Phase 4: Testing and Refactoring

Note: *It is expected that all members of the group will make a reasonable contribution in writing parts of this report, that way one group member does not get overstretched. Furthermore, it ensures all members get practice of writing technical reports.*

Testing is a process of executing the program with the intent of finding errors. This process usually involves 4 different steps, i.e., 1. unit testing, 2. integration testing, 3. validation testing, and 4. system testing. Hence, you will have to test your program according to these steps. However, since we cannot test all at this time, we will do selective testing, i.e., selecting one or more classes for testing purposes.

1. Introduction

1.1 Purpose – Put here why you wrote this document including when, who was involved and contributed what - *contributions of each member listed individually to this phase of project and to this report, etc.*

The Testing and Refactoring document provides an overview of the testing process in our project, highlighting the unit testing in which we perform a class test and independent path test, validation testing and the refactoring of our code/design after performing such tests.

Ryan: My contribution to this project was working on the backend of the application which is the API. I also participated in writing several parts of the report. For this reason, I helped create the unit tests for the API and documented those unit tests through out the phase 4 document.

Elliot: My contribution to this project was working on the backend of the application which includes Configuring Twilio to post specific Flask endpoints, and Adding flask endpoint that processes Twilio Post. For phase 4, I helped document the API testing as well as creating the flow charts for the independent path testing.

Christina: My contribution to this project was working on the frontend of the application which includes, improving the home screen styling, creating a new view for messaging, and creating a new Flutter. For this reason, I helped test the UI which I also documented throughout the document.

1.2 Definitions, Acronyms or Abbreviations – Any definition, acronyms or abbreviations used in this document.

SMS: Short Message Service

1.3 References – Any references used in this document

- Jira: https://webcraft.atlassian.net/jira/software/projects/TEX/boards/1
- GitHub: https://github.com/elliotfayman/TextPy

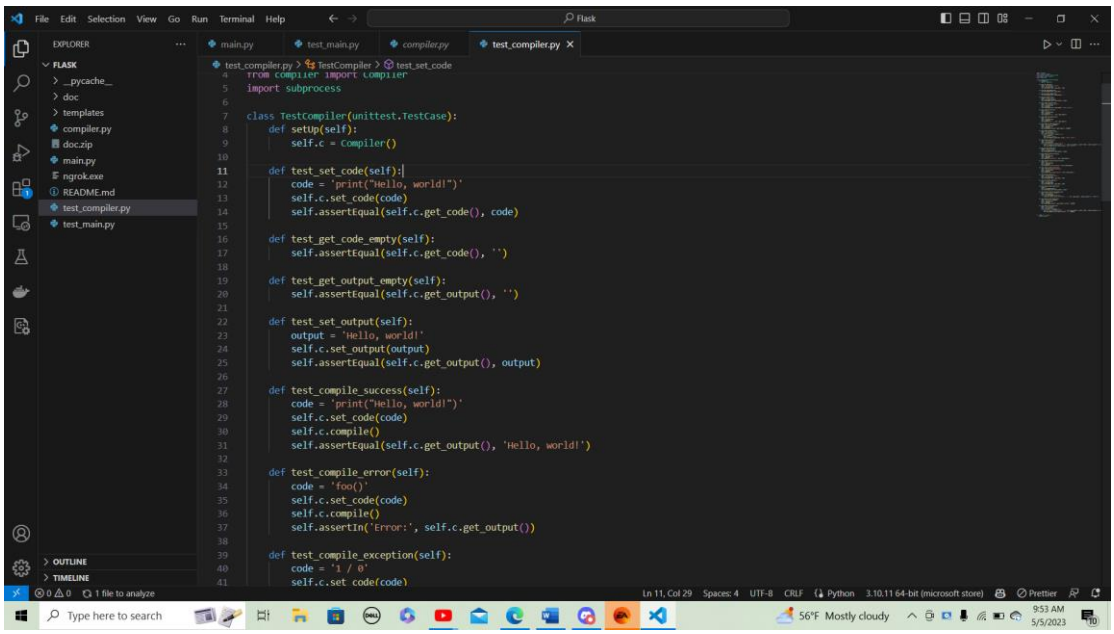
Note: If you are a group of four then split your group into two pairs. Each pair separately performs number 2 and 3 below. Compile the results in this report and submit.

2. Unit Testing – (Provide screen shots of testing results)

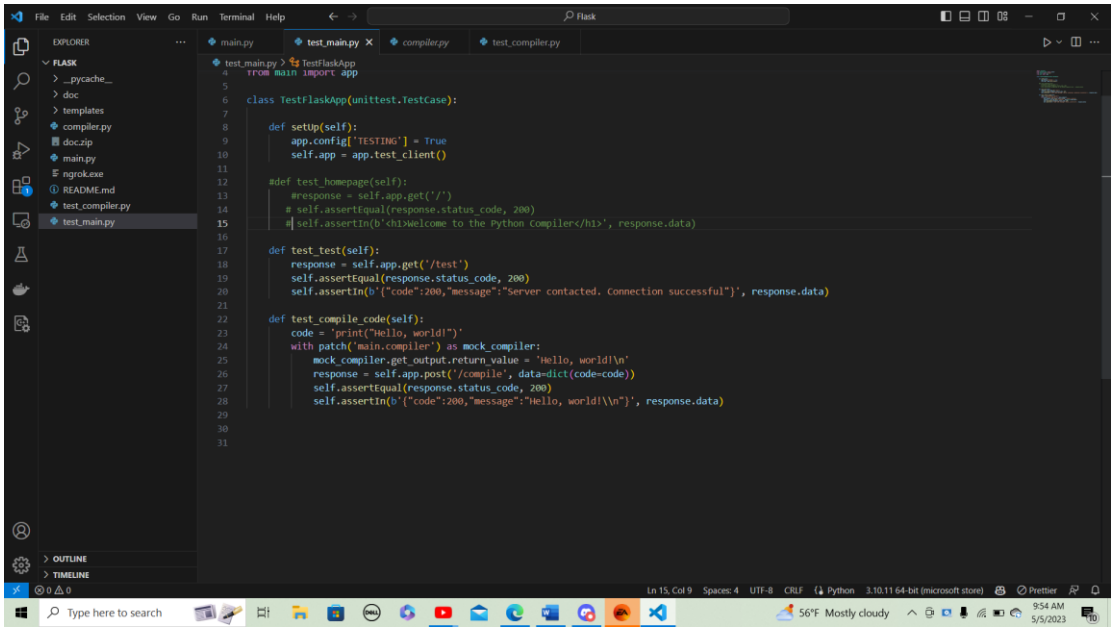
You should perform automated unit testing such as use Junit in JAVA, PyUnit/PyTest in Python, etc.

