Muddassar Sharif, Virgil, Matthew
Prof. Li Hua
Software Engineering Class Project

Auto Grader

Table Of Contents

1. Project Description

While computer science class can be fun for so many students, it might not be for the professor or the teaching assistant responsible for checking and grading all the assignments. Auto Grader aims to streamline the process of assigning, submitting, and grading homework assignments for everyone involved. A simple and clean web frontend allows for professors to login and post assignments that only users in their class can view. The students can then login, view the assignment and its specifications, and submit their code. Once the assignment deadline passes, the code will automatically be graded using the AutoGrader API that we developed, providing feedback on the successes and fails of the code on the professor-submitted test cases. Students will be able to see their results after the deadline, as well as see any errors that might have occurred during the running of their programs. Included is a progress report for each assignment, allowing professors to quickly get an idea of how an assignment went for the class overall. Compare this to the previous method: (1) students submitting code on NYUClasses; (2) professors downloading it and running it themselves, noting where the code succeeded and where it failed; and (3) manually uploading the grade of each assignment for the students to see, with no feedback provided. It is easy to see why this project was not only beneficial and maybe even necessary.

The architecture is split into an internal API that generates data based on the code provided and a client that consumes the API endpoints. Due to time constraints, the 2 parts have not been merged, but provide individual functionality.

2. Project Requirements and Specifications

There were four major requirements for the API

Code submission

The user should be able to submit the code using a couple of ways: One, directly copy and paste the code or in other words, submit as a string. Second, submit the code file.

Test case submission

Similarly, the test cases can also be submitted as a file, or the user can just provide inputs and outputs.

Max number of lines checker

The program should be able to return the number of lines and give an error message if the number of lines is more than the number specified by the teacher.

Plagiarism Checker.

Check all the submission of an assignment to see if two or more submissions are similar and return a confidence percentage using MOSS

Database

All of this should be recorded and easily accessible on an internal database.

Login

On the client side, professors should be able to login, upload homework sets and have access to students records while students should be able to login and submit their homework to the automated checker.

3. Team Process

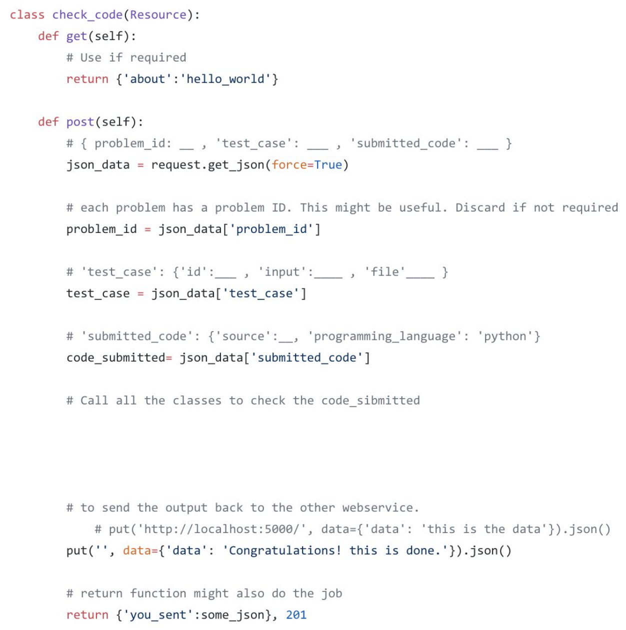
During development, we followed the XP development framework. We also placed a heavy emphasis on continuous integration and testing using Travis CI and the open-closed principle when designing our architecture.

For each iteration we had a developer meeting, were everyone was assigned a task to implement that built on the features that were implemented by that same person in the weeks prior. Over the course of the iteration we also had short 10-minute presentations to discuss progress and potential setbacks, so we could adjust our timeline if necessary.

In terms of testing, we agreed on the philosophy that everyone should write their own tests for the code they are committing on each iteration, and adjust the Travis script so the built is automatically checked when commits are pushed to Github.

