CSC3065 Cloud Computing

Assessment 2: QUBGradeMe Submission

Student Number: 40266405

Name: Michal Guzy

Task A: Functions

Function One

Function: Total Marks

Repository URL: http://gitlab.hal.davecutting.uk/40266405/qubgrademe-totalmarks
Live Service URL: http://qubgrademe-totalmarks.40266405.qpc.hal.davecutting.uk/

Description of Implementation:

The total marks microservice has been implemented using Python Flask with Python version 3.9, it has been deployed on QPC Rancher as a PaaS (Platform as a Service). It uses the Flask framework to implement the endpoints and expose the microservice onto the web. The microservice is comprised of two files, one named “flaskApp.py” and another named “configureRoutes.py”.

The first file, “flaskApp.py” initialises the microservice, imports the routes from the other file and runs the microservice on the host machine with port “0.0.0.0”. This ensures that the microservice is seen from a docker container.

The “configureRoutes.py” file includes all the endpoints and any necessary logic to calculate the total marks. There are two endpoints in this file, one of them welcomes the user with a message when they call the URL + “/” and the other endpoint calculates the total mark when the URL + “/totalmarks” is called. Parameters also need to be added onto the URL in the form of “mark1”, “mark2” and so on to fulfil the request. An example of a working URL to get a total mark of 350:

http://qubgrademe-totalmarks.40266405.qpc.hal.davecutting.uk/totalmarks?mark1=70&mark2=70&mark3=70&mark4=70&mark5=70

The first function which welcomes the user only returns a response code of 200. But the second function of total marks also validates the input to ensure that the marks are correct before completing a calculation. If the marks are invalid, the total marks function will return a bad request error with code 400. Figure 1 shows how input validation ensures that the mark is not null, ensures that the mark is greater than zero and less than 100, and ensures that the mark is not an integer.

```python
# Looping through the array and calculating the total marks
for i in range(0,5):
    # Checking if the marks are integers
    try:
        moduleArray[i] = int(moduleArray[i].strip())
    except:
        response = make_response(jsonify({'message':"Mark " + str(i+1)  + " needs to be an integer"}))
        response.headers.add("Access-Control-Allow-Origin", "*")
        return response, 400

    # If the module is less than zero it return that it is invalid
    if moduleArray[i] <= 0:
        response = make_response(jsonify({'message': "Mark " + str(i+1) + " needs to be greater than zero"}))
        response.headers.add("Access-Control-Allow-Origin", "*")
        return response, 400

    # If the modules is greater than one hunedred it is invalid
    if moduleArray[i] > 100:
        response = make_response(jsonify({'message': "Mark " + str(i+1) + " needs to be less than one hundred"}))
        response.headers.add("Access-Control-Allow-Origin", "*")
        return response, 400

    # Add up the total marks
    if moduleArray[i] >= 0 and moduleArray[i] <= 100:
        totalMarks = totalMarks + moduleArray[i]

# Return the response to the user
strmessage = 'Your total marks are: ' + str(totalMarks)
response = make_response(jsonify({'message': strmessage}))
response.headers.add("Access-Control-Allow-Origin", "*")
return response, 200
```

Figure 1: Total Marks Validation Checking

The parameters are retrieved as strings and are then parsed into integers in the try except block at the top of figure 1. If a mark is not an integer, it will return bad request with a message of “Mark needs to be an integer”. Finally, an “Access-Control-Allow-Origin” header is added to the response so that chrome does not throw a CORS validation error.

The microservice is containerised using docker and uses the “python:3.9” docker image. As seen in figure 2, the Dockerfile copies over both the “flaskApp.py” and the “configureRoutes.py” files onto the image so that they are accessible. It also installs the necessary packages using “pip install <package name>” so that the docker image can access the necessary packages. Packages that were necessary for this microservice include flask for the framework and PyTest for unit testing. It then exposes port 5000 and sets the entry point to be “python” “flaskApp.py” (this command will run the Flask server).

```dockerfile
FROM python:3.9
WORKDIR /src
COPY ["flaskApp.py", "/src"]
COPY ["configureRoutes.py", "/src"]
RUN pip install flask
RUN pip install pytest
RUN pip install pytest-flask
RUN pip install flask_cors
EXPOSE 5000
ENTRYPOINT [ "python" ]
CMD [ "flaskApp.py" ]
```

Figure 2: Total Marks Dockerfile

Finally, the microservice is deployed on Queen’s Private Cloud (QPC) Rancher using the image created from the Dockerfile in figure 2. It has a workload to run the code, and an ingress with port 5000 to expose the microservice to the web.

Description of Testing:

Testing the microservice was first done manually by calling the microservice in Postman and seeing if it returns the correct results as seen in figure 3.

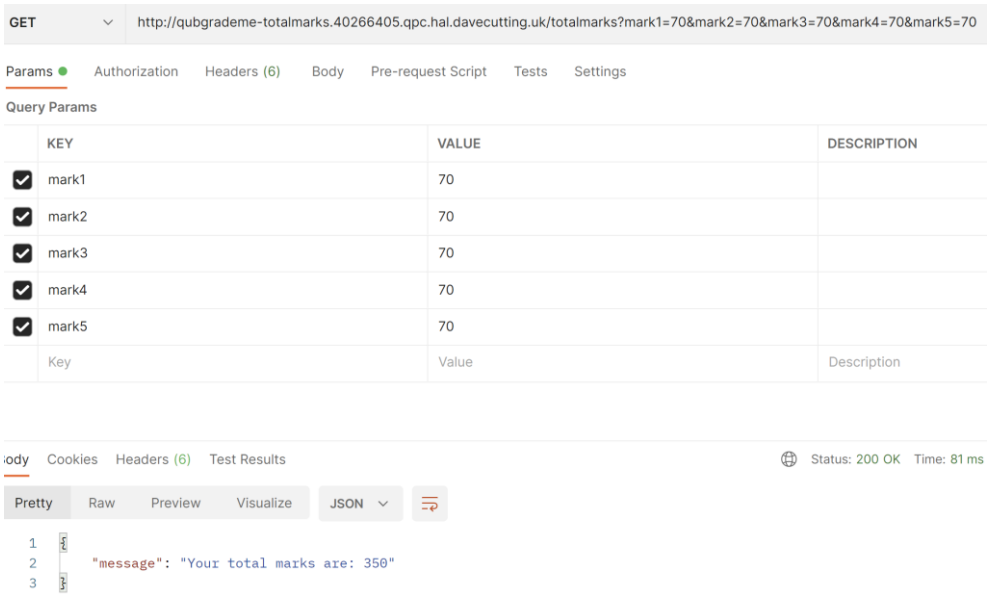
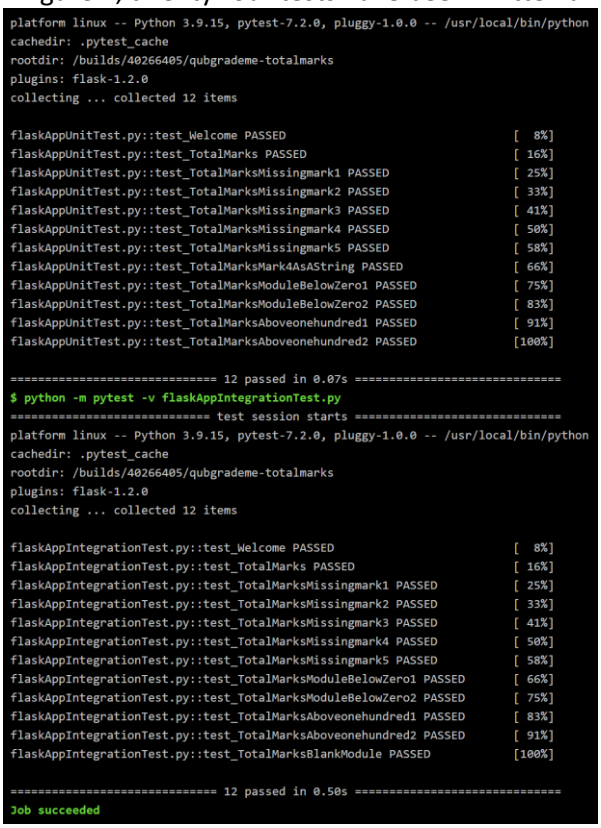


Figure 3: Manual Postman Testing of Total Marks

Once the manual postman testing was completed, a "gitlab-ci.yml" (Continuous Integration) file was written to tell the GitLab runner how to test the microservice. Unit tests were written using PyTest to test the application, and these unit tests were tested on the GitLab QPC runner and all tests pass. Once the application was deployed to QPC Rancher, integration (HTTP) tests were able to be performed to check that the deployed microservice was returning the correct data, and the correct response codes. As seen in figure 4, twenty-four tests have been written and they all pass.



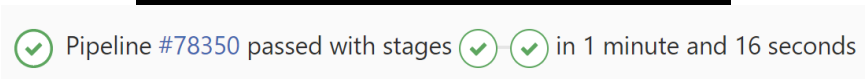


Figure 4: CI/CD results for total marks

The tests for this microservice test if the function returns the correct values if good data is passed in, and test if the function does the right thing if bad data is passed in. Figure five shows an example of a good and bad data unit test and a good and bad data HTTP test. The image on the left-hand side is a unit test, and the image on the right-hand side is a HTTP test.

```python
# Testing total marks with valid data
def test_TotalMarks(client):
    url = '/totalmarks?mark1=75&mark2=75&mark3=75&mark4=57&mark5=87'
    print("Testing total marks")
    response = client.get(url)
    assert response.get_data() == b'{"message":"Your total marks are: 369"}\n'
    assert response.status_code == 200

# Testing the total marks with missing module 1
def test_TotalMarksMissingmark1(client):
    url = '/totalmarks?mark2=75&mark3=75&mark4=75&mark5=75'
    print("Testing total marks")
    response = client.get(url)
    assert response.get_data() == b'{"message":"Mark 1 needs to be greater than zero"}\n'
    assert response.status_code == 400
```

```python
# Testing total marks with valid data
def test_TotalMarks():
    url = '/totalmarks?mark1=75&mark2=75&mark3=75&mark4=57&mark5=87'
    response = get(URLFORTOTALMARKS + url)
    assert response.text == '{"message":"Your total marks are: 369"}\n'
    assert response.status_code == 200

# Testing the total marks with missing module 1
def test_TotalMarksMissingmark1():
    url = '/totalmarks?mark2=75&mark3=75&mark4=75&mark5=75'
    response = get(URLFORTOTALMARKS + url)
    assert response.text == '{"message":"Mark 1 needs to be greater than zero"}\n'
    assert response.status_code == 400
```

Figure 5: Unit and HTTP Testing of Total Marks

Anything else to highlight:

Continuous deployment has also been implemented into the total marks microservice by adding a deploy job into the CI/CD file (".gitlab-ci.yml"). As seen in figure 6, the deploy job uses the "docker:latest" docker image to build the image, login to the container registry and then push that image to the container registry. The "dind" tag specifies to use docker in docker as this is the only way to push to the queen's QPC docker registry.

```yaml
# Uses the python docker image
image: python:3.9

# Test the project
test:
  script:
    - python --version  # For debugging
    - pip install flask
    - pip install pytest
    - pip install pytest-flask
    - pip install flask_cors
    - pip install requests
    - python -m pytest -v flaskAppUnitTest.py
    - python -m pytest -v flaskAppIntegrationTest.py

# Deploy the project
deploy:
  image: docker:latest
  stage: deploy
  tags:
    - "dind"
  script:
    - docker build -t registry.hal.davecutting.uk/40266405/qubgrademe-totalmarks .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker push registry.hal.davecutting.uk/40266405/qubgrademe-totalmarks
```

Figure 6: Total Marks CI/CD File

Function Two

Function: Classification of a grade

Repository URL: http://gitlab.hal.davecutting.uk/40266405/qubgrademe-classifygrade
Live Service URL: https://qubgrademe-classifygrade.azurewebsites.net/api/qubgrademe-classifygrade?mark1=90&mark2=90&mark3=90&mark4=90&mark5=90

Description of Implementation:

The classify grade microservice has been deployed to Microsoft Azure as an Azure Python function written using Python version 3.9. This microservice uses the FaaS (Function as a Service) paradigm and is not containerised like the rest of the microservices in this system. Due to this microservice being FaaS, Azure takes care of the deployment and a Dockerfile is not necessary.