2.1 Class Testing – Each pair in the team chooses at least one class. Ideally select classes that have at least one method of size between 20 and 35 lines of executable code. Alternatively, you may select classes that you consider most critical. This should be easy to automate. Perform automated unit testing of at least two methods in each of the selected classes. If you find the methods difficult to test then refactor the code in a way so they are easier to test.

Compiler.py Class Testing



Main.py Class Testing



```python
from main import app

class TestFlaskApp(unittest.TestCase):

    def setUp(self):
        app.config['TESTING'] = True
        self.app = app.test_client()

    #def test_homepage(self):
        #response = self.app.get('/')
        # self.assertEqual(response.status_code, 200)
        # self.assertIn(b'<h1>Welcome to the Python Compiler</h1>', response.data)

    def test_test(self):
        response = self.app.get('/test')
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'{"code":200,"message":"Server contacted. Connection successful"}', response.data)

    def test_compile_code(self):
        code = 'print("Hello, world!")'
        with patch('main.compiler') as mock_compiler:
            mock_compiler.get_output.return_value = 'Hello, world!\n'
            response = self.app.post('/compile', data=dict(code=code))
            self.assertEqual(response.status_code, 200)
            self.assertIn(b'{"code":200,"message":"Hello, world!\\n"}', response.data)
```

Unit Testing Results

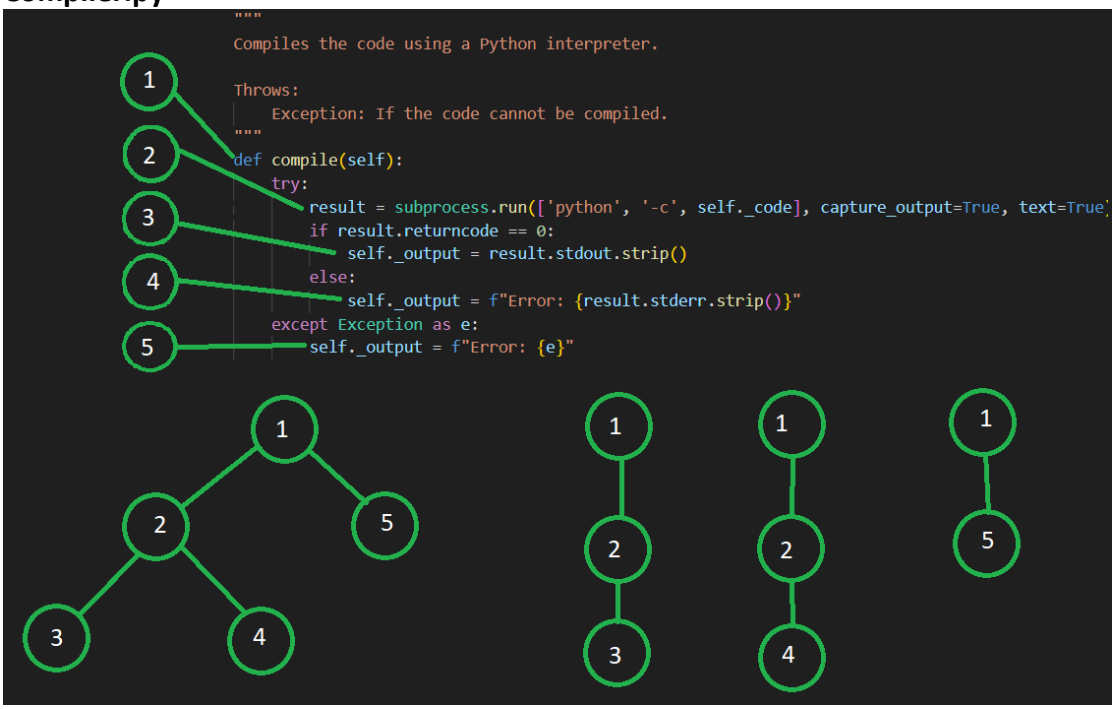
```
PS C:\Users\ellio\OneDrive\Desktop\School\Comp380\TextPy\Flask> python -m unittest
.....................
----------------------------------------------------------------------
Ran 22 tests in 1.763s

OK
PS C:\Users\ellio\OneDrive\Desktop\School\Comp380\TextPy\Flask>
```