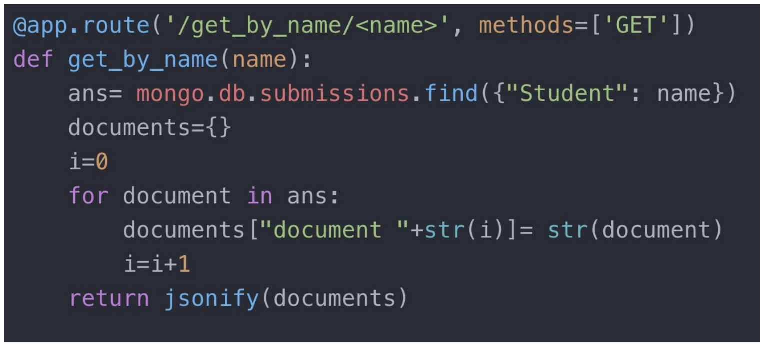
Refactoring.

Initially, when designing the API, we had one class with a couple of methods for each API route and it worked until a point. The diagram below shows part of the original API class.

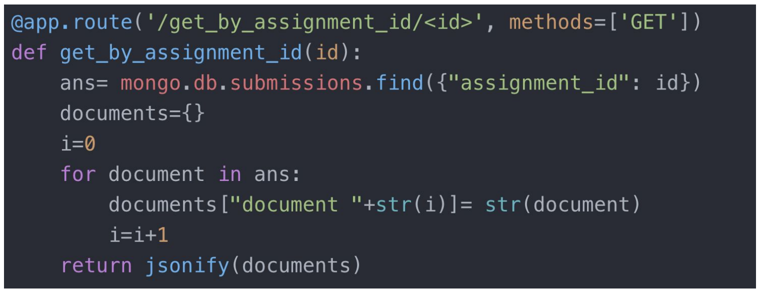


```
class check_code(Resource):
    def get(self):
        # Use if required
        return {'about':'hello_world'}

    def post(self):
        # { problem_id: __ , 'test_case': ___ , 'submitted_code': ___ }
        json_data = request.get_json(force=True)

        # each problem has a problem ID. This might be useful. Discard if not required
        problem_id = json_data['problem_id']

        # 'test_case': {'id':___ , 'input':____ , 'file'____ }
        test_case = json_data['test_case']

        # 'submitted_code': {'source':__, 'programming_language': 'python'}
        code_submitted= json_data['submitted_code']

        # Call all the classes to check the code_sibmitted




        # to send the output back to the other webservice.
            # put('http://localhost:5000/', data={'data': 'this is the data'}).json()
        put('', data={'data': 'Congratulations! this is done.'}).json()

        # return function might also do the job
        return {'you_sent':some_json}, 201
```

As we were adding more API methods, it became really difficult to extend the code. So using the Single Responsibility Principle, we decided to split the API into individual components. In the revised version, we did not use classes. We have one separate function for each API route. Diagrams below show two revised APIs' codes.



```
@app.route('/get_by_name/<name>', methods=['GET'])
def get_by_name(name):
    ans= mongo.db.submissions.find({"Student": name})
    documents={}
    i=0
    for document in ans:
        documents["document "+str(i)]= str(document)
        i=i+1
    return jsonify(documents)
```