The folder structure consists of an “src” folder which holds all source code for this microservice, inside the “src” folder is an automatically generated folder structure for a Python Azure function. This includes the “function.json” which points to which script file should be ran on start and the “qubgrademe_classifygrade_function.py” holds all the necessary logic to fulfil a request. Another important file in this folder structure is the “requirements.txt” file which holds all the necessary dependencies that the Python script uses.

In the “requirements.txt” file there is only one dependency which is the “azure-functions” library. The “qubgrademe_classifygrade_function.py” file is where the endpoint is declared. It is declared in the format of URL + “api/qubgrademe-classifygrade” and onto the end of the URL parameters need to be added so that the function can carry out the necessary calculations. These parameters are in the format of “?mark1=<mark>&mark2=<mark>” and so on until mark 5.

The function uses logging so that the system administrator can log in to the Azure Portal and see exactly what is happening within the application. An example of one of these log commands can be seen in figure 7. When the endpoint is hit by the user, this log will tell the system admin that the function is running.

```
def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function for classifying a grade is processing a request.')
```

Figure 7: Logging Command in Python Azure Function

The function firstly appends all the marks as strings to an array and validates the marks to ensure they are correct. As seen in figure 8, it is checked to see if a mark is an integer and to see if the mark is between 0 and 100 (Checking if it is an integer also catches the module being an empty string or null). If a mark is invalid with any of these reasons, a personalised error message in the form of a HTTP Bad Request (error 400) is returned to the user.

```
# Looping through and calculating the average
logging.info('Looping through the marks to validate them.')
for i in range(0,5):
    try:
        moduleArray[i] = int(moduleArray[i].strip())
    except:
        logging.info('The marks are invalid.')
        return func.HttpResponse(
        "Missing module or invalid data for mark " + str(i+1),
        status_code=400
        )

    # If the module is less than zero it return that it is invalid
    if moduleArray[i] <= 0:
        logging.info('A mark is less than or equal to 0.')
        return func.HttpResponse(
        "Mark " + str(i+1) + " needs to be greater than zero",
        status_code=400
        )

    # If the modules is greater than one hunedred it is invalid
    if moduleArray[i] > 100:
        logging.info('A mark is greater than 100.')
        return func.HttpResponse(
        "Mark " + str(i+1) + " needs to be less than one hundred",
        status_code=400
        )

    logging.info('The marks are being added up.')
    # Add up the total marks
    if moduleArray[i] >= 0 and moduleArray[i] <= 100:
        sumOfModules = sumOfModules + moduleArray[i]
```

Figure 8: Validating a Mark

If the marks are all valid, they are then added up to get the sum of modules. The average mark is then calculated and as seen in figure 9 there is a collection of if statements to decide which classification is returned dependent on the marks given.

```
# 1st and above 70%
# 2:1 60-69%
# 2:2 50-59%
# Third class 40-49%
# Pass less than 40%
# Classifying the grade

classification = ""
if average >= 70:
    classification = "First Class"
elif average >= 60:
    classification = "2:1"
elif average >= 50:
    classification = "2:2"
elif average >= 40:
    classification = "Third Class"
else:
    classification = "Pass"

logging.info('The marks were all valid, returning  a message to the user with their classification.')
return func.HttpResponse(
    '{"Classification" :' + '"' + classification + '"}',
    status_code=200
)
```

Figure 9: Deciding the Classification of Marks

Description of Testing:

Testing of the microservice was done by running it locally and checking in Postman that for all given valid and invalid marks the function is returning the correct data. Figure 10 shows an example of one of these tests returning the correct data.

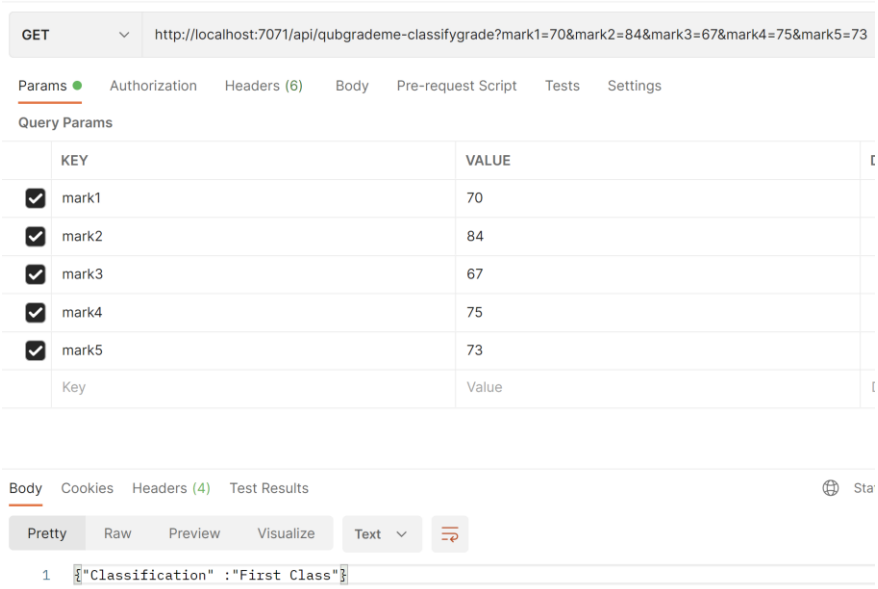


Figure 10: Postman Test for the Classification Microservice

After the manual testing was completed, a “gitlab-ci.yml” file was written to add CI (Continuous Integration) testing, this CI testing was comprised of unit tests at the start, but later HTTP (Integration) tests were also added to check that the deployed Azure function was returning the correct data. Figure 11 shows the “.gitlab-ci.yml” file which uses the “python:3.9” docker image to test the application. The CI tests are run on the QPC GitLab runner, it installs the necessary dependencies using pip install and then runs both the unit and integration test scripts.

```
# Uses the python docker image
image: python:3.9

# Install the necessary packages
before_script:
  - python --version  # For debugging
  - pip install pytest
  - pip install azure.functions
  - pip install requests

# Test the project
test:
  script:
    - python -m pytest -v ./src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py
    - python -m pytest -v ./src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py
```

Figure 11: CI File for the Classify Grade Microservice

These scripts have 13 tests each and all pass as seen in figure 12.

```
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_main PASSED [  7%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_main21 PASSED [ 15%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_main22 PASSED [ 23%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainInvalidBlankMark1 PASSED [ 30%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainInvalidMissingmark1 PASSED [ 38%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainInvalidMissingmark2 PASSED [ 46%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainInvalidMissingmark3 PASSED [ 53%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainInvalidMissingmark4 PASSED [ 61%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainInvalidMissingmark5 PASSED [ 69%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainInvalidStringmark1 PASSED [ 76%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainInvalidStringmark3 PASSED [ 84%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainPass PASSED [ 92%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-test.py::TestQubGradeMeClassifyGrade::test_mainThirdClass PASSED [100%]

============================== 13 passed in 0.13s ==============================
$ python -m pytest -v ./src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py
============================= test session starts ==============================
platform linux -- Python 3.9.15, pytest-7.2.0, pluggy-1.0.0 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /builds/40266405/qubgrademe-classifygrade
collecting ... collected 13 items

src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_main PASSED [  7%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_main21 PASSED [ 15%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_main22 PASSED [ 23%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainStringInmark1 PASSED [ 30%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainStringInmark1Missing PASSED [ 38%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainStringInmark2Missing PASSED [ 46%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainStringInmark3Missing PASSED [ 53%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainStringInmark4 PASSED [ 61%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainStringInmark4Missing PASSED [ 69%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainStringInmark5Blank PASSED [ 76%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainStringInmark5Missing PASSED [ 84%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainpass PASSED [ 92%]
src/qubgrademe-classifygrade/tests/qubgrademe-classifygrade-integrationtest.py::TestQubGradeMeClassifyGradeIntegrationTest::test_mainthirdclass PASSED [100%]

============================== 13 passed in 2.82s ==============================
Saving cache for successful job
Creating cache default...
WARNING: .cache/pip: no matching files
WARNING: venv/: no matching files
Archive is up to date!
Created cache
Job succeeded
```

Pipeline #78413 passed with stage in 37 seconds

Figure 12: CI/CD Results for Classify Grade

The unit and HTTP tests have been written using PyTest and are ran using the script in the CI file. These files test both valid and invalid data, all grade classifications are tested, missing modules/marks as strings are also tested. Response codes are checked to ensure that the correct code is returned for valid and invalid data.