2.2 Independent Path Testing

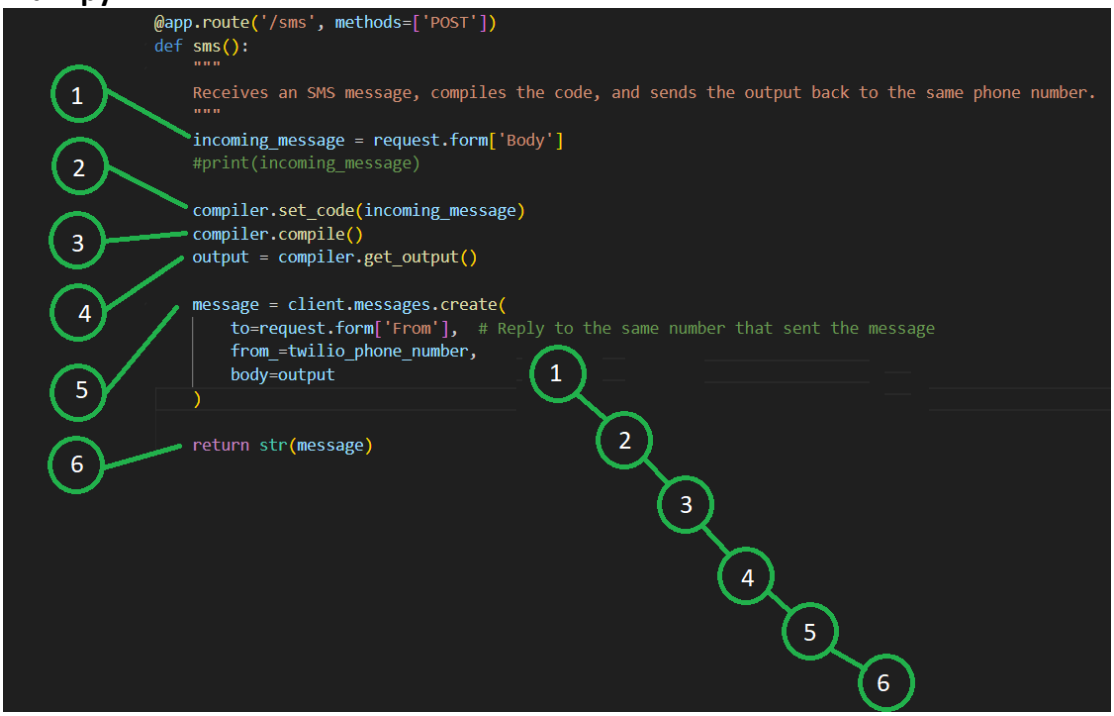
Each pair in the team draws the Flow Graph, determine the independent paths, prepare test cases and compare the results.

2.2.1 Flow Graph and Independent paths

Compiler.py



Main.py



```python
@app.route('/compile', methods=['POST'])
def compile_code():
    """
    Compiles the code received from the POST request using the Compiler class and returns the output as a JSON response.
    """
    code = request.form['code']
    compiler.set_code(code)
    compiler.compile()
    output = compiler.get_output()

    return jsonify({'code': 200, 'message': output})
```

2.2.2 Test cases and Test results – compare the test results with the expected values. If it is difficult to automate such tests then perform manual testing.
a) Test case b) Expected Value c) Test Results d) Conclusion: Passed/Not Passed

Compiler.py

1. Test Case #1
    a. Test Case: Test Set Code
    b. Expected Value: 'print("Hello, World!")'
    c. Test Results: The code instance variable was able to be changed to a different string value.
    d. Conclusion: **Passed**/Not Passed
2. Test Case #2
    a. Test Case: Test get Code Empty
    b. Expected Value: ''
    c. Test Results: When the value of code is not set, the return value of get code is the empty string.
    d. Conclusion: **Passed**/Not Passed
3. Test Case #3
    a. Test Case: Test get output Empty
    b. Expected Value: ''
    c. Test Results: When the compile method has not been called, the value of the output instance variable is the empty string.
    d. Conclusion: **Passed**/Not Passed
4. Test Case #4
    a. Test Case: Test set output
    b. Expected Value: 'Hello, world!'
    c. Test Results: The output instance variable is mutable when the set output function is called.
    d. Conclusion: **Passed**/Not Passed

5. Test Case #5
    a. Test Case: Test compile success
    b. Expected Value: ‘Hello World!’
    c. Test Results: Testing whether or not the built in python interpreter successfully compiles python code and mutates the output instance variable
    d. Conclusion: **Passed**/Not Passed
6. Test Case #6
    a. Test Case: Test compile Error
    b. Expected Value: ‘Error’
    c. Test Results: Testing whether compiler recognizes compiler errors in python code.
    d. Conclusion: **Passed**/Not Passed
7. Test Case #7
    a. Test Case: Test compile Exception
    b. Expected Value: ‘Error’
    c. Test Results: Testing whether or not compiler recognizes compiler exceptions when interpreting code.
    d. Conclusion: **Passed**/Not Passed

8. Test Case #8
    a. Test Case: Test compile large output
    b. Expected Value: ‘1000000
    c. Test Results: Testing whether compiler can process large integer inputs.
    d. Conclusion: **Passed**/Not Passed
9. Test Case #9
    a. Test Case: Test compile input
    b. Expected Value: ‘Hello World!’
    c. Test Results: Testing whether or not interpreter receives user input.
    d. Conclusion: **Passed**/Not Passed
10. Test Case #10
    a. Test Case: Test compile stdout
    b. Expected Value: ‘Hello World!
    c. Test Results: Testing whether compiler can successfully print to virtual console.
    d. Conclusion: **Passed**/Not Passed
11. Test Case #11
    a. Test Case: Test compile syntax error
    b. Expected Value: "SyntaxError"
    c. Test Results: The test checks whether the compiler can detect a syntax error in the code and return the correct error message.
    d. Conclusion: **Passed**/Not Passed

12. Test Case #12
    a. Test Case: Test compile divide by zero error
    b. Expected Value: "ZeroDivisionError"
    c. Test Results: The test checks whether the compiler can detect a divide by zero error in the code and return the correct error message.
    d. Conclusion: **Passed**/Not Passed