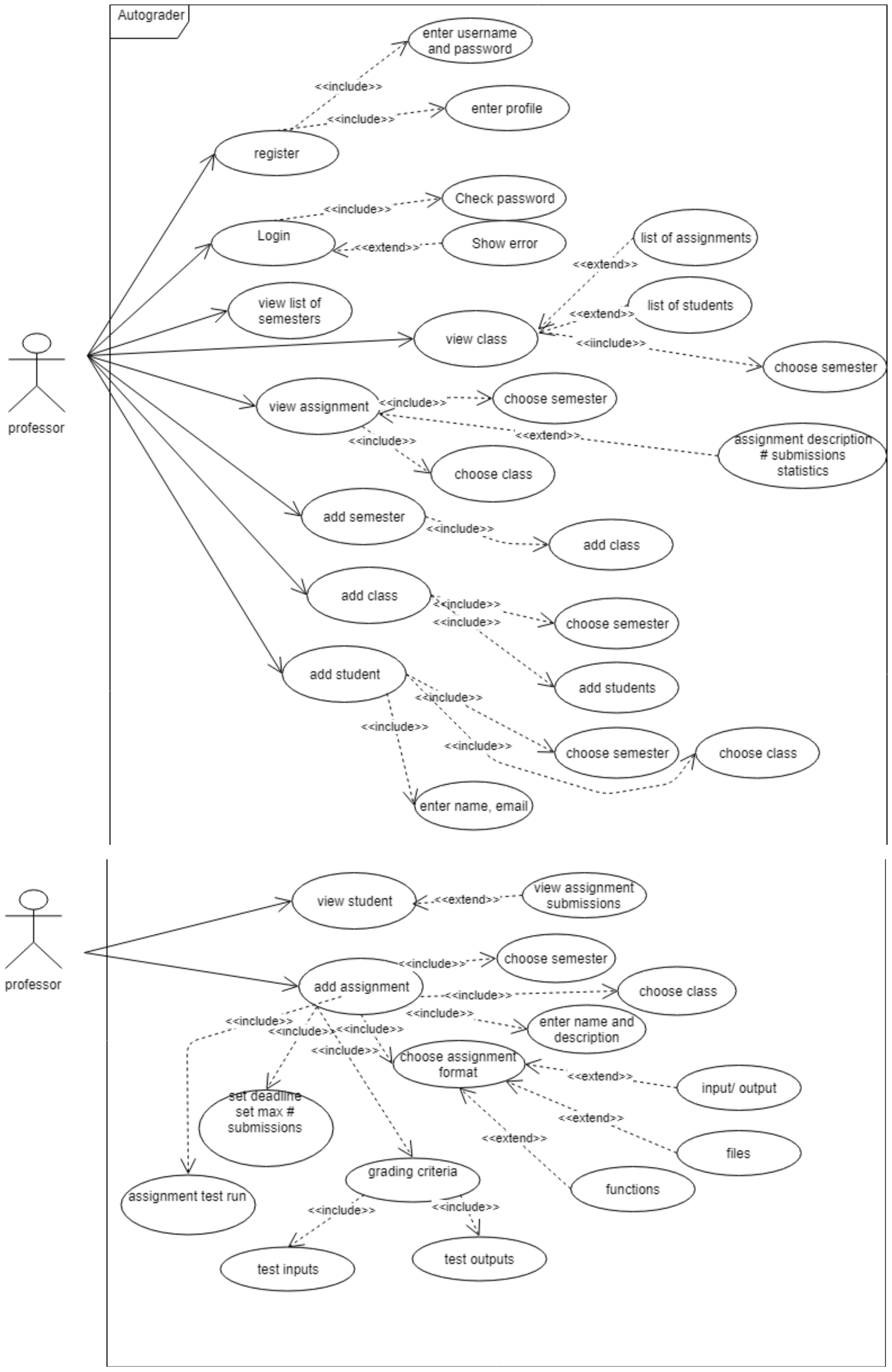


```
@app.route('/get_by_assignment_id/<id>', methods=['GET'])
def get_by_assignment_id(id):
    ans= mongo.db.submissions.find({"assignment_id": id})
    documents={}
    i=0
    for document in ans:
        documents["document "+str(i)]= str(document)
        i=i+1
    return jsonify(documents)
```