An example of valid/invalid unit and HTTP tests using PyTest is shown in figure 13. On the left-hand side, the unit test first checks to see if a classification is a “pass”, and the second test checks to see if the correct error message is displayed when invalid data is passed through. On the other side, the HTTP test first checks to see if a classification is a “pass” and then a few more invalid tests are shown. (Not all tests are shown in the screenshot)

```python
# Test main with valid data for a Pass
def test_mainPass(self):
    # Arrange
    request = func.HttpRequest(
        method='GET',
        url='/api/qubgrademe-classifygrade',
        params={ 'mark1': '39', 'mark2': '39', 'mark3': '39', 'mark4': '39', 'mark5': '39'  },
        body=None
    )
    # Act
    response = main(request)
    # Assert
    # Assert we have a success code
    assert response.status_code == 200
    # Assert the response is as expected
    assert 'Pass' in response.get_body().decode()

# Test main with a string as one of the modules
def test_mainInvalidStringmark1(self):
    # Arrange
    request = func.HttpRequest(
        method='GET',
        url='/api/qubgrademe-classifygrade',
        params={ 'mark1': 'ald', 'mark2': '39', 'mark3': '39', 'mark4': '39', 'mark5': '39'  },
        body=None
    )
    # Act
    response = main(request)
    # Assert
    # Assert we have a success code
    assert response.status_code == 400
    # Assert the response is as expected
    assert 'Missing module or invalid data for mark 1' in response.get_body().decode()
```

```python
    def test_mainpass(self):
        url = '?mark1=35&mark2=37&mark3=25&mark4=35&mark5=37'
        response = get(self.URLFORCLASSIFYGRADE + url)
        assert "Pass" in response.text
        assert response.status_code == 200

    # Test main with a string as one of the modules
    def test_mainStringInmark1(self):
        url = '?mark1=al&mark2=37&mark3=25&mark4=35&mark5=37'
        response = get(self.URLFORCLASSIFYGRADE + url)
        assert "Missing module or invalid data for mark 1" in response.text
        assert response.status_code == 400

    # Test main with a string as one of the modules
    def test_mainStringInmark4(self):
        url = '?mark1=78&mark2=37&mark3=25&mark4=po&mark5=37'
        response = get(self.URLFORCLASSIFYGRADE + url)
        assert "Missing module or invalid data for mark 4" in response.text
        assert response.status_code == 400

    # Test main with mark 1 missing
    def test_mainStringInmark1Missing(self):
        url = '?&mark2=37&mark3=25&mark4=78&mark5=37'
        response = get(self.URLFORCLASSIFYGRADE + url)
        assert "Missing module or invalid data for mark 1" in response.text
        assert response.status_code == 400

    # Test main with mark 2 missing
    def test_mainStringInmark2Missing(self):
        url = '?mark1=78&mark3=25&mark4=78&mark5=37'
        response = get(self.URLFORCLASSIFYGRADE + url)
        assert "Missing module or invalid data for mark 2" in response.text
        assert response.status_code == 400
```

Figure 13: Unit and HTTP Tests of the Classify Grade Microservice

Anything else to highlight:

Continuous deployment has not been implemented for this function because it is deployed to Azure, this would be possible if the Azure account did not use SSO. However, it is difficult to log in to the QUB (Queen's University Belfast) Azure account using the CLI due to it being SSO. A way around this would be to deploy a VM on Azure which does the continuous deployment, or do CI/CD on Azure DevOps, but this was not feasible for the CSC-3065 project.

CORS has been set up in the Azure portal so that it trusts the QPC environment, and a CORS error does not occur. This ensures that the responses do not need the "access-control-allow-origins" header.

Function Three

Function : Average Grade

Repository URL: http://gitlab.hal.davecutting.uk/40266405/qubgrademe-averagegrade
Live Service URL: http://qubgrademe-averagegrade.40266405.qpc.hal.davecutting.uk/

Description of Implementation:

The average grade microservice is containerised and deployed as a PaaS on QPC Rancher. It has been developed using Java, specifically the spring boot framework with maven (spring boot version 2.7.5). Spring Intializr has been used to automatically generate the folder structure necessary for the microservice, including the "qubgrademeaveragegrade" and "test" folders which will be most important for this implementation.

In the outer folder, a few important files have been generated which are necessary to make the Java application run with no issues. The first of these files is the "mvnw" file which is used as a command in CLI to build, test and run the application. For example, a command in PowerShell to build and test the application would be "./mvnw install". The other important file is the "pom.xml" file which includes all the necessary dependencies, plugins and descriptions for the average grade application. These dependencies include two packages from spring boot, and one plugin from spring boot.

The main "qubgrademeaveragegrade" folder includes two files, the application file and the controller file. The application file has a main method which runs the spring boot application, and the controller file includes all the endpoints which are necessary for the application to run.

The first method in the controller is the default route for the application, it can be hit by the user by calling URL + "/" and it welcomes the user to the microservice. It only responds with status code 200 and does not require any parameters. To ensure that any CORS errors are not thrown, a response header of "Access-Control-Allow-Origin" is added. There are two error handling methods, one of them to return an error message just as a function and the other error method is called if a missing parameter exception is thrown, as seen in figure 14 this method listens for the exception and is executed once a "MissingServletRequestParameterException" is thrown.

```java
// Exception handler for if a parameter is missing
@CrossOrigin
@ExceptionHandler(MissingServletRequestParameterException.class)
public ResponseEntity<String> handleMissingParams(MissingServletRequestParameterException ex) {
    String name = ex.getParameterName();
    String message = "{\"Message\": \" " + name + " parameter is missing \"}";
    // Setting the HTTP headers to allow for CORS
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("Access-Control-Allow-Origin", "*");
    return ResponseEntity.badRequest()
    .body(message); // Returning the correct message
}
```

Figure 14: Error Handler for a Missing Parameter Exception

The average grade endpoint is the final endpoint and method in the controller class. It can be executed by calling the API (Application Programming Interface) with the URL + "/averagegrade" and with parameters in the form of "?mark1=<mark>&mark2=<mark>" and so on up to mark 5. An example of a valid URL for this microservice is:

http://qubgrademe-averagegrade.40266405.qpc.hal.davecutting.uk/averagegrade?mark1=56&mark2=70&mark3=65&mark4=78&mark5=76

If an exception is not thrown about a missing parameter, this method then tries to parse all the marks into integers, if they fail to parse then it will return a bad request with a message saying that the mark/module is invalid. The parsing and returning a bad request in this instance can be seen in figure 15.

```java
try {
    // Calculating the average grade
    averageGrade = Integer.parseInt(mark1) + Integer.parseInt(mark2) + Integer.parseInt(mark3) + Integer.parseInt(mark4) + Integer.parseInt(mark5);
```

```java
} catch (org.springframework.web.method.annotation.MethodArgumentTypeMismatchException e) {
    // Setting the HTTP headers to allow for CORS
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("Access-Control-Allow-Origin", "*");
    return ResponseEntity.badRequest()
    .body("{\"Message\": \"One of the modules is invalid, try making it an integer\"}"); // Returning the correct message


} catch (java.lang.NumberFormatException e) {
    // Setting the HTTP headers to allow for CORS
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("Access-Control-Allow-Origin", "*");
    return ResponseEntity.badRequest()
    .body("{\"Message\": \"One of the modules is invalid, try making it an integer\"}"); // Returning the correct message
}
```

Figure 15: Error handling for if a mark is not an integer

Once this check has been completed, the method then validates to see if a mark is between 0-100 and returns a bad request if the mark is outside this range. Finally, if all the validation checks pass then the average grade is calculated and returned to the user with a HTTP code of 200 as seen in figure 16.

```java
averageGrade = averageGrade / 5;

// Setting the HTTP headers to allow for CORS
HttpHeaders responseHeaders = new HttpHeaders();
responseHeaders.set("Access-Control-Allow-Origin", "*");
return ResponseEntity.ok()
.body("{\"Message\": \"Your average grade is: " + Integer.toString(averageGrade) + "\"}"); // Returning the correct message
```

Figure 16: Returning the message to the user with the Average Grade

The average grade microservice has a JAR file which acts as the executable and is generated using a "./mvnw install" command in PowerShell. Once the new JAR file has been created, the Dockerfile can copy it onto the docker image so that it can be executed in a container. For this microservice, the "openjdk:20-ea-19-jdk" docker image has been used and port 8080 has been exposed so that the application is accessible from outside of the container. Finally, as seen in figure 17 the entry point is set to the JAR file to run the application.

```
FROM openjdk:20-ea-19-jdk
COPY target/qubgrademe-averagegrade-0.0.1-SNAPSHOT.jar qubgrademe-averagegrade-0.0.1-SNAPSHOT.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/qubgrademe-averagegrade-0.0.1-SNAPSHOT.jar"]
```

Figure 17: Dockerfile for the Average Grade Microservice

Description of Testing:

Testing in the average grade application was first done manually before the application was deployed, every endpoint was checked in postman with all possibilities of valid and invalid data to ensure that the microservice is functioning correctly. Figure 18 shows an example of invalid data being tested to see if the correct response is received. The data is invalid because mark5 is empty, and it returns the correct response with status 400 bad request.

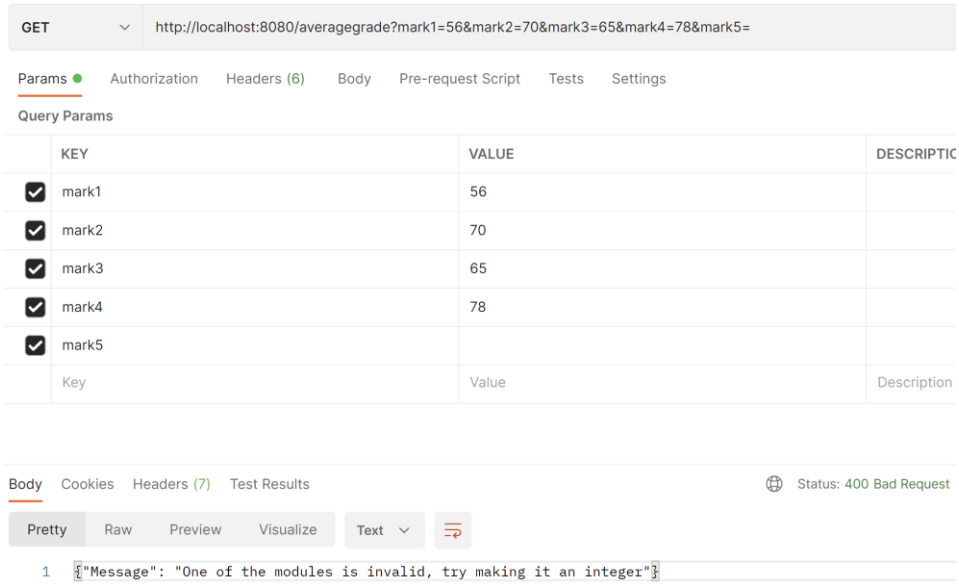


Figure 18: Testing the Average Grade Microservice Using Postman

After manual postman testing was complete, the application was deployed to QPC rancher and a CI/CD file with unit and HTTP (integration) testing could be written. Junit 5 has been used as the testing framework to assert that the responses returned are correct.