13. Test Case #13
    a. Test Case: Test compile import error
    b. Expected Value: "ModuleNotFoundError"
    c. Test Results: The test checks whether the compiler can detect an import error in the code and return the correct error message.
    d. Conclusion: **Passed**/Not Passed
14. Test Case #14
    a. Test Case: Test set and get code
    b. Expected Value: The code returned by the get_code method should be the same as the code passed to the set_code method.
    c. Test Results: The test checks whether the compiler object can store and retrieve the code correctly.
    d. Conclusion: **Passed**/Not Passed

15. Test Case #15
    a. Test Case: Test set and get output empty
    b. Expected Value: ‘’
    c. Test Results: Set an empty output, get the output and check that it is the same.
    d. Conclusion: **Passed**/Not Passed

16. Test Case #16
    a. Test Case: Test set and get output empty
    b. Expected Value: ‘’
    c. Test Results: Set an empty output, get the output and check that it is the same.
    d. Conclusion: **Passed**/Not Passed

Main.py

1. Test Case #1
    a. Test Case: Test Flask homepage response status code and content
    b. Expected Value: ‘Response status code 200 and content with the string '<h1>Welcome to the Python Compiler</h1>'
    c. Test Results: Test whether the Flask app homepage is returning the expected response status code and content.
    d. Conclusion: **Passed**/Not Passed

2. Test Case #2
    a. Test Case: Test Flask test endpoint response status
    b. Expected Value: Response status code 200 and content with the string '{"code":200,"message":"Server contacted. Connection successful"}'
    c. Test Results: Test whether the Flask app test endpoint is returning the expected response status code and content.
    d. Conclusion: **Passed**/Not Passed
3. Test Case #3
    a. Test Case: Test Flask compile endpoint response status code and content
    b. Expected Value: Response status code 200 and content with the string '{"code":200,"message":"Hello, world!\n"}'
    c. Test Results: Response status code 200 and content with the string '{"code":200,"message":"Hello, world!\n"}'
    d. Conclusion: **Passed**/Not Passed

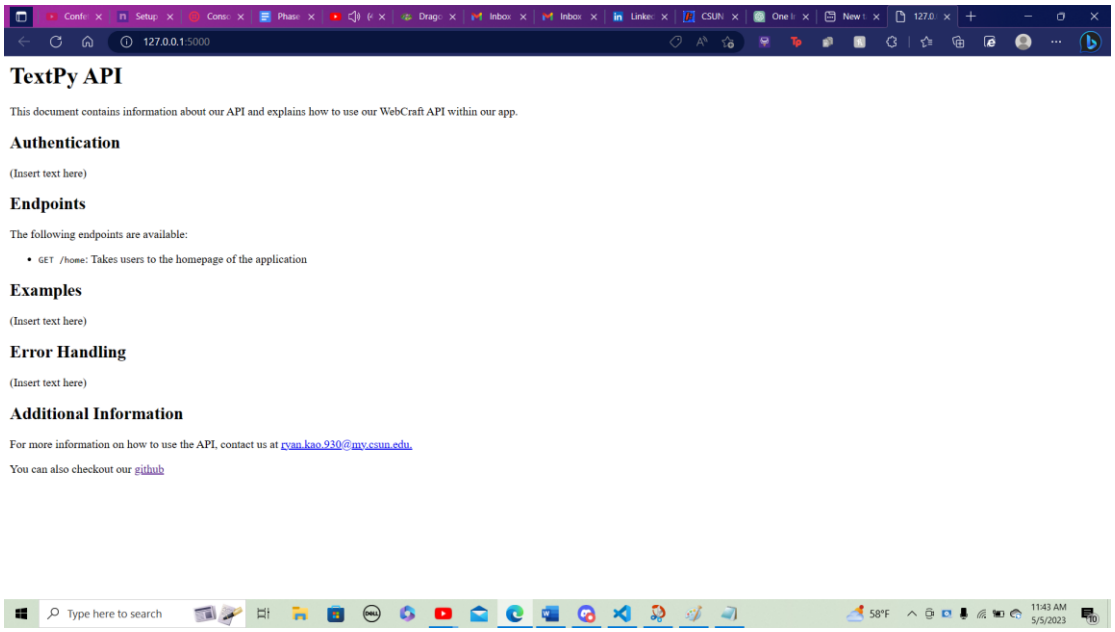
3. Validation Testing

Each pair in the team chooses at least one requirement from Requirement Analysis and performs validation tests. Provide appropriate screen shots of GUI to show the test being carried out.