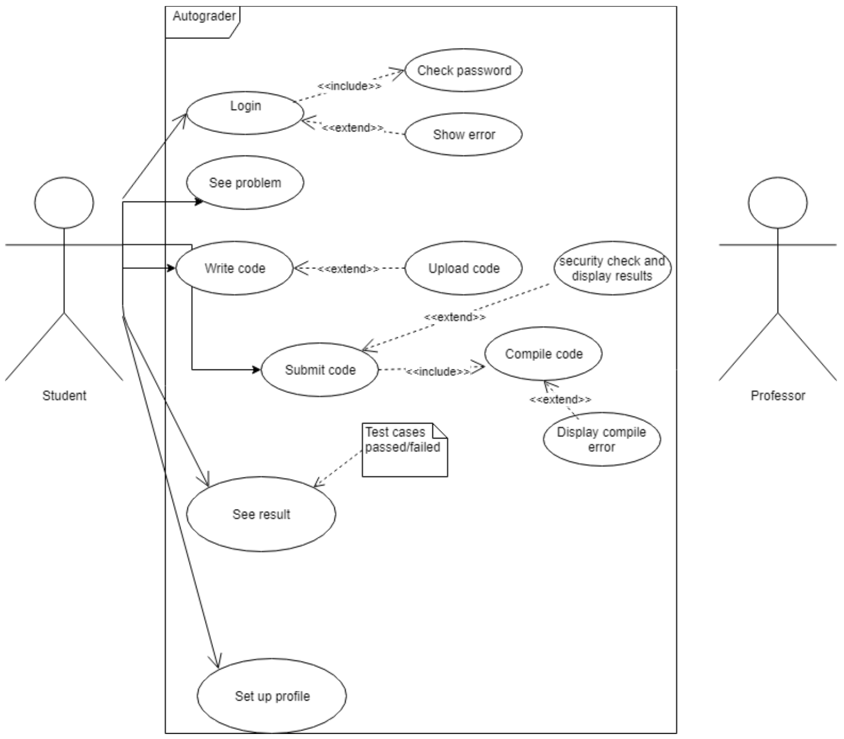
Another change in the specification was our decision to use MongoDB. We implemented the SQL before and it worked fine, but the structural database is not good to store unstructured data such as student's submitted code. We originally thought of storing the code in a local directory, but finally decided that MongoDB is the better database that can store code and other information easily at one place.

