  ],
  "DEFAULT_CLASSROOM_URL_FRAGMENT": "math",
  "CAN_SEND_ANALYTICS_EVENTS": true,
  "SVG_ATTRS_WHITELIST": {
    "a": [
      "about",
      "alignment-baseline",
      "baseline-shift",
      "class"
    ],
    "altglyph": [
      "about",
      "alignment-baseline",
      "baseline-shift"
    ],
    "altglyphdef": [
      "about",
      "class",
      "content"
    ]
  },
  "SUPPORTED_CONTENT_LANGUAGES": [{
    "code": "en",
    "description": "English",
    "direction": "ltr"
  }, {
    "code": "ar",
```

```

    "description": "العربية (Arabic)",
    "direction": "rtl"
  }, {
    "code": "sq",
    "description": "shqip (Albanian)",
    "direction": "ltr"
  }
]
}

```

#### 4.5 Reading from JSON and exporting constants Typescript

The assets/constants.ts file needs to be modified so that it exports the protobuf message.

```

// assets/constants.ts
import {constants} from '../proto/constants_pb2'
var fs = require('fs')
var data = fs.readFileSync('assets/constants.json', 'utf8');

data = JSON.parse(data)

const constants_list = constants.fromJSON(data)
export default constants_list

```

#### 4.6 Reading from JSON and exporting constants Python

The /constants.py file needs to be modified to read assets/constants.json and not assets/constants.ts and then parse to protobuf message.

```

# /constants.py
from __future__ import absolute_import # pylint:
disable=import-only-modules
from __future__ import unicode_literals # pylint:
disable=import-only-modules

import json
import os
import re
from google.protobuf.json_format import Parse

import python_utils
from proto import constants_pb2

```

```

with python_utils.open_file(os.path.join('assets', 'constants.json'), 'r')
as f:
    constants = constants_pb2.constants()
    Parse(f.read(), constants)

with python_utils.open_file('release_constants.json', 'r') as f:
    release_constants = Constants(json.loads(f.read())) #
pylint:disable=invalid-name

```

Why JSON instead of textproto?

Here are the examples of both.

#### 1. JSON

```

{
  "MAX_COMMIT_MESSAGE_LENGTH": 375,
  "CLASSROOM_URL_FRAGMENT_FOR_UNATTACHED_TOPICS": "staging",
  "ALL_CATEGORIES": [
    "Algebra",
    "Algorithms",
    "Architecture"
  ],
  "DEFAULT_CLASSROOM_URL_FRAGMENT": "math",
  "CAN_SEND_ANALYTICS_EVENTS": true,
  "SVG_ATTRS_WHITELIST": {
    "a": [
      "about",
      "alignment-baseline",
      "baseline-shift",
      "class"
    ],
    "altglyph": [
      "about",
      "alignment-baseline",
      "baseline-shift"
    ],
    "altglyphdef": [
      "about",
      "class",
      "content"
    ]
  }
}

```

```

]
},

  "SUPPORTED_CONTENT_LANGUAGES": [{
    "code": "en",
    "description": "English",
    "direction": "ltr"
  }, {
    "code": "ar",
    "description": "العربية (Arabic)",
    "direction": "rtl"
  }, {
    "code": "sq",
    "description": "shqip (Albanian)",
    "direction": "ltr"
  }]
}

```

## 2. Textproto

```

CAN_SEND_ANALYTICS_EVENTS: true
CLASSROOM_URL_FRAGMENT_FOR_UNATTACHED_TOPICS: "staging"
DEFAULT_CLASSROOM_URL_FRAGMENT: "math"
ALL_CATEGORIES: "Algebra"
ALL_CATEGORIES: "Algorithms"
ALL_CATEGORIES: "Architecture"
MAX_COMMIT_MESSAGE_LENGTH: 375
SVG_ATTRS_WHITELIST {
  a: "about"
  a: "alignment-baseline"
  a: "baseline-shift"
  a: "class"
  altglyph: "about"
  altglyph: "alignment-baseline"
  altglyph: "baseline-shift"
  altglyphdef: "about"
  altglyphdef: "class"
  altglyphdef: "content"
}
SUPPORTED_CONTENT_LANGUAGES {

```