Unit and HTTP testing both check to see if the microservice behaves correctly for good and bad data. All tests are written in the "tests" folder which is inside of the "qubgrademeaveragegrade" folder. There is an application test which ensures that the controller has been loaded in. The welcome page is also tested with a unit and HTTP test to ensure that that endpoint works correctly locally and deployed. Figure 19 shows how an example valid and invalid test has been written with unit tests being the top image and HTTP tests being the bottom image. All tests check to ensure that the response code is the correct one and check to see if the message returned is also correct. Finally, as

the httpURLConnection object throws an exception if an error 400 is returned, it is only checked if that exception is thrown (the body of the response is not asserted).

```java
// Testing the average grade method with valid data
@Test
public void shouldCalculateAverageGrade() throws Exception {
    this.mockMvc.perform(get("/averagegrade?mark1=75&mark2=75&mark3=75&mark4=75&mark5=75")).andExpect(status().isOk())
            .andExpect(content().string(containsString("Your average grade is: 75")));
}

// Testing the average grade method with marks over 100
@Test
public void shouldCalculateAverageGrademark1Over100() throws Exception {
    this.mockMvc.perform(get("/averagegrade?mark1=156&mark2=75&mark3=75&mark4=75&mark5=75")).andExpect(status().isBadRequest())
            .andExpect(content().string(containsString("Module 1 is invalid, has to be greater than zero and less than 100")));
}
```

```java
// Testing the average grade with 53
@Test
public void shouldReturnAverageGrade53() throws Exception {
    String GET_URL = "http://qubgrademe-averagegrade.40266405.qpc.hal.davecutting.uk/averagegrade?mark1=53&mark2=53&mark3=53&mark4=53&mark5=53";
    URL obj = new URL(GET_URL);
    HttpURLConnection con = (HttpURLConnection) obj.openConnection();
    con.setRequestMethod("GET");
    int responseCode = con.getResponseCode();

    BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
    String inputLine;
    StringBuffer response = new StringBuffer();

    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
    in.close();

    assertEquals(responseCode, 200);
    assertEquals(response.toString(), "{\"Message\": \"Your average grade is: 53\"}");
}
```

```java
// Testing the average grade with a missing mark
@Test
public void shouldReturnAverageGradeMarkMissing() throws Exception {
    try {
        String GET_URL = "http://qubgrademe-averagegrade.40266405.qpc.hal.davecutting.uk/averagegrade?mark1=70&mark3=70&mark4=70&mark5=70";
        URL obj = new URL(GET_URL);
        HttpURLConnection con = (HttpURLConnection) obj.openConnection();
        con.setRequestMethod("GET");
        int responseCode = con.getResponseCode();

        BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
        String inputLine;
        StringBuffer response = new StringBuffer();

        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        in.close();

        System.out.print("Response on new line");
        System.out.print(response.toString());
    } catch(Exception e)
    {
        assertEquals("java.io.IOException: Server returned HTTP response code: 400 for URL: http://qubgrademe-averagegrade.40266405.qpc.hal.davecutting.uk/averagegrade?mark1=70&mark3=70&mark4=70&mark5=70"
        , e.toString());
        System.out.println("Exception string" + e.toString());
    }
}
```

Figure 19: Unit and HTTP Testing the Average Grade Microservice

CI has been achieved using a “gitlab-ci.yml” file and the unit/HTTP tests are run on Gitlab QPC using a GitLab runner. 17 tests have been ran and they test all the possible returns from the function, figure 20 shows the tests passing in Gitlab CI.

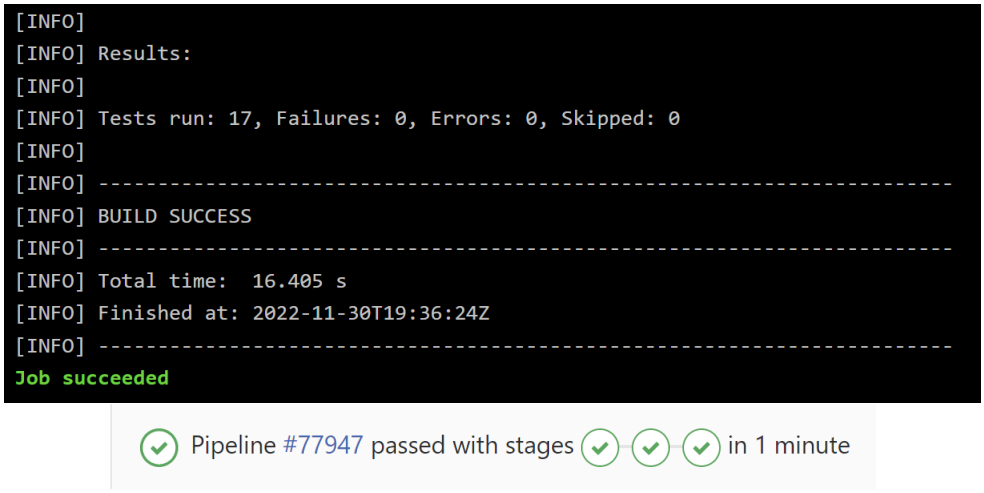


Figure 20: Average grade CI/CD results

Anything else to highlight:

In the CI/CD file there are two extra jobs which are done on the runner, as seen in figure 21 there is a build job and a deploy job. Both the build and test jobs use the “maven:latest” docker image and the continuous deployment part uses the “docker-latest” image to build, login to the docker registry and push the image up.

```yaml
# Using the maven docker image
image: maven:latest

# Building the java microservice
build:
  stage: build
  script:
    - mvn compile

# Testing the java microservice
test:
  stage: test
  script:
    - mvn test -X

# Deploying the java microservice
deploy:
  image: docker:latest
  stage: deploy
  tags:
    - "dind"
  script:
    - docker build -t registry.hal.davecutting.uk/40266405/qubgrademe-averagegrade .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker push registry.hal.davecutting.uk/40266405/qubgrademe-averagegrade
```

Figure 21: Average Grade CI/CD File

Function Four

Function: Percent Needed For First

Repository URL: http://gitlab.hal.davecutting.uk/40266405/qubgrademe-percentneededforfirst
Live Service URL: http://qubgrademe-percentneededforfirst.40266405.qpc.hal.davecutting.uk/

Description of Implementation:

The percent needed for first microservice is written in C# using the ASP.Net Core framework with .Net version 6.0. The microservice uses the PaaS (Platform as a Service) paradigm and is deployed on QPC Rancher as a docker image. The docker image for this microservice is generated using a Dockerfile which uses the "mcr.microsoft.com/dotnet/sdk:6.0" base image.

The folder structure has been generated using visual studio as a .net core web API. This generates an src folder and a project folder named "qubgrademe-percentneededforfirst". There are many folders and files in here, but the most important ones include the controller's folder, the properties folder, the "program.cs" file and the "qubgrademe-percentneededforfirst.csproj" file.

Inside of the main folder there are "QubGradeMePercentNeededForFirstController.cs" and the "Modules.cs" files, the first file includes all the endpoints and necessary logic to calculate the percentage needed for a first. The constructor uses dependency injection to instantiate the logger, this logger is used for the system admin to see what is happening with the API and for debugging any issues that may occur.

The first endpoint in the percent needed for first microservice is the welcome endpoint, which can be called by the user using the URL + "/". This endpoint only returns a response code of 200 and returns a message to welcome the user.

The second endpoint is the percent needed for first endpoint; it is called by using the URL + "/api/qubgrademe_percentneededforfirst/getpercentneeded" + all of the marks in the format of "?mark1=<mark1>&mark2=<mark2>" and so on up to mark 5. When this endpoint is called, the marks are de-serialised into a Modules class that has all the marks as values. These marks are constructed with an empty string, so if a mark is missing from the parameter list it will be instantiated as an empty string. Here is an example of a working valid URL to calculate the percent needed for a first:

http://qubgrademe-percentneededforfirst.40266405.qpc.hal.davecutting.uk/api/QubGradeMe_PercentNeededForFirst/getPercentNeeded?mark1=54&mark2=54&mark3=64&mark4=64&mark5=43

Data validation in this method ensures that all the marks are integers, and that they are between 0 and 100. If a mark is blank or an empty string it will also catch this, and return a HTTP 400 bad request if any validation errors occur. Figure 22 shows an example of how validation is done in this microservice.

```
_logger.LogInformation("Validating marks");
// Checking if the modules are blank
if (modules.mark1.Trim() == "")
{
    return BadRequest("{\"Message\": \"You are missing module 1 \"}");
}
```

Figure 22: Example of Mark Validation in the Percent Needed for First Microservice

Once the marks have been validated, the average is calculated and if it is above 70 it returns the percentage received. If the average is below 70, it calculates how much the user needs to get up to a 70 and returns that. This implementation is shown in figure 23.

```
// Calculating the average
_logger.LogInformation("Calculating the average");
var sum = intmark1 + intmark2 + intmark3 + intmark4 + intmark5;
var average = sum / 5;

// Returning the response
_logger.LogInformation("Returning the correct message to the user");
if(average >= 70)
{
    return Ok("{\"Message\": \"You have already achieved a first with " + average + "%\"}");
}
return Ok("{\"Message\": \"You need " + (70 - average) + "% to get a first\"}");
```

Figure 23: Calculating the Percentage Needed for First and Returning a Message

Finally, the “program.cs” and “.csproj” files function as management files. In the “program.cs” file, the controller and CORS is set up to ensure that it can trust whoever is calling the microservice and that the endpoints are exposed. The “.csproj” file holds all the dependencies for the project. These dependencies are just ones needed to run an ASP.net Core web API.

Description of Testing:

Testing for the percent needed for first microservice was first done manually like the rest of the applications. However, because C# ASP.Net Core comes with Swagger ready to go it was much easier to test with it writing HTTP requests for the programmer. All aspects of the microservice were tests with good and bad data, shown in figure 24 is an example of a HTTP test in Swagger to check if a correct error is thrown for invalid data. Visual Studio was used for debugging to find any issues with the code and it was extremely useful for this process.