4. Architecture and Design

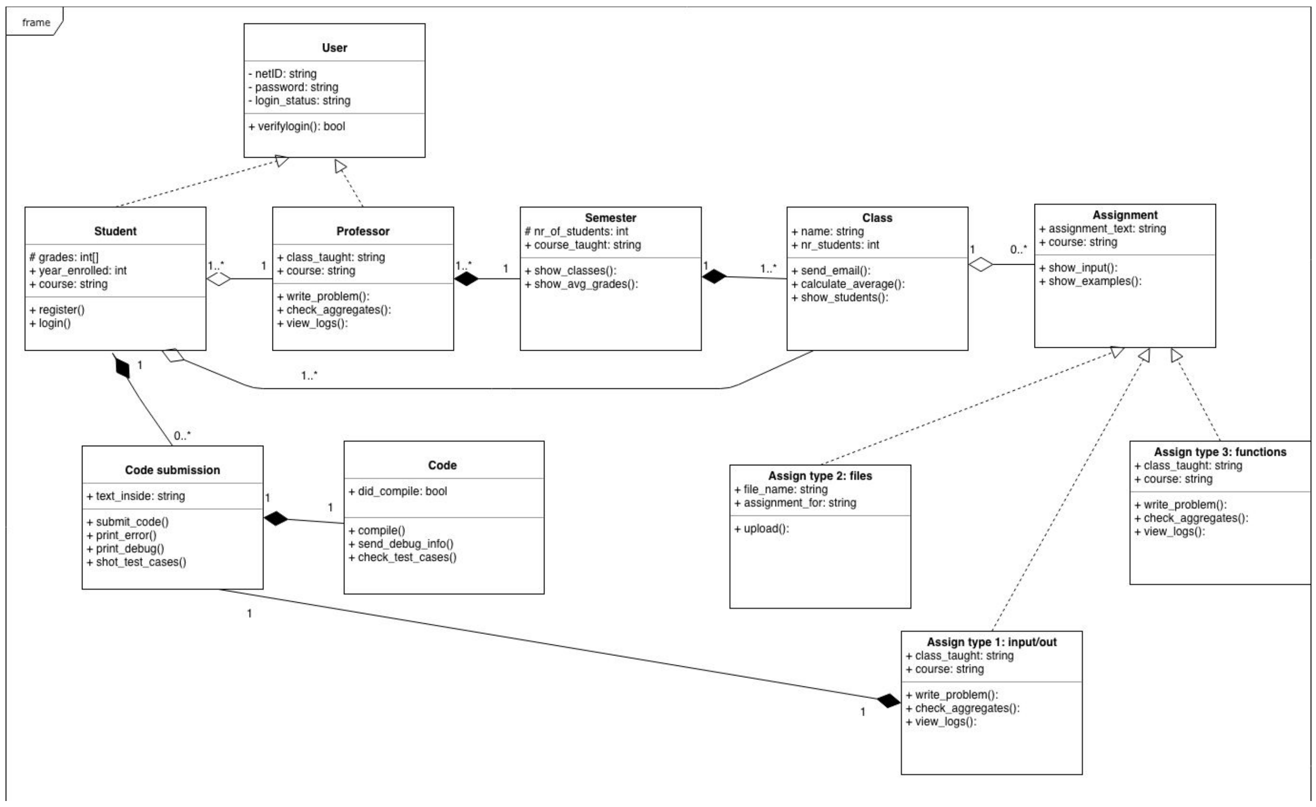
Professor Use case diagram



Student use case diagram

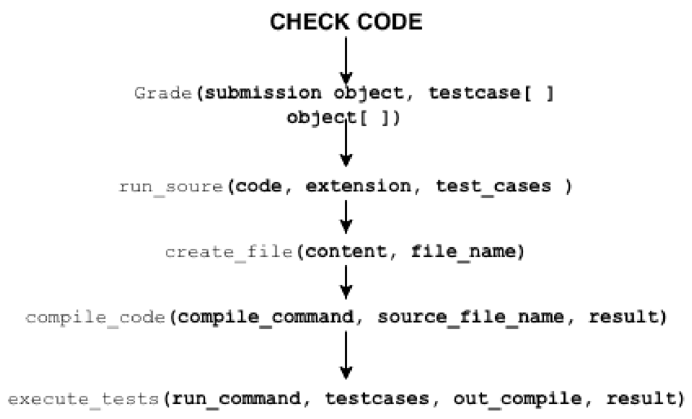


System Class Diagram



All the features described in **Part 3** have been implemented using an internal REST API built with Python and Flask. Flask is a framework choice for building REST APIs as it's easy to set up and integrate endpoints into an existing Python codebase. It also works very well with MongoDB, which was our framework of choice for storing submissions data and API logs.

Our architecture is based on the function-oriented design philosophy, with most of the logic coming from functions and classes being used primarily for storing state (meaning few methods). Most of the functions make use of the `grade` function that takes the code as a string and is able to generate and run a file with the given code inside. The following diagram demonstrates how the series of functions get executed when an API to check to check a submission gets called.


CHECK CODE
Grade(submission object, testcase[ ] object[ ])
run_soure(code, extension, test_cases)
create_file(content, file_name)
compile_code(compile_command, source_file_name, result)
execute_tests(run_command, testcases, out_compile, result)

From that function, we derive others that specifically implement the requirements given in **Part 3** and expose the logic through API endpoints to our front-end client.

The API documentation together with example calls can be accessed here:

- https://app.swaggerhub.com/apis-docs/ms8909/AutoGrader/1.0.0

5. Reflection and Lesson Learned

(i) Virgil Tataru

The project was really interesting because it really showed the importance of planning ahead and following a release schedule. While none of the individual parts are very difficult to implement, fitting them together to get the resulted outcome and in a way that fits the open-closed principle makes every decision important, because a wrong one could make our job more difficult down the line. It was the first project where the complexity made this type of decision very important, and as a result we ended up spending a large amount of time discussing architecture and system design, as opposed to just hacking together a working solution fast. This project was very good at showcasing how software development works in the industry and teaching skills that are not often necessary to complete school projects.