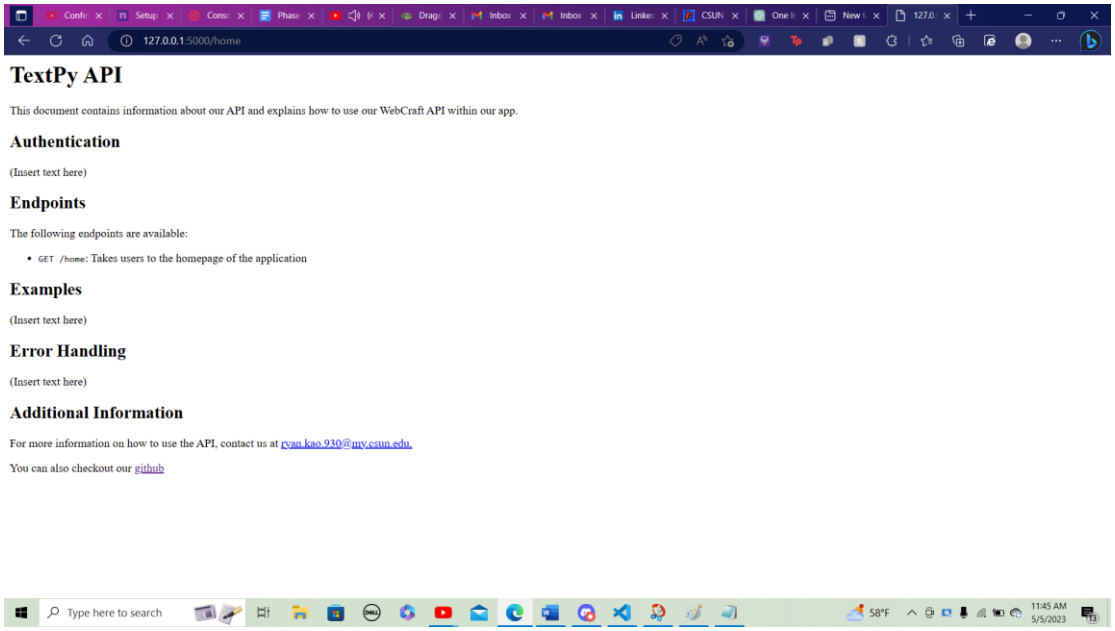
3.1 Requirement name

Requirement: User must be taken to homepage when the base URL or the /home URL is entered into the browser

Root URL

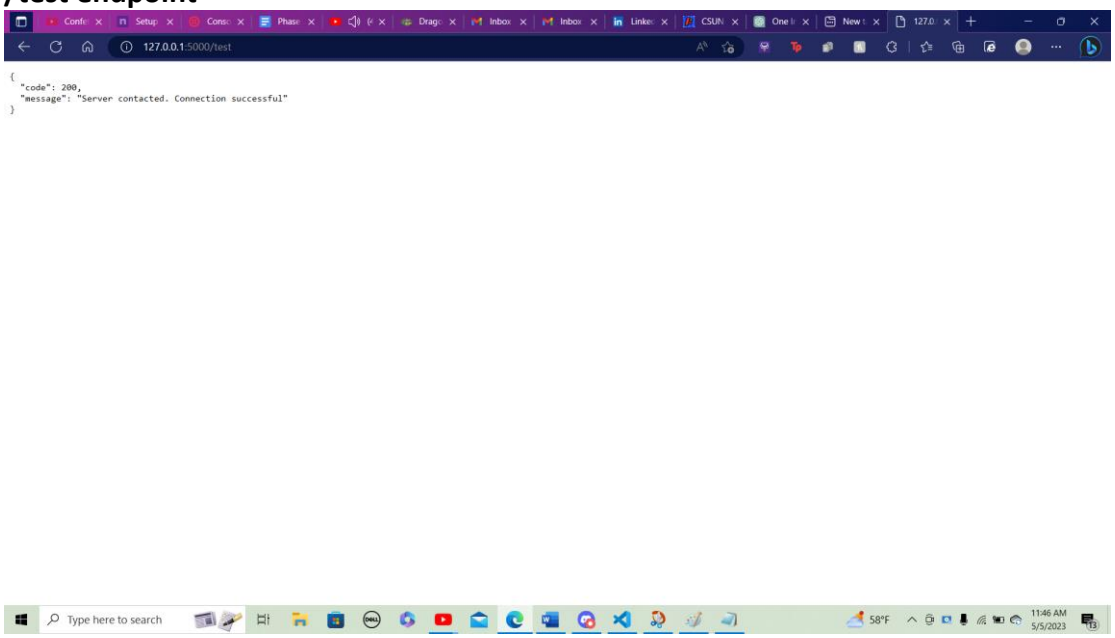


/home endpoint



Requirement: User must be able to ping API using the test endpoint. User must receive a successful connection confirmation response.

/test endpoint



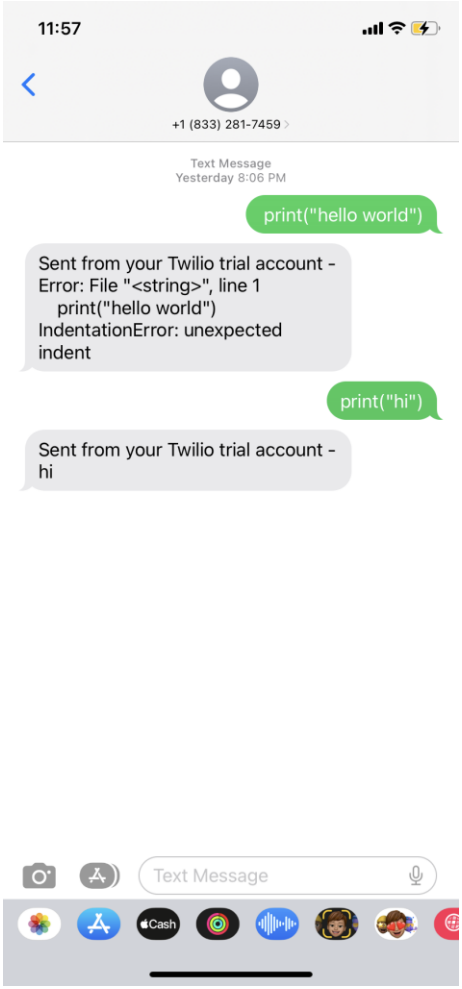
Requirement: User must be able to ping API and compile code via HTTP POST request.