```

code: "en"
description: "English"
direction: "ltr"
}
SUPPORTED_CONTENT_LANGUAGES {
code: "ar"
description: "\330\247\331\204\330\271\330\261\330\250\331\212\330\251
(Arabic)"
direction: "rtl"
}
SUPPORTED_CONTENT_LANGUAGES {
code: "sq"
description: "shqip (Albanian)"
direction: "ltr"
}

```

Here, we see that JSON is more readable and developers know how to edit JSON files. We see that the how 'SUPPORTED\_CONTENT\_LANGUAGES', 'ALL\_CATEGORIES' and other list like elements are written in text-proto, each element of a list is added independently but in JSON we can add element like a normal list.

## 5. Adding type annotations

For this, we can use third party library [MonkeyType](#) for adding type annotations.

**Note** - MonkeyType will be used temporarily for adding type annotations to existing codebase in this project. Developers will be writing type annotations themselves when they write code. This library will only be used as an aid during the project.

Our codebase has 100% test coverage. This will help us a lot in adding type annotations.

Let's say we have a file main.py:

```

# main.py
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

```

It has a test file test.py:

```
# test.py
import main

assert(main.add('a', 'b') == 'ab')
assert(main.add(1, 2) == 3)
assert(main.subtract(2,1) == 1)
```

Now we use MonkeyType for adding type annotation to main.py

We run:

```
monkeytype run test.py
```

This collect data about types of function used in main.py

Then we run:

```
monkeytype apply main
```

This modifies our main.py to:

from typing import Union

```
# main.py
from typing import Union

def add(x: Union[int, float, str], y: Union[int, float, str]) -> Union[int, float, str]:
    return x + y

def subtract(x: Union[int, float], y: Union[int, float]) -> Union[int, float]:
    return x - y
```

Adding type annotations to python files won't affect the functioning of any of the python files. The only issue here can be errors in running MyPy tests. This issue is solved by maintaining a list of files and directories which don't have type annotations. This will prevent running of type annotation checks on untyped files.

There was one other issue, if a non-typed file is imported in a typed file, non-typed file's type checking was also done. This problem was solved using the configuration 'follow\_imports = skip'. This won't type check the imported files. This skipping of following imports has some

drawbacks like the imported files function's type definition will not be available to the code and the type will be 'Any'. This reduces the strictness of checks. So this loss of strictness can be revived after our complete backend codebase is typed. This can be done as we can allow the imports to be followed and the imported objects will also have their types after the whole codebase is typed.

While the backend is in a hybrid state, we cannot implement the checks which are very strict. We must start off with skipping following imports and suppress errors of a particular code that arise due to this. Then we can slowly add type annotation to the codebase. When it is fully complete, we can increase the strictness.

## Why MonkeyType?

MonkeyType is developed by Instagram. It works for Python3 and is configurable. We can add type annotations using MonkeyType and then verify these annotations. There won't be much changes and this will help in making the codebase typed.

Examples of adding type annotation to code

Without type annotations:

```
def multiply_numbers(numbers):
    ans = 1.0
    for number in numbers:
        ans = ans * number

    return ans
```

With type annotations:

```
from typing import List

def multiply_numbers(numbers: List[float] ) -> float:
    ans = 1.0
    for number in numbers:
        ans = ans * number

    return ans
```

Example of adding type annotation in core/storage/activity/gae\_models.py

Before:

```
from __future__ import absolute_import # pylint:
disable=import-only-modules
from __future__ import unicode_literals # pylint:
disable=import-only-modules
```

```

from core.platform import models
import core.storage.base_model.gae_models as base_models
import feconf

datastore_services = models.Registry.import_datastore_services()

class ActivityReferencesModel(base_models.BaseModel):
    """Storage model for a list of activity references.