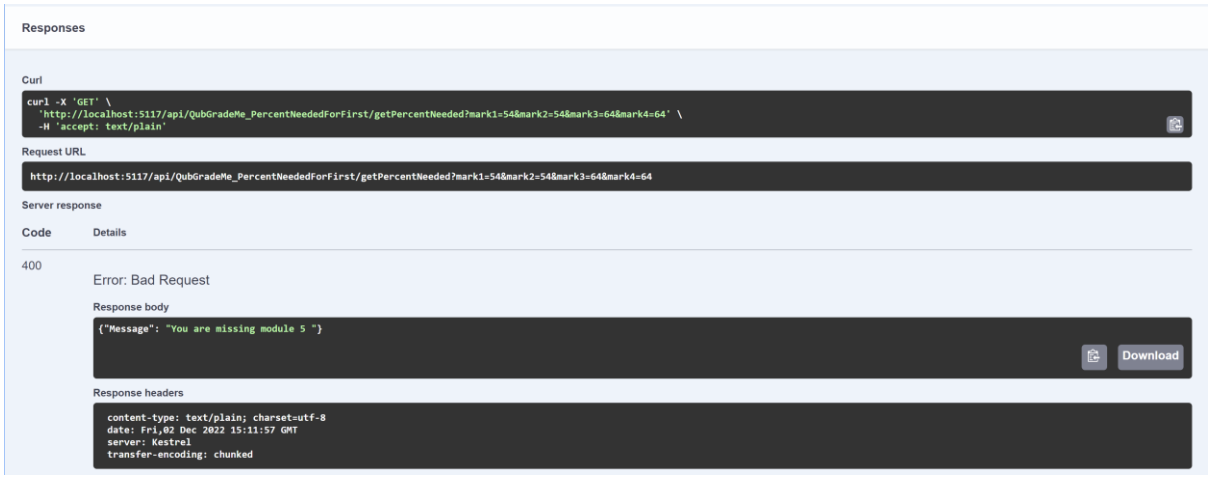


Figure 24: Swagger Manual Testing for the Percent Needed for First API

XUnit with moq and Fluent Assertions was then used to test the microservice with valid and invalid data, ensuring that all cases are covered in the tests to catch out any bugs in the system. Unit and HTTP tests were performed to ensure that the application works locally and deployed on QPC Rancher. The logger is mocked out using moq so that the tests are only testing the unit logic, and fluent assertions is used to check that the values returned are the correct ones. Figure 25 shows examples of valid and invalid unit and HTTP (Integration) tests.

```
[Fact]
0 references | Run Test | Debug Test
public void TestPercentNeededForFirstValidDataBelowAFirst()
{
    // Arrange
    var controller = new QubGradeMe_PercentNeededForFirstController(_logger.Object);
    Modules modules = new Modules
    {
        mark1 = "70",
        mark2 = "65",
        mark3 = "70",
        mark4 = "65",
        mark5 = "65"
    };

    // Act
    var result = controller.getPercentNeededForFirst(modules);

    // Assert
    OkObjectResult? okResult = result.Result as OkObjectResult;

    result.Should().NotBeNull();
    result.Should().BeOfType<ActionResult<string>>();
    okResult.Should().BeOfType<OkObjectResult>();
    okResult.Should().NotBeNull();
    okResult.Value.Should().Be("{\"Message\": \"You need " + 3 + "% to get a first\"}");
}

[Fact]
0 references | Run Test | Debug Test
public void TestPercentNeededForFirstMissingmark1()
{
    // Arrange
    var controller = new QubGradeMe_PercentNeededForFirstController(_logger.Object);
    Modules modules = new Modules
    {
        mark5 = "70",
        mark2 = "70",
        mark3 = "70",
        mark4 = "70",
    };

    // Act
    var result = controller.getPercentNeededForFirst(modules);

    // Assert
    BadRequestObjectResult? okResult = result.Result as BadRequestObjectResult;

    result.Should().NotBeNull();
    result.Should().BeOfType<ActionResult<string>>();
    okResult.Should().BeOfType<BadRequestObjectResult>();
    okResult.Should().NotBeNull();
    okResult.Value.Should().Be("{\"Message\": \"You are missing module 1 \"}");
}
```

```
[Fact]
0 references
public async void TestPercentNeededForFirstValidDataBelowAFirst()
{
    // Arrange
    var newUrl = url + "?mark1=65&mark2=70&mark3=65&mark4=70&mark5=65";

    using var client = new HttpClient();

    // Act
    var response = await client.GetAsync(newUrl);

    // Assert
    string responseBody = await response.Content.ReadAsStringAsync();
    responseBody.Should().Be("{\"Message\": \"You need " + 3 + "% to get a first\"}");
}

[Fact]
0 references
public async void TestPercentNeededForFirstMissingmark1()
{
    // Arrange
    var newUrl = url + "?mark2=70&mark3=70&mark4=70&mark5=70";

    using var client = new HttpClient();

    // Act
    var response = await client.GetAsync(newUrl);

    // Assert
    string responseBody = await response.Content.ReadAsStringAsync();
    responseBody.Should().Be("{\"Message\": \"You are missing module 1 \"}");
}
```

Figure 25: Unit and HTTP Tests for Percent Needed for First Microservice

CI testing has been implemented to run on the GitLab runner with twenty-four tests being written, figure 26 shows the test results returned from the runner.

```
Passed!  - Failed:     0, Passed:    24, Skipped:     0, Total:    24, Duration: 179 ms - /builds/40266405/qubgrademe-percentneededforfirst/src/qubgrademe-percentneededforfirst.test/bin/Debug/net6.0/qubgrademe-percentneededforfirst.test.dll (net6.0)
Job succeeded
```

Pipeline #71657 passed with stages in 1 minute and 46 seconds

Figure 26: CI/CD Pipeline Results for the Percent Needed for First Function

Anything else to highlight:

CI/CD is achieved using ".gitlab-ci.yml" file which has three jobs, a build, a test, and a deploy job as seen in figure 27. The build and test jobs use the "mcr.microsoft.com/dotnet/sdk:6.0" docker image and the deploy job uses the "docker:latest" image to build, login and push to the QPC container registry.

```yaml
# Uses the c# dotnet docker image
image: mcr.microsoft.com/dotnet/sdk:6.0

# Build the project and restore dependencies
build:
  stage: build
  script:
    - 'dotnet build ./src/qubgrademe-percentneededforfirst'

# Test the project
test:
  stage: test
  script:
    - 'dotnet test ./src/qubgrademe-percentneededforfirst.test'

# Deploying the project
deploy:
  image: docker:latest
  stage: deploy
  tags:
    - "dind"
  script:
    - cd ./src/qubgrademe-percentneededforfirst
    - docker build -t registry.hal.davecutting.uk/40266405/qubgrademe-percentneededforfirst .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker push registry.hal.davecutting.uk/40266405/qubgrademe-percentneededforfirst
```

Figure 27: CI/CD File for the Percent Needed for First Microservice

Task B: Improvements

Front end repository URL: http://gitlab.hal.davecutting.uk/40266405/qubgrademe-frontend
Front end live URL: http://qubgrademe-frontend.40266405.qpc.hal.davecutting.uk/

Front End Improvements

The front end has been deployed as a HTML, CSS and JS container to QPC rancher on the link at the start of this section. Any improvements that have been made for other sections have been highlighted in the relevant sections further on in the report.

Error handling in the front end

Error handling is achieved through a “CheckInput” function which is called at the start of every call to a microservice. This function checks to see if the modules have been put into the input fields and checks if marks have been input and if they are valid. A mark would be invalid if it is blank, not an integer or if it is not between 0 and 100. Figure 28 shows how the modules and marks are validated.

```javascript
// Validating each module
for(var i = 0; i < moduleArray.length; i++)
{
    // Checking if the module is blank
    if(moduleArray[i].trim() == '')
    {
        alert("Please insert a value for module: " + (i+1));
        return false;
    }
}
```

```
// Validating each mark
for(var i = 0; i < moduleArray.length; i++)
{
    // Checking if the mark is blank
    if(markArray[i].trim() == '')
    {
        alert("Please insert a value for mark: " + (i+1));
        return false;
    }
    // Checking if the mark is a number
    if(isNaN(markArray[i].trim()))
    {
        alert("Mark " + (i+1) + " has to be a number");
        return false;
    }
    // Checking if the number is greater than or equal to 0 or less than or equal to 100
    if(parseInt(markArray[i]) < 0 || markArray[i] > 100)
    {
        alert("Mark " + (i+1) + " has to be >= 0 and <= 100");
        return false;
    }
}
return true;
```

Figure 28: Validating Modules and Marks in the Front End

If a mark is invalid, it will alert the user with what is wrong and will ensure that the API is not called. Figure 29 shows how this alert looks for the user. The rest of the alert messages can be seen explained in the video.

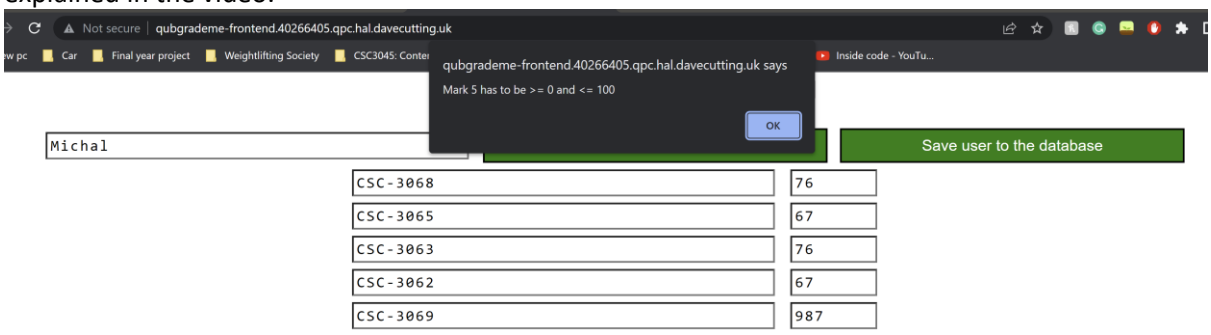


Figure 29: Alert on the Front End for an Invalid Mark

The save user to database and retrieve user functions also validate the username by checking if it is empty or not.

Making the Front End Run Asynchronously

The HTTP requests that used to call each microservice have been replaced with fetch requests which run asynchronously allowing the user to continue using the application whilst waiting for a response from the web API. Figure 30 shows an example of a fetch statement calling the average grade microservice and then displaying the results from that fetch. These fetch statements also have a catch clause ensuring that if any exception is thrown it does not break the application.

```
await fetch(url, {
method: 'GET',
headers: {
},
}).then((response) => response.json())
.then((data) => {
    messageReceivedFlag = true;
    displayTotal(data.Message);
}).catch(err => {
    console.log("Calling backup");
});
```

Figure 30: Fetch Statement in the Front End Calling the Get Total Microservice

Continuous Deployment for the Front End

A “gitlab-ci.yml” (CI/CD) file has been implemented to ensure continuous deployment for the front end. As seen in figure 31, it uses the “docker:latest” image to build the Dockerfile, login to the container registry and push the image to the container registry hosted on QPC Gitlab.

```
deploy:
  image: docker:latest
  stage: deploy
  tags:
    - "dind"
  script:
    - docker build -t registry.hal.davecutting.uk/40266405/qubgrademe-frontend .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker push registry.hal.davecutting.uk/40266405/qubgrademe-frontend
```

Figure 31: CD File for the Front End

Configuration of Routes to the Proxies in a JSON File

The URLs to the proxies are stored in a “config.json” file in the same folder as the “index.html” file. This configuration file includes an array of all proxies which can be updated dependent on the number of proxies deployed. Figure 32 shows this file and how the proxies are stored as a JSON.

```
{
    "proxyURLS": ["http://qubgrademe-proxy.40266405.qpc.hal.davecutting.uk/proxy", "http://qubgrademe-proxy-backup.40266405.qpc.hal.davecutting.uk/proxy"]
}
```

Figure 32: Config.json in the Front End

This configuration file is read into the front end using the fetch API and is then stored in a proxy’s array on the front end. Figure 33 shows how this is done. How load balancing is done is explained in task D. Using a config file within the proxy is explained in task C.

```
// Retrieving the URL's from the configuration file
fetch("config.json")
  .then(response => response.json())
  .then(json => proxyArray = json.proxyURLS);
```