/compile endpoint

```
PS C:\Users\ellio\OneDrive\Desktop\School\Comp380\TextPy\Flask> Invoke-WebRequest -Uri http://localhost:5000/compile -Method POST -Body @{code='print("Hello, world!
")'}


StatusCode        : 200
StatusDescription : OK
Content           : {
                      "code": 200,
                      "message": "Hello, world!"
                    }

RawContent        : HTTP/1.1 200 OK
                    Connection: close
                    Content-Length: 48
                    Content-Type: application/json
                    Date: Fri, 05 May 2023 18:54:00 GMT
                    Server: Werkzeug/2.2.2 Python/3.10.11

                      "code": 200,
                      "message": "Hel...
Forms             : {}
Headers           : {[Connection, close], [Content-Length, 48], [Content-Type, application/json], [Date, Fri, 05 May 2023 18:54:00 GMT]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 48
```

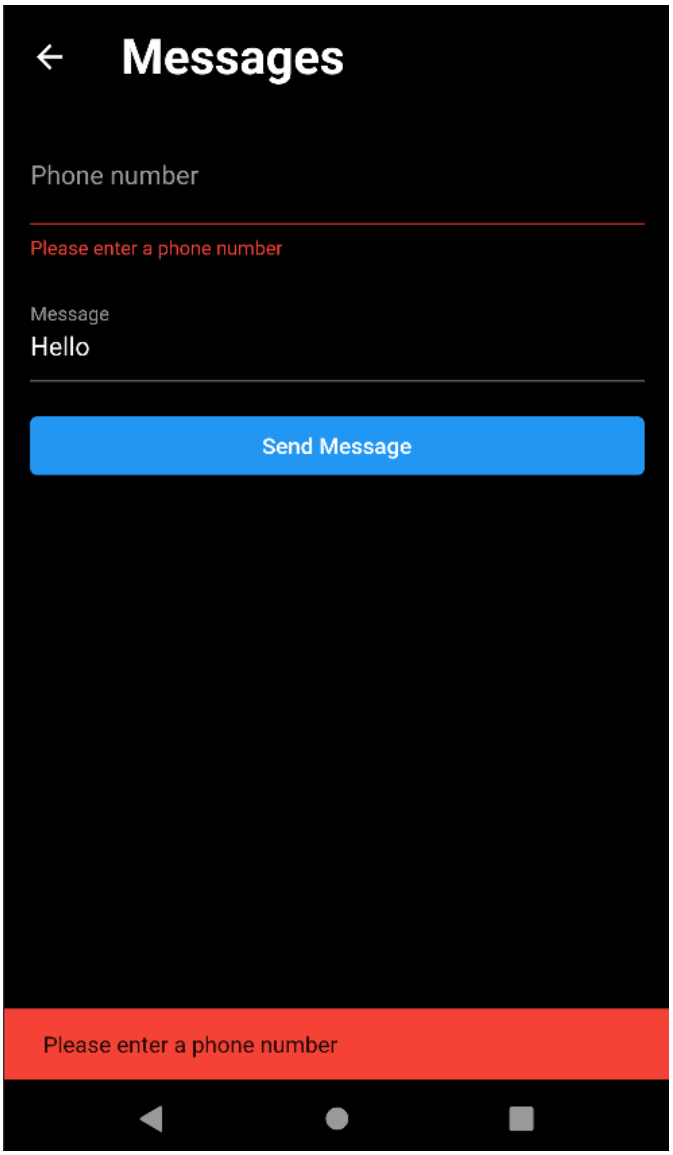
Requirement: User must be able to ping API and compile code via SMS.

/SMS endpoint



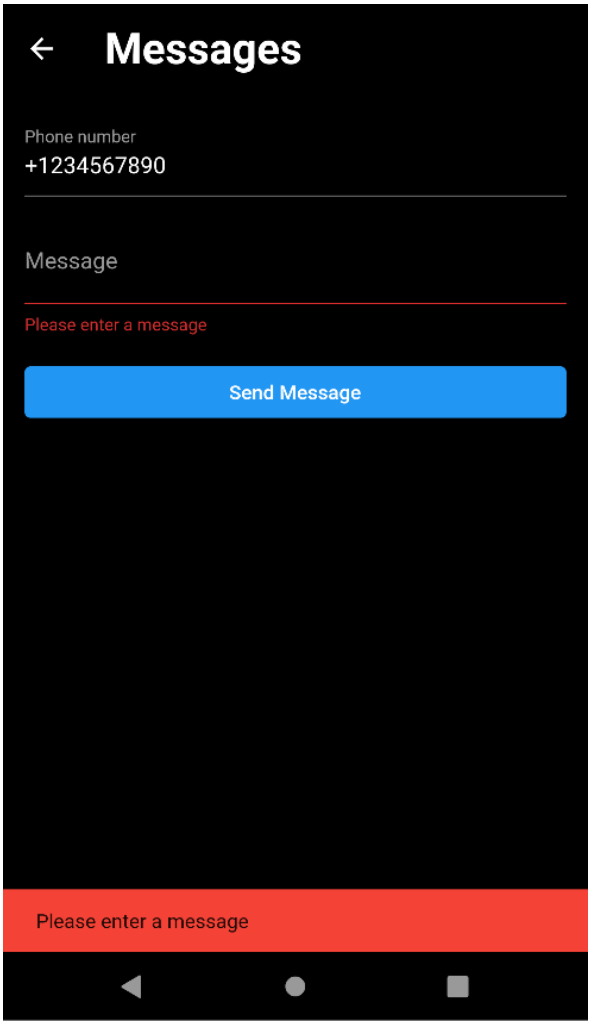
Requirement: User must submit a phone number in send message screen.