    The id of each model instance is the name of the list. This should be
    one
    of the constants in feconf.ALL_ACTIVITY_REFERENCE_LIST_TYPES.
    """

    # The types and ids of activities to show in the library page. Each
    item
    # in this list is a dict with two keys: 'type' and 'id'.
    activity_references = datastore_services.JsonProperty(repeated=True)

    @staticmethod
    def get_deletion_policy():
        """Model doesn't contain any data directly corresponding to a
        user."""
        return base_models.DELETION_POLICY.NOT_APPLICABLE

    @staticmethod
    def get_model_association_to_user():
        """Model does not contain user data."""
        return
base_models.MODEL_ASSOCIATION_TO_USER.NOT_CORRESPONDING_TO_USER

    @classmethod
    def get_export_policy(cls):
        """Model doesn't contain any data directly corresponding to a
        user."""
        return dict(super(cls, cls).get_export_policy(), **{
            'activity_references': base_models.EXPORT_POLICY.NOT_APPLICABLE
        })

    @classmethod
    def get_or_create(cls, list_name):

```



```

"""This creates the relevant model instance, if it does not already
exist.
"""
if list_name not in feconf.ALL_ACTIVITY_REFERENCE_LIST_TYPES:
    raise Exception(
        'Invalid ActivityListModel id: %s' % list_name)

entity = cls.get(list_name, strict=False)
if entity is None:
    entity = cls(id=list_name, activity_references=[])
    entity.update_timestamps()
    entity.put()

return entity

```

After:

```

from __future__ import absolute_import # pylint:
disable=import-only-modules
from __future__ import unicode_literals # pylint:
disable=import-only-modules

from core.platform import models
import core.storage.base_model.gae_models as base_models
import feconf
from typings import Dict

datastore_services = models.Registry.import_datastore_services()

class ActivityReferencesModel(base_models.BaseModel):
    """Storage model for a list of activity references.

    The id of each model instance is the name of the list. This should be
    one
    of the constants in feconf.ALL_ACTIVITY_REFERENCE_LIST_TYPES.
    """

    # The types and ids of activities to show in the library page. Each
    item
    # in this list is a dict with two keys: 'type' and 'id'.
    activity_references = datastore_services.JsonProperty(repeated=True)

```

```

@staticmethod
def get_deletion_policy() -> str:
    """Model doesn't contain any data directly corresponding to a
user."""
    return base_models.DELETION_POLICY.NOT_APPLICABLE

@staticmethod
def get_model_association_to_user() -> str:
    """Model does not contain user data."""
    return
base_models.MODEL_ASSOCIATION_TO_USER.NOT_CORRESPONDING_TO_USER

@classmethod
def get_export_policy(cls) -> Dict[str, str]:
    """Model doesn't contain any data directly corresponding to a
user."""
    return dict(super(cls, cls).get_export_policy(), **{
        'activity_references': base_models.EXPORT_POLICY.NOT_APPLICABLE
    })

@classmethod
def get_or_create(cls, list_name: str) -> ActivityReferencesModel:
    """This creates the relevant model instance, if it does not already
exist.
"""
    if list_name not in feconf.ALL_ACTIVITY_REFERENCE_LIST_TYPES:
        raise Exception(
            'Invalid ActivityListModel id: %s' % list_name)

    entity = cls.get(list_name, strict=False)
    if entity is None:
        entity = cls(id=list_name, activity_references=[])
        entity.update_timestamps()
        entity.put()

    return entity

```

## Third-party Libraries\*

1. MyPy: This library is needed to do static type checks on the codebase. It is licensed under [MIT license](#).
2. Monkey Type: This library will be used as an aid for adding types to existing files. It is licensed under [BSD license](#). This will be used temporarily by me locally. There is no need to add it to the requirements.
3. ts-proto: This library will be used to compile .proto into TypeScript file. It is licensed under [Apache License 2.0](#) .

## Testing Approach

In this project, we are adding mypy check script. There will be tests for this script.