(ii) Muddassar Sharif

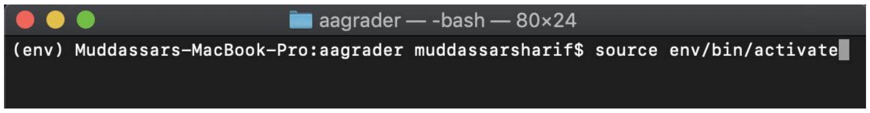
It was my first-semester long group project and it was exciting that we were able to simulate how teams actually work in the industry. Having deadlines every two weeks actually helped us keep up with the work and deliver the API in the end, and I am happy that I was able to complete my part of the work. I introduced pair programming in my startup and that did work as we are able to meet our deadlines before time, but for pair programming to work, at least two people have to be at the same place at the same time, which was not the case for our team as everyone had different schedules, and it was hard to find time overlap. Our decision then to split up the work so that everyone has to work on one type of work really helped. Design patterns were also really helpful, and it helped me a lot when simplifying the API development and I explained this more in the refactoring part. Applying design patterns is a hard skill that needs time to master, but it is good that we were exposed to the concepts in the class.

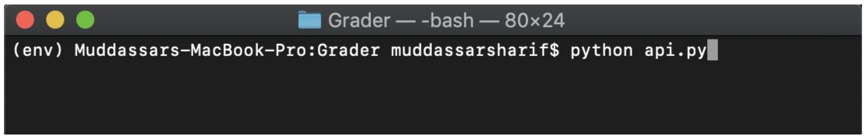
(iii) Matthew Couch

This was my first project that included any “professional” aspect to it. Prior to this course, I hadn’t even heard of almost any of the things we included in our documentation, such as UML, class diagrams, etc. I didn’t really know how to work on a project that involved multiple people (and I’m still not too experienced, due to my initial lack of involvement that led to me working separately just so that I could make sure I had finished something). I learned that using programs that help generate some basic code (in regards to HTML and CSS; I initially used a program to get the basic layout) can lead to lots of future problems. Right now it is very difficult to edit the CSS and HTML; it took me a longer time to figure out the proper way to do so than it would have for me to personally hand write everything from the get-go. In the end, I realized that the project isn't so intimidating after all. If I worked on what we planned to work on from the beginning, I think we could have seen all of our initial goals out. Breaking everything down into little individual parts helps not only with programming but with tasks in general; I wish I had learned this earlier on. When I first started working on the front-end I was completely overwhelmed, constantly overthinking what pages would be needed and what priority each page would have. This led me to complete much less than I could have.

6. Mini User Manual

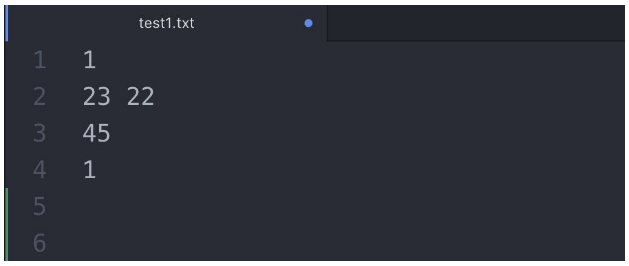
To run Auto-Grader

- Download the repository from:
    - https://github.com/virgiltataru/aagrader
- Start with running MongoDB database on the localhost using the following terminal commands.
    - `mongod`
    - `mongo --host localhost:27017`
    - The following link explains how to set up MongoDB on Mac.
        - https://treehouse.github.io/installation-guides/mac/mongo-mac.html
- Go in the 'aautograder' folder and launch the virtual environment using the following terminal command.
    - `source env/bin/activate`
    - 

```
(env) Muddassars-MacBook-Pro:aagrader muddassarsharif$ source env/bin/activate
```
- To run the API, go to the 'grader' folder to find api.py file and run the following terminal command in the virtual env.
    - `python api.py`
    - 

```
(env) Muddassars-MacBook-Pro:Grader muddassarsharif$ python api.py
```
- To use the API, download postman that can be found at:
    - https://www.getpostman.com/apps
    - The following API documentation explains how each endpoint can be used.
        - https://app.swaggerhub.com/apis-docs/ms8909/AutoGrader/1.0.0

The following diagram is an example of a test case submitted using a file.



```
1
23 22
45
1


```

The first line is the test-case-id. All the inputs are in the second line separated by a space. The third line should contain all the outputs also separated by space if there is more than one output. The fourth line should contain the time that the code is allowed to run, and this is to make sure that the faulty code does not keep running in Auto-Grader forever.