Form Class



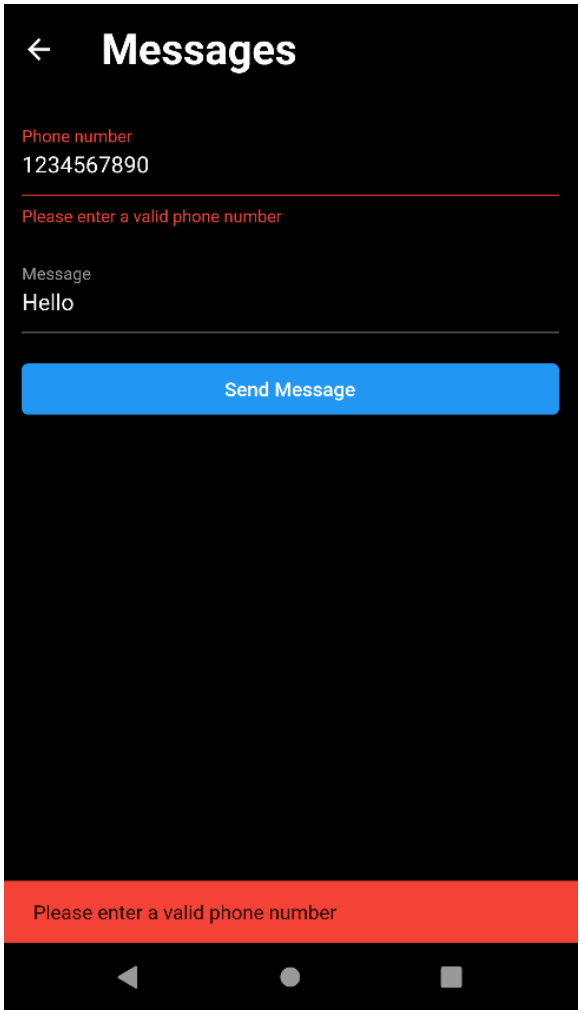
Requirement: User must submit a message text in send message screen.

Form Class



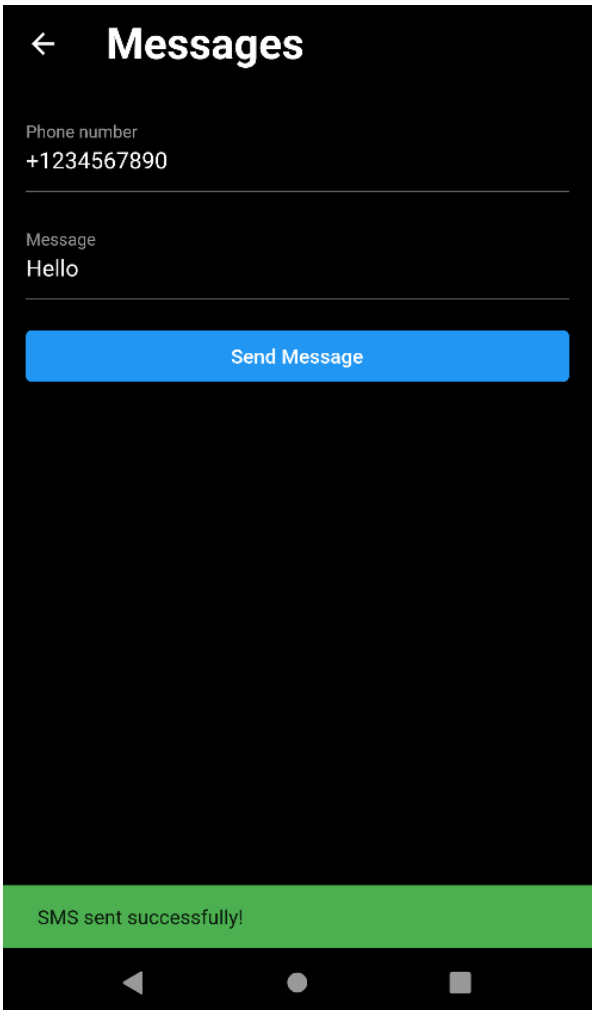
Requirement: User must submit a valid phone number in send message screen.

Form Class



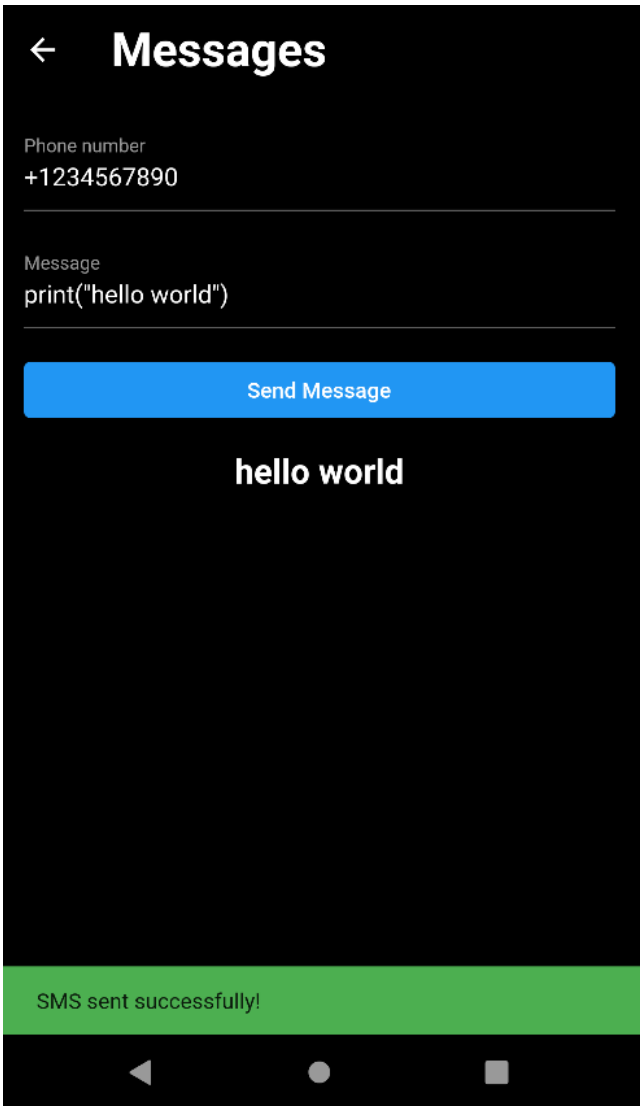
Requirement: User prompted with valid message text when all fields in the form are filled out correctly.