```
# scripts/run_mypy_checks_test.py
from __future__ import absolute_import # pylint: disable=import-only-modules
from __future__ import unicode_literals # pylint: disable=import-only-modules

import subprocess
import tempfile

from core.tests import test_utils

import python_utils

PYTHON_CMD = 'python3'
MYPY_SCRIPT_MODULE = 'scripts.run_mypy_checks'

class MypyCheckTests(test_utils.GenericTestBase):

    def setUp(self):
        super(MypyCheckTests, self).setUp()

    def test_mypy_valid(self):
        tmpfile = tempfile.NamedTemporaryFile(suffix='.py')
        tmpfile.write(
            "def add(x: float, y: float) -> float:\n"
            "    return x + y"
        )
        tmpfile.seek(0)
        cmd = [PYTHON_CMD, '-m', MYPY_SCRIPT_MODULE, '--files', tmpfile.name]
```

```

        process = subprocess.Popen(cmd, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
        output = process.communicate()
        tmpfile.close()
        self.assertEqual(output[0], 'Success: no issues found in 1 source file\n')

def test_mypy_invalid(self):
    tmpfile = tempfile.NamedTemporaryFile(suffix='.py')
    tmpfile.write(
        "def add(x, y):\n"
        "    return x + y"
    )
    tmpfile.seek(0)
    cmd = [PYTHON_CMD, '-m', MYPY_SCRIPT_MODULE, '--files', tmpfile.name]
    process = subprocess.Popen(cmd, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
    output = process.communicate()
    tmpfile.close()
    self.assertIn('error', output[0])

```

Explanation of the code:

1. The tests in this test class use tempfile. These are files which are created temporarily and here are used only for the testing purpose.
2. Then the subprocess module is used to run the mypy checks script on the temporarily created file.
3. The output of the runs is then checked.

There will also be tests for pre\_push\_hook.py and they will be added in pre\_push\_hook\_test.py

There will be tests for protobuf:

```

# scripts/protobuf_test.py
from __future__ import absolute_import # pylint:
disable=import-only-modules
from __future__ import unicode_literals # pylint:
disable=import-only-modules

from core.tests import test_utils
from proto import constants_pb2

import python_utils

```

```

class ProtobufTests(test_utils.GenericTestBase):

    def setUp(self):
        super(ProtobufTests, self).setUp()

    def test_protobuf_equal(self):
        constants = constants_pb2.constants()
        constants.CAN_SEND_ANALYTICS_EVENTS = True;
        constants.ALL_CATEGORIES.append('A')
        constants.ALL_CATEGORIES.append('B')
        constants.ALL_CATEGORIES.append('C')

        json = MessageToJson(constants)

        new_constants = constants_pb2.constants()
        Parse(json, new_constants)

        self.assertEqual(constants, new_constants)

    def test_protobuf_not_equal(self):
        constants = constants_pb2.constants()
        constants.CAN_SEND_ANALYTICS_EVENTS = True;
        constants.ALL_CATEGORIES.append('A')
        constants.ALL_CATEGORIES.append('B')
        constants.ALL_CATEGORIES.append('C')

        new_constants = constants_pb2.constants()
        new_constants.CAN_SEND_ANALYTICS_EVENTS = False;

        self.assertNotEqual(constants, new_constants)

```

The protobuf tests verify that when a protobuf message is converted to bytes and then converted back to protobuf message, the message remains the same.

Now we can add a test which reads the JSON file and imports the protobuf message 'constants'. Then we can compare the keys of the JSON file and the fields of empty protobuf message 'constants' to ensure that the JSON file constants.json has all the keys which are defined in the proto definition constants.proto.