Figure 33: Fetching the Proxy URLs from the Config File

PHP Improvements

PHP Sorted Modules repository URL:
http://gitlab.hal.davecutting.uk/40266405/qubgrademe-sortmodules
PHP Sorted Modules live URL: http://qubgrademe-sortmodules.40266405.qpc.hal.davecutting.uk/
PHP Max Min repository URL:
http://gitlab.hal.davecutting.uk/40266405/qubgrademe-maxmin
PHP Max Min URL:
http://qubgrademe-maxmin.40266405.qpc.hal.davecutting.uk/

PHP Sorted Modules and Max Min Error Handling

Error handling in the sorted modules and max min microservices is done in the same way using a new function that has been added into the “functions.inc.php” file. As seen in figure 34, it loops through the modules, marks, and checks if the marks are null, not an integer or outside the range of 0 to 100. It then also checks to see if the modules are blank or null. If the module/mark is invalid, it returns a HTTP 400 bad request error with a personalised message that says what is invalid about

the data that has been passed in. The URLS at the start of the PHP improvements section show an example of an error message printing for invalid data.

```php
function checkInput($modules, $marks)
{
  for($i = 0; $i < count($marks); $i++)
{
    // Check if marks are null
    if(is_null($marks[$i])) {
        http_response_code(400);
        $output = array(
            "error" => true,
            "Message" => "A mark can't be null"
        );
        echo json_encode($output);
        exit();
    }
    // Check if mark is not an integer
    $int_value = ctype_digit($marks[$i]) ? intval($marks[$i]) : null;
    if($int_value === null) {
        http_response_code(400);
        $output = array(
            "error" => true,
            "Message" => "A mark has to be an integer"
        );
        echo json_encode($output);
        exit();
    }
    // Check if marks are greater than 0 and less than 100
    if($marks[$i] < 0 || $marks[$i] > 100) {
        http_response_code(400);
        $output = array(
            "error" => true,
            "Message" => "A mark has to be between 0 and 100"
        );
        echo json_encode($output);
        exit();
    }
}
```

```php
for($i = 0; $i < count($modules); $i++)
{
    // Check if modules are null
    if(is_null($modules[$i])) {
        http_response_code(400);
        $output = array(
            "error" => true,
            "Message" => "A module can't be null"
        );
        echo json_encode($output);
        exit();
    }
    if($modules[$i] == "") {
        http_response_code(400);
        $output = array(
            "error" => true,
            "Message" => "A module can't be blank"
        );
        echo json_encode($output);
        exit();
    }
```

Figure 34: Validating the Parameters in PHP

A valid URL for the PHP Max Min can be seen below:
http://qubgrademe-maxmin.40266405.qpc.hal.davecutting.uk/?mark1=25&mark2=89&mark3=56&mark4=12&mark5=87&module1=ds&module2=fd&module3=fdsg&module4=dsf&module5=sdf

A valid URL for the PHP Sorted Modules can be seen below:
http://qubgrademe-sortmodules.40266405.qpc.hal.davecutting.uk/?mark1=25&mark2=89&mark3=56&mark4=12&mark5=87&module1=ds&module2=fd&module3=fdsg&module4=dsf&module5=sdf

PHP Sorted Modules and Max Min CI/CD

Sorted modules and max min both now include CI/CD files which unit test and HTTP (Integration) test the application with all possible valid and invalid marks/modules. There a few important files needed to get testing working with PHP, these include the “composer.json”, “phpunit.xml”, “bootstrap.php” and the actual test files.

The composer file states which dependencies are necessary for the microservice, the main ones are PHPUnit for writing assertions and declaring test cases, Guzzle HTTP for sending HTTP requests. The “phpunit.xml” file then states where the tests are located (in the tests folder) and where the bootstrap for the project is located. The job of the bootstrap is to import any .php files that it finds in the test folder so that PHPUnit knows exactly where the tests are.

The unit tests for both microservices are written in the same format, an associative array is generated for both the marks and the modules and the get max min/get sorted modules function is called. The results from that function call are asserted to an expected string to see if they return the correct values. Figure 35 shows an example of unit tests with valid and invalid data for both the max min and sorted modules microservices. The invalid data is checked against the check input function instead of the get max min/get sorted modules function as it is always called before the data is manipulated.

```php
// Test check input with valid data
0 references | 0 overrides
public function testGetMaxMinValidData(): void
{
    $marks = array("70", "60", "70", "50", "70");
    $modules = array("m1", "m2", "m3", "m4", "m5");

    $this->assertContains('m1 - 70', getMaxMin($modules, $marks));
    $this->assertContains('m4 - 50', getMaxMin($modules, $marks));
}

// Test check input with missing module 1
0 references | 0 overrides
public function testGetMaxMinMissingModule1(): void
{
    $marks = array("70", "60", "70", "50", "70");
    $modules = array("m2", "m3", "m4", "m5");

    $this->assertEquals(checkInput($marks, $modules), false);
}
```

```php
// Test max min with valid data
0 references | 0 overrides
public function testSortModules(): void
{
    $marks = array(70, 60, 70, 50, 70);
    $modules = array("m1", "m2", "m3", "m4", "m5");

    $firstCheck = array("module"=>"m1", "marks"=>"70");
    $secondCheck = array("module"=>"m3", "marks"=>"70");
    $thirdCheck = array("module"=>"m5", "marks"=>"70");
    $fourthCheck = array("module"=>"m2", "marks"=>"60");
    $fifthCheck = array("module"=>"m4", "marks"=>"50");

    $this->assertContains($firstCheck, getSortedModules($modules, $marks)[0]);
    $this->assertContains($secondCheck, getSortedModules($modules, $marks)[1]);
    $this->assertContains($thirdCheck, getSortedModules($modules, $marks)[2]);
    $this->assertContains($fourthCheck, getSortedModules($modules, $marks)[3]);
    $this->assertContains($fifthCheck, getSortedModules($modules, $marks)[4]);
}

// Test check input with missing module 1
0 references | 0 overrides
public function testCheckInputMissingModule1(): void
{
    $marks = array("70", "60", "70", "50", "70");
    $modules = array("m2", "m3", "m4", "m5");

    $this->assertEquals(checkInput($marks, $modules),false);
}
```

Figure 35: Unit tests of the Max Min and Sorted Modules Microservices

HTTP (Integration) testing in the PHP microservices is also done in the same way for both. The Guzzle HTTP package is used to call the microservices, for valid data the body of the request and HTTP status code is checked. For invalid data only the response code is checked, because if a HTTP error 400 request is returned Guzzle HTTP throws an exception and the test fails (So these exceptions need to be caught). Tests have been written to validate all responses from the system, but figure 36 only shows a valid and invalid integration test for both the max min and get sorted modules microservices. (The rest of the tests can be found in the code)

```php
// Test the API with valid data
0 references | 0 overrides
public function testGetMaxMinIntegration(): void
{
    $client = new Client();
    $res = $client->request('GET', 'http://qubgrademe-maxmin.40266405.qpc.hal.davecutting
    $json = $res->getBody()->getContents();

    print_r($json);

    $this->assertContains("\"max_module\":\"m1 - 70\",\"min_module\":\"m4 - 50\"", $json);
    $this->assertTrue(str_contains($json, "false"));
}

// Test the API with a missing module
0 references | 0 overrides
public function testGetMaxMinIntegrationMissingModule(): void
{
    $this->expectException(ClientException::class);
    $client = new Client();
    $res = $client->request('GET', 'http://qubgrademe-maxmin.40266405.qpc.hal.davecutting
}
```

```php
// Test the API with valid data
0 references | 0 overrides
public function testSortModulesIntegration(): void
{
    $client = new Client();
    $res = $client->request('GET', 'http://qubgrademe-sortmodules.40266405.qpc.hal.davecut
    $json = $res->getBody()->getContents();
    $this->assertTrue(str_contains($json, "false"));
    $responseCheck = "\"sorted_modules\":[{\"module\":\"m1\",\"marks\":\"70\"},{\"module\"
    $this->assertTrue(str_contains($json, $responseCheck));
}

// Test the API with a missing module
0 references | 0 overrides
public function testSortModulesIntegrationMissingModule(): void
{
    $this->expectException(ClientException::class);
    $client = new Client();
    $res = $client->request('GET', 'http://qubgrademe-sortmodules.40266405.qpc.hal.davecut
}
```

Figure 36: Integration Tests for the Max Min and Sorted Modules Microservices

A CI/CD pipeline is made for both microservices using a ".gitlab-ci.yml" with two jobs including a test, and a deploy job. The test job runs the unit and HTTP tests whereas the deploy job uses the

“docker:latest” image to build push and deploy the docker images to the QPC GitLab container registry. Figure 37 shows the “.gitlab-ci.yml” files for both the max min and sorted modules microservices. The test job uses the “php:8.0” image and firstly updates, then installs the dependencies and finally runs the tests using PHPUnit.

```yaml
# Uses the python docker image
image: php:7.2-apache

# Test the project
test:
  image: php:8.0
  tags:
    - "dind"
  script:
    # Update and install necessary packages
    - apt -y update
    - apt -y install git zip

    # Install composer
    - curl -sS https://getcomposer.org/installer | php

    # Create the composer lock file
    - php composer.phar install

    # Run the tests
    - vendor/bin/phpunit --configuration phpunit.xml --coverage-text

deploy:
  image: docker:latest
  stage: deploy
  tags:
    - "dind"
  script:
    - docker build -t registry.hal.davecutting.uk/40266405/qubgrademe-maxmin .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker push registry.hal.davecutting.uk/40266405/qubgrademe-maxmin
```

```yaml
# Uses the python docker image
image: php:7.2-apache

# Test the project
test:
  image: php:8.0
  tags:
    - "dind"
  script:
    # Update and install necessary packages
    - apt -y update
    - apt -y install git zip

    # Install composer
    - curl -sS https://getcomposer.org/installer | php

    # Create the composer lock file
    - php composer.phar install

    # Run the tests
    - vendor/bin/phpunit --configuration phpunit.xml --coverage-text

deploy:
  image: docker:latest
  stage: deploy
  tags:
    - "dind"
  script:
    - docker build -t registry.hal.davecutting.uk/40266405/qubgrademe-sortmodules .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker push registry.hal.davecutting.uk/40266405/qubgrademe-sortmodules
```

Figure 37: CI/CD Files for the PHP Microservices (Max Min on the Left) (Sort Modules on the Right)

There are nineteen tests for the max min microservice, and they all pass as seen in figure 38, the microservice has been successfully redeployed as seen in the pipeline in figure 38. There is also a screenshot of the container redeploying to the container registry and passing.

```
$ vendor/bin/phpunit --configuration phpunit.xml --coverage-text
PHPUnit 9.5.26 by Sebastian Bergmann and contributors.

Warning:       No code coverage driver available

E{"error":false,"modules":["m1","m2","m3","m4","m5"],"marks":["70","60","70","50","70"],"max_module":"m1 - 70","min_module":"m4 - 50"}........
{"error":true,"Message":"A mark has to be an integer"}Job succeeded
```

```
latest: digest: sha256:fe83a8e8904473f824dcd786d8015996918641041df28cdcf0139b0381a5a69a size: 3450
Job succeeded
```