Form Class



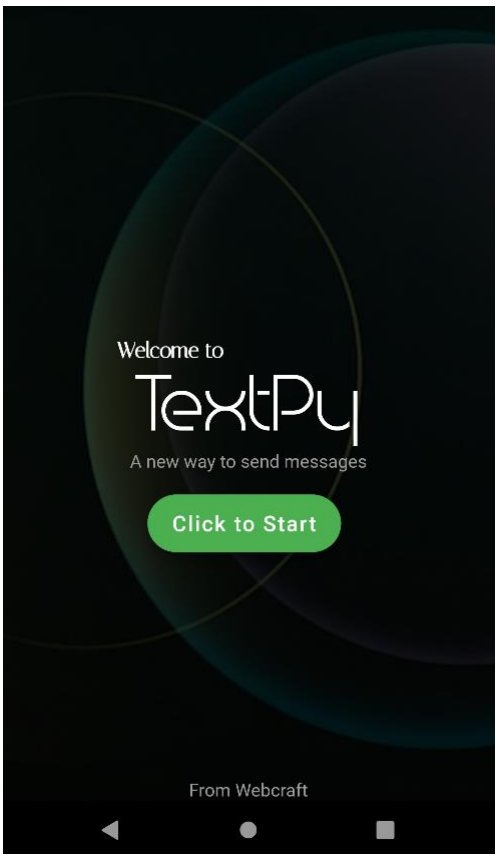
Requirement: User prompted with return message when application receives sms message from the same number.

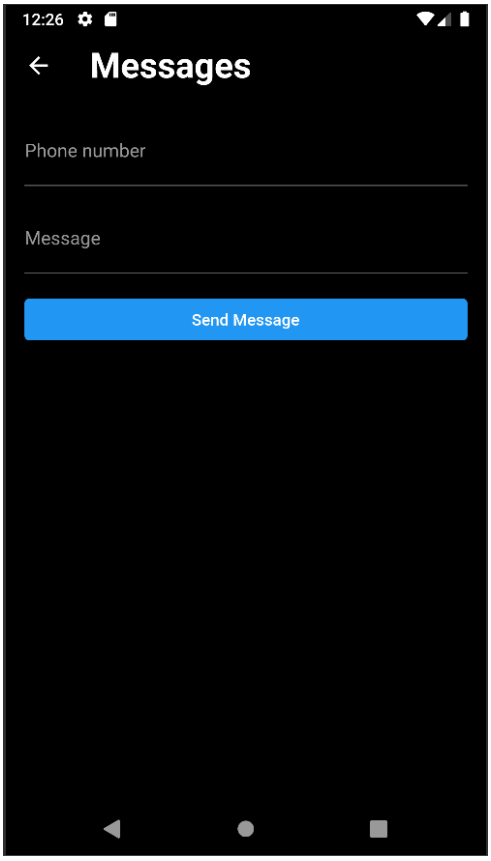
Form Class



Requirement: User can press home button to go to the texting screen.

HomeScreen Class





3.2 ECs (Equivalence classes)

1. Main.py Class
    a. Valid Partition: User enters a valid endpoint (/, /home, /test, /compile). User is connected to the internet.
    b. Invalid Partition: User enters a endpoint not listed in the above list. User is not connected to the internet. User adds additional parameters to the URL.
2. Form.dart Class
    a. Valid Partition: User enters text inside both the message and the phone number text fields. The phone number is also a valid phone number written in E.164 standard format.
    b. Invalid Partition: User Leaves either the phone number blank or the message blank. User enters a invalid phone number. Either the number is not a registered number or it is not in E.164 format.

1.3 Test Cases and Test Results – same as above

1. Main.py Class
   a. Test Case #1
      i. Test Case: Render proper endpoint/JSON response when valid URL is typed.
      ii. Expected Value: The Page should render either the home page html or a JSON response.
      iii. Test Results: The respective html/JSON response was rendered.
      iv. Conclusion: **Passed**/Not Passed
   b. Test Case #2
      i. Test Case: 'Not Found' Error should be thrown when invalid URL is typed.
      ii. Expected Value: The Page should render the 'Not Found' template.
      iii. Test Results: The 'Not Found' template was rendered.
      iv. Conclusion: **Passed**/Not Passed
2. Form.dart Class
   a. Test Case #1
      i. Test Case: Phone Number Blank Test
      ii. Expected Value: Pop up error shown to user prompting them to enter phone number in the phone number field.
      iii. Test Result: The user is prompted with a red strip error message when user does not enter a phone number prompting them to enter a phone number.
      iv. Conclusion: **Passed**/Not Passed
   b. Test Case #2
      i. Test Case Phone Number Invalid Test
      ii. Expected Value: Pop up error shown to user prompting them to enter valid phone number in phone number field.
      iii. Test Results: The user is prompted with a red strip error message when the user does not enter a valid phone number prompting them to enter a valid phone number.
      iv. Conclusion: **Passed**/Not Passed
   c. Test Case #3
      i. Test Case Message Blank Test
      ii. Expected Value: Pop up error shown to user prompting them to enter message in the message field.
      iii. Test Results: The user is prompted with a red strip error message when user does not enter a message prompting them to enter a message.
      iv. Conclusion: **Passed**/Not Passed
   d. Test Case #4
      i. Test Case Message and Phone Number Valid Test
      ii. Expected Value: Pop up Success message shown.
      iii. Test Results: The user presses the send button, and the user receives a green bar success pop up.
      iv. Conclusion: **Passed**/Not Passed

4. Refactoring

Do you recall performing any refactoring to your code/design? If so then give a brief description of the instances along with the reasons.

There was not much refactoring done other than some minor changes to improve code readability and documentation. These changes were made to ensure that the code was more maintainable and easier to understand for other developers who might work on the project in the future.