```

class ProtobufConsistentWithJSONTests(test_utils.GenericTestBase):

    def setUp(self):
        super(ProtobufConsistentWithJSONTests, self).setUp()

    def test_json_is_consistent_with_json(self):
        constants = constants_pb2.constants()

        with python_utils.open_file(JSON_PATH, 'r') as f:
            data = json.loads(f.read())
            json_keys = set(data.keys())
            protobuf_keys = set(constants.DESRIPTOR.fields_by_name.keys())

        self.assertEqual(json_keys, protobuf_keys)

```

## Milestones

### Milestone 1

7 June - 12 July

**Key Objective:** The objectives are:

1. Adding mypy script
2. Adding mypy pre-push check to check the type annotations
3. Adding mypy CI check to check the type annotations
4. Replacing assets/constants.ts with protobuf messages

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Add mypy type checking script		13/06/2021	20/06/2021
1.2	Add mypy pre-push check to check type annotations	1.1	16/06/2021	23/06/2021
1.3	Add mypy CI check to check type	1.1	20/06/2021	27/06/2021

	annotations			
1.4	Add documentation for running mypy checker script and how to add type annotations		21/06/2021	28/06/2021
1.5	Add constants.proto file and add the compilation of .proto to python and typescript		25/06/2021	02/07/2021
1.6	Add constants.json, edit constants.py and constants.ts to migrate to protobuf	1.5	28/06/2021	05/07/2021
1.7	Create an issue for type annotation to be added to the codebase.	1.1-1.4	28/06/2021	-

## Milestone 2

13 July - 23 August

**Key Objective:** The objective is adding type annotations to:

1. core/storage
2. core/platform
3. root files

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Add type annotations to core/storage files of: <ul style="list-style-type: none"> <li>• activity</li> <li>• app_feedback_report</li> <li>• audit</li> <li>• auth</li> </ul>		03/07/2021	10/07/2021

2.2	<p>Add type annotations to core/storage files of:</p> <ul style="list-style-type: none"> <li>• base_model</li> <li>• classifier</li> <li>• collections</li> </ul>		08/07/2021	15/07/2021
2.3	<p>Add type annotations to core/storage files of:</p> <ul style="list-style-type: none"> <li>• config</li> <li>• email</li> <li>• exploration</li> <li>• feedback</li> </ul>		13/07/2021	20/07/2021
2.4	<p>Add type annotations to core/storage files of:</p> <ul style="list-style-type: none"> <li>• improvement</li> <li>• job</li> <li>• opportunity</li> <li>• question</li> <li>• recommendation</li> <li>• skill</li> </ul>		17/07/2021	24/07/2021
2.5	<p>Add type annotations to core/storage files of:</p> <ul style="list-style-type: none"> <li>• statistics</li> <li>• story</li> <li>• subtopic</li> <li>• suggestion</li> <li>• topic</li> </ul>		24/07/2021	31/07/2021
2.6	<p>Add type annotations to core/platform files of :</p> <ul style="list-style-type: none"> <li>• user</li> <li>• app_identity</li> </ul>		30/07/2021	06/08/2021



	<ul style="list-style-type: none"> <li>• auth</li> </ul>			
2.7	Add type annotations to core/platform files of : <ul style="list-style-type: none"> <li>• cache</li> <li>• datastore</li> <li>• email</li> <li>• search</li> <li>• taskqueue</li> <li>• transactions</li> <li>• users</li> </ul>		04/08/2021	11/08/2021
2.8	Add type annotations to files in root folder and core/platform file model.py		10/08/2021	17/08/2021

## Optional Sections

### Additional Project-Specific Considerations

#### Privacy

No, this project does not collect additional user data.

#### Security

No, this project does not provide any new opportunities for users to gain unauthorized access. This project will only add scripts for checking python type annotation, change the constants.ts file and add type annotations to python files.

#### Accessibility (if user-facing)

No, this project is not user-facing.

## Documentation Changes

A wiki page will be added to show how mypy checks script works.

The documentation will have:

1. Reasons as to why we need type annotations and static type checks.
2. Steps to run mypy checks script for the whole codebase and for a particular file.
3. Basics on how to add type annotations.
4. Steps to add new constants in constants.json. This will require changes in both proto file and json file.

## Ethics

No, there are no ethical considerations that should be taken into account.

## Future Work

I will be opening an issue for adding type annotation to files.

These files will be all the python files except core/storage , core/platform and root folder files.

After all the files are type-annotated. Mypy Tests can be made more strict and 'following import' restrictions can be lifted and changes can be made accordingly.