Pipeline #79998 passed with stages ✓ ✓ in 31 seconds

Figure 38: Pipeline Results for the Max Min Microservice

There are eighteen tests for the sorted modules microservice, and they all pass as seen in figure 39, the microservice is also successfully redeployed as seen in the pipeline screenshot in figure 39. There is also a screenshot of the container redeploying to the container registry and passing.

```
$ vendor/bin/phpunit --configuration phpunit.xml --coverage-text
PHPUnit 9.5.26 by Sebastian Bergmann and contributors.

Warning:       No code coverage driver available

........F{"error":true,"Message":"A mark has to be an integer"}Job succeeded
```

```
latest: digest: sha256:4924e026373f2ccb327a38cb55cff669049edcca1d2018831a7daebeef9296d4 size: 3450
Job succeeded
```

Pipeline #78811 passed with stages ✓ ✓ in 26 seconds

Figure 39: Pipeline Results for the Sorted Modules Microservice

Task C: Proxy

Repository URL: http://gitlab.hal.davecutting.uk/40266405/qubgrademe-proxy
Live Service URL: http://qubgrademe-proxy.40266405.qpc.hal.davecutting.uk/

Implementation Details:

The proxy has been implemented with C# using the ASP.Net Core framework and has been containerised and deployed onto QPC rancher using the PaaS (Platform as a Service) paradigm. It uses .Net Version 6.0 and the container uses the "mcr.microsoft.com/dotnet/sdk:6.0" base image. The proxy is completely configurable by using environment variables to store the endpoint URLs and service discovery is implemented with endpoints to add and update URLs in the service registry. (The service registry is stored using environment variables).

The proxy has four endpoints to perform all the necessary logic that it needs to perform. These include a welcome endpoint which is called by URL + "/" which welcomes the user to the microservice. The next endpoint is the proxy endpoint which forwards the necessary requests to all the correct microservices and is called by using the URL + "/proxy" endpoint. The final two endpoints are used for service discovery to update and add to the service registry and are explained further on in this section.

The proxy endpoint works by checking which parameters are in the URL request. Firstly, the connection string parameter is checked to see if a URL exists for that connection string, if it does not exist then a bad request is returned to the user. As seen in figure 40, the IConfiguration class is used to get the URLs from the environment variables. This class is extremely useful because when running locally, the "appsettings.json" file is used for the configuration and can be updated very easily, whereas once the application is deployed to QPC rancher the IConfiguration object checks the environment variables to see if the mapping for the URL exists there.

```
// Getting the endpoint (URL of the correct MicroService)
var endpoint = _configuration["ConnectionStrings:" + modules.ConnectionString];

_logger.LogInformation("The chosen endpoint is: " + endpoint);

// If the endpoint doesn't exist in the dictionary then return bad request
if(endpoint == "" || endpoint == null)
{
    return BadRequest("This endpoint is invalid");
}
```

Figure 40: Using the Configuration Object to Retrieve Values from the Configuration (Env Variables in Deployment)

Figure 41 shows an example of how the service registry can be created on start up in QPC Rancher to ensure that all the URLs can be retrieved by the proxy.

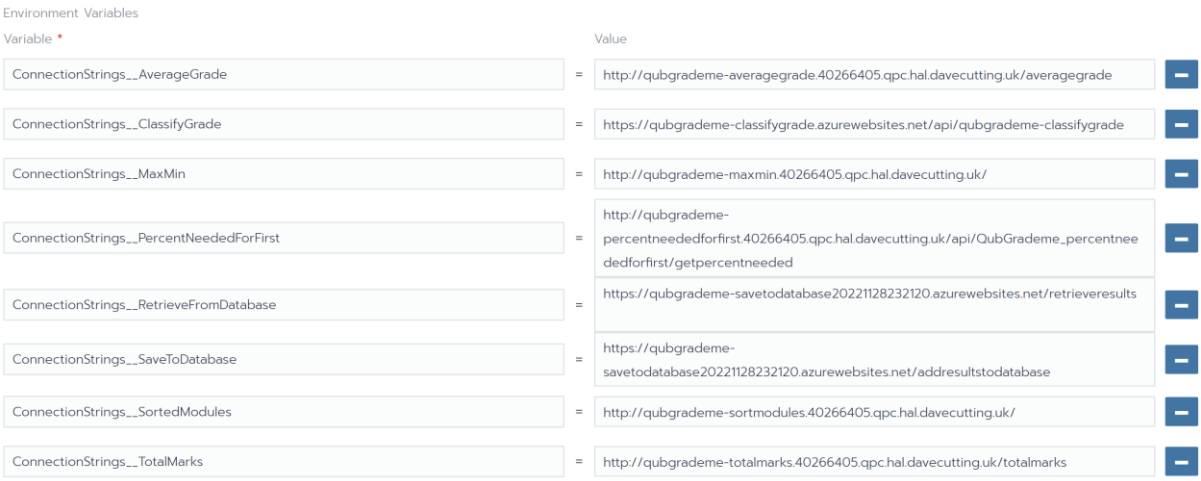


Figure 41: Declaring the Service Registry in QPC Rancher

An example of a valid proxy call has been shown below, calling the average grade microservice.

http://qubgrademe-proxy.40266405.qpc.hal.davecutting.uk/proxy?ConnectionString=AverageGrade&UserName=Michal&Module1=CSC-3068&Module2=CSC-3067&Module3=CSC-3064&Module4=CSC-3069&Module5=CSC-3062&Mark1=64&Mark2=64&Mark3=86&Mark4=65&Mark5=68

An example of an invalid call to the proxy with a wrong connection string has been shown below with the correct validation message.

http://qubgrademe-proxy.40266405.qpc.hal.davecutting.uk/proxy?ConnectionString=Average&UserName=Michal&Module1=CSC-3068&Module2=CSC-3067&Module3=CSC-3064&Module4=CSC-3069&Module5=CSC-3062&Mark1=64&Mark2=64&Mark3=86&Mark4=65&Mark5=68

The endpoint then moves on to validate the marks and modules if they exist in the connection string. The validation to check modules and marks is done in a similar way as the functions in task A. Validation occurs to see if a module is null or empty and to see if a mark is null, outside of the range of 0 to 100, or a string. The username and results are also checked to ensure they are not null/ empty when completing stateful saving actions. If any of these validation checks fail, then the proxy returns a HTTP 400 error code to the user.

Once the validation for the modules, marks, results and connection string has been completed, the URL for the correct microservice is constructed. Marks are always added onto the URL, modules are only added if they are necessary, and the result strings are only added on the end if saving to the database is necessary. The username is also added onto the URL if it is not empty in the request. Figure 42 shows an example of how the marks are added onto the end of the URL whenever it is being constructed.

```
// Else only the marks need added onto the endpoint
else
{
    for (int i = 0; i < intMarkList.Count(); i++)
    {
        if (i == 0)
        {
            endpoint += "?mark" + (i + 1) + "=" + intMarkList[i];
        }
        else
        {
            endpoint += "&mark" + (i + 1) + "=" + intMarkList[i];
        }
    }
}
```

Figure 42: Constructing the URL to Call the Necessary Microservice

Once the URL has been constructed, a HttpClient is used to call up the correct microservice and retrieve the information. If the microservice returns a HTTP code of 200, then this is also returned from the proxy and the message is relayed back to the front end. If the microservice returns a HTTP code of 400, then the proxy relays this back to the front end with the necessary message. For any other HTTP Codes, the message is set to Invalid Request and that HTTP code is sent to the front end. Finally, if the HttpClient throws an exception, which is due to the request timing out, this message is then also relayed to the front end as a HTTP 400 bad request. (Figure 43 shows how this works)

```
// Calling the microservice using a HTTP Client
using var client = new HttpClient();
var response = new HttpResponseMessage();
try
{
    response = await client.GetAsync(endpoint);
    _logger.LogInformation(endpoint);
}
catch(Exception e)
{
    _logger.LogInformation("Error with getting response");
    return BadRequest("The call to the neccessary microservice has timed out");
}

// Storing the response and code
string responseBody = await response.Content.ReadAsStringAsync();
var responseCode = response.StatusCode;

if(responseCode == HttpStatusCode.OK)
{

    return Ok(responseBody);
}
else if(responseCode == HttpStatusCode.BadRequest)
{
    return BadRequest(responseBody);
}

return StatusCode((int)responseCode, "Invalid Request");
```

Figure 43: Requesting the Data from the Correct Microservice and Relaying it Back to the Front End

CORS has been set up in the “program.cs” file to trust the front end so that chrome does not throw any errors when it is called.

Testing Details:

Local manual testing was first completed using Swagger to see if all the data validation is done correctly and if it maps to the correct addresses. Figure 44 shows this working to retrieve the correct

data for the average grade microservice. Invalid data was also checked to see if the proxy correctly validates bad input, such as a blank module, blank mark or if the mark is not between 0 and 100.

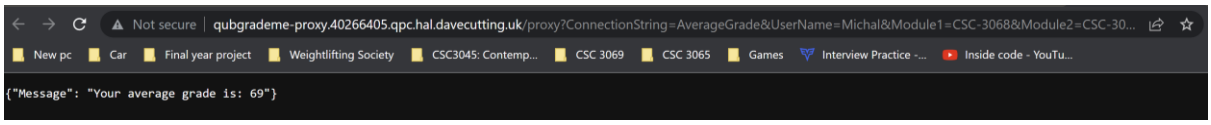


Figure 44: Manual Testing of the Proxy

Once manual testing was completed, unit and HTTP (integration) testing using XUnit and moq was completed. A configuration dictionary has been created to set up the controller as the test files cannot see the “appsettings.json” file. This tested all the possible endpoints encompassed within Task A and B with valid and invalid data to ensure that everything was running correctly. Figure 45 shows an example of unit tests with valid and invalid data.

```
// Test total marks
[Fact]
0 references
public async void TestTotalMarks()
{
    // Arrange
    var inMemorySettings = new Dictionary<string, string> {
        {"ConnectionStrings:MaxMin", "http://qubgrademe-maxmin.40266405.qpc.hal.davecutting.uk/" },
        {"ConnectionStrings:SortedModules", "http://qubgrademe-sortmodules.40266405.qpc.hal.davecutting.uk/"},
        {"ConnectionStrings:ClassifyGrade", "https://qubgrademe-classifygrade.azurewebsites.net/api/qubgrademe-classifygrade"},
        {"ConnectionStrings:PercentNeededForFirst", "http://qubgrademe-percentneededforfirst.40266405.qpc.hal.davecutting.uk/api/QubGrademe_percentneededforfirst/getpercentneeded"},
        {"ConnectionStrings:TotalMarks", "http://qubgrademe-totalmarks.40266405.qpc.hal.davecutting.uk/totalmarks"},
        {"ConnectionStrings:AverageGrade", "http://qubgrademe-averagegrade.40266405.qpc.hal.davecutting.uk/averagegrade"},

        {"Modules:MaxMin", "true"},
        {"Modules:SortedModules", "true"},
        {"Modules:ClassifyGrade", "false"},
        {"Modules:PercentNeededForFirst", "false"},
        {"Modules:TotalMarks", "false"},
        {"Modules:AverageGrade", "false"},
    };

    IConfiguration configuration = new ConfigurationBuilder()
        .AddInMemoryCollection(inMemorySettings)
        .Build();

    var controller = new qubgrademe_proxycontroller(configuration, _logger.Object);

    var proxyCommand = new ProxyCommand
    {
        ConnectionString = "TotalMarks",
        Module1 = "CSC-3065",
        Module2 = "CSC-3061",
        Module3 = "CSC-3067",
        Module4 = "CSC-3068",
        Module5 = "CSC-3062",
        Mark1 = "78",
        Mark2 = "75",
        Mark3 = "79",
        Mark4 = "80",
        Mark5 = "75"
    };

    // Act
    var result = await controller.getResponseFromService(proxyCommand);

    // Assert
    OkObjectResult? okResult = result.Result as OkObjectResult;

    string shouldBe = "{\"message\":\"Your total marks are: 387";

    result.Should().NotBeNull();
    result.Should().BeOfType<ActionResult<string>>();
    okResult.Should().BeOfType<OkObjectResult>();
    okResult.Value.ToString().Contains(shouldBe).Should().BeTrue();
}
```

```
// Test wrong connection string
[Fact]
0 references
public async void TestInvalidConnectionString()
{
    // Arrange
    var inMemorySettings = new Dictionary<string, string> {
        {"ConnectionStrings:MaxMin", "http://qubgrademe-maxmin.40266405.qpc.hal.davecutting.uk/" },
        {"ConnectionStrings:SortedModules", "http://qubgrademe-sortmodules.40266405.qpc.hal.davecutting.uk/"},
        {"ConnectionStrings:ClassifyGrade", "https://qubgrademe-classifygrade.azurewebsites.net/api/qubgrademe-classifygrade"},
        {"ConnectionStrings:PercentNeededForFirst", "http://qubgrademe-percentneededforfirst.40266405.qpc.hal.davecutting.uk/api/QubGrademe_percentneededforfirst/getpercentneeded"},
        {"ConnectionStrings:TotalMarks", "http://qubgrademe-totalmarks.40266405.qpc.hal.davecutting.uk/totalmarks"},
        {"ConnectionStrings:AverageGrade", "http://qubgrademe-averagegrade.40266405.qpc.hal.davecutting.uk/averagegrade"},

        {"Modules:MaxMin", "true"},
        {"Modules:SortedModules", "true"},
        {"Modules:ClassifyGrade", "false"},
        {"Modules:PercentNeededForFirst", "false"},
        {"Modules:TotalMarks", "false"},
        {"Modules:AverageGrade", "false"},
    };

    IConfiguration configuration = new ConfigurationBuilder()
        .AddInMemoryCollection(inMemorySettings)
        .Build();

    var controller = new qubgrademe_proxycontroller(configuration, _logger.Object);

    var proxyCommand = new ProxyCommand
    {
        ConnectionString = "AverageGradeasdf",
        Module1 = "CSC-3065",
        Module2 = "CSC-3061",
        Module3 = "CSC-3067",
        Module4 = "CSC-3068",
        Module5 = "CSC-3062",
        Mark1 = "78",
        Mark2 = "75",
        Mark3 = "79",
        Mark4 = "80",
        Mark5 = "75"
    };

    // Act
    var result = await controller.getResponseFromService(proxyCommand);

    // Assert
    BadRequestObjectResult? okResult = result.Result as BadRequestObjectResult;

    string shouldBe = "This endpoint is invalid";

    result.Should().NotBeNull();
    result.Should().BeOfType<ActionResult<string>>();
    okResult.Should().BeOfType<BadRequestObjectResult>();
    okResult.Value.Should().Be(shouldBe);
}
```

Figure 45: Example of a Valid and Invalid Unit Test for the Proxy

Brief Review of Success:

Figure 44 shows an example of the proxy succeeding at routing the request to the average grade API, the video shows the rest of the microservices being called upon correctly. A “gitlab.ci-yml” file has been written for CI testing to run all the unit tests on the QPC GitLab runner and figure 46 shows all thirteen of these tests passing.

```
Passed!  - Failed:     0, Passed:    13, Skipped:     0, Total:    13, Duration: 1 s - /builds/40266405/qubgrademe-proxy/qubgrademe-proxy.test/bin/Debug/net6.0/qubgrademe-proxy.test.dll (net6.0)
Job succeeded
```

Pipeline #79910 passed with stages in 2 minutes and 29 seconds

Figure 46: Pipeline and Unit Testing Results

Anything else to highlight:

The “gitlab-ci.yml” also includes a build and deploy job, the build job ensures that the application can compile correctly and there are no syntax errors. The deploy job then uses the “docker:latest” image to build the image, login to the QPC Gitlab container registry, and then pushes the new image to the container registry. Figure 46 shows this pipeline passing and figure 47 shows the “gitlab-ci.yml”.

```yaml
# Uses the c# dotnet docker image
image: mcr.microsoft.com/dotnet/sdk:6.0

# Build the project and restore dependencies
build:
  stage: build
  script:
    - 'dotnet build ./qubgrademe-proxy'

# Test the project
test:
  stage: test
  script:
    - 'dotnet test ./qubgrademe-proxy.test'

# Build the new docker image and push it up to the QPC container registry
deploy:
  image: docker:latest
  stage: deploy
  tags:
    - "dind"
  script:
    - cd ./qubgrademe-proxy
    - docker build -t registry.hal.davecutting.uk/40266405/qubgrademe-proxy .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker push registry.hal.davecutting.uk/40266405/qubgrademe-proxy
```

Figure 47: CI/CD File for the Proxy

Service Discovery Implementation

Service discovery is implemented on the proxy by having two extra endpoints to update and add to the service registry (the environment variables stored on Rancher). The microservices can call up these endpoints to update the URLs so that the proxy knows exactly how to connect to them and retrieve data from them.

Adding to the service registry can be done by calling the URL + “/addtoserviceregistry” plus the DNSName (the mapping) and the new URL for said mapping in the body. If the mapping already exists in the registry, then it will return a bad request, but if not, then the new mapping will be added to the registry. Figure 48 shows the code for this and figure 49 shows an example call which tries to add a mapping that already exists in the registry.

```
[HttpPost("/addToServiceRegistry", Name = "AddToServiceRegistry")]
[ProducesResponseType((int)HttpStatusCode.OK)]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
0 references
public ActionResult<string> AddToServiceRegistry([FromBody] URLMapping url)
{
    _logger.LogInformation("Adding URL for " + url.dnsName);

    _logger.LogInformation("ConnectionStrings:" + url.dnsName);
    var endpoint = _configuration["ConnectionStrings:" + url.dnsName];

    // If the endpoint exists then don't add it again
    if (endpoint != null)
    {
        return BadRequest("This endpoint already exists in the service registry");
    }

    // Create the endpoint in the service registry
    _configuration["ConnectionStrings:" + url.dnsName] = url.url;

    // Returning an OK message
    return Ok("The URL for " + url.dnsName + " has been successfully added to the service registry");
}
```

Figure 48: Adding the Endpoint to the Service Registry

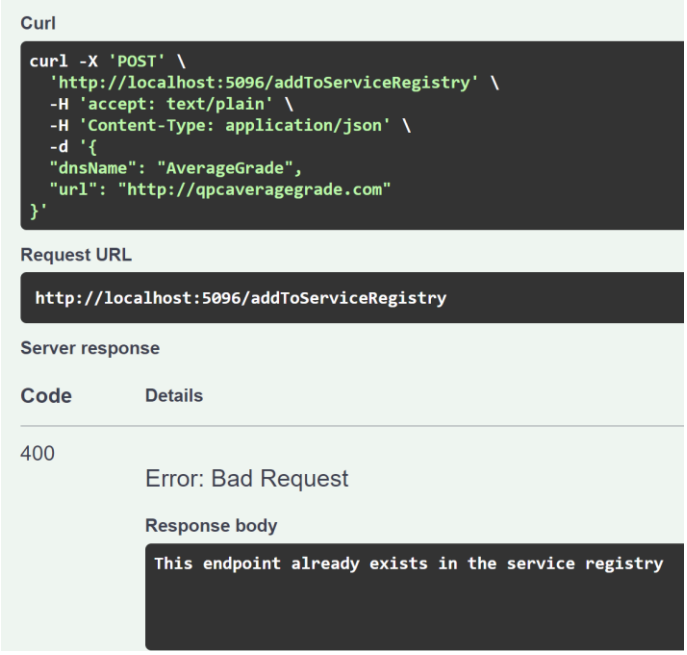


Figure 49: Example of Adding an Already Existing Mapping to the Service Registry

Updating the service is done in a similar way, however the validation to check to see if it exists is different because the URL should only be updated if the mapping already exists. Figure 50 shows how this is done in the code and figure 51 shows it working in swagger to update the mapping for the average grade.

```
[HttpPut("/updateServiceRegistry", Name = "UpdateServiceRegistry")]
[ProducesResponseType((int)HttpStatusCode.OK)]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
0 references
public ActionResult<string> UpdateServiceRegistry([FromBody] URLMapping url)
{
    _logger.LogInformation("Updating URL for " + url.dnsName);

    var endpoint = _configuration["ConnectionStrings:" + url.dnsName];

    // If the endpoint exists then update that endpoint
    if (endpoint != null)
    {
        // Create the endpoint in the service registry
        _configuration["ConnectionStrings:" + url.dnsName] = url.url;

        // Returning an OK message
        return Ok("The URL for " + url.dnsName + " has been successfully updated in the service registry");
    }

    return BadRequest("This endpoint doesn't yet exist in the service registry so can't be updated");
}
```

Figure 50: Updating the Endpoint in The Service Registry

Figure 51: Example of Updating a Mapping in the Service Registry

Task D: Frontend Failure Handler

Proxy backup URL: http://qubgrademe-proxy-backup.40266405.qpc.hal.davecutting.uk/

Implementation Details:

The front-end failure handler has been implemented into the front end to ensure that there is always a backup proxy for the front end to use and the application is never down. In rancher there are two workloads for the proxy using the same image and two ingresses with two different hosts. These function as the proxy and proxy backup. The front end has a “config.json” which stores these proxy URLs in an array as seen in figure 32. Task B and figure 33 explains how the array of proxy URLs is retrieved from the config file.

Whenever a call to the proxy is being completed, the front-end loops through this array and tries to fetch data from the proxy, if the proxy does not respond and/or times out then a console.log is called to say that the backup is being called and it calls the backup. If the proxy responds, the front end breaks out of this loop and shows the results to the user. If all microservices are down, then an alert is displayed to the user and the application returns from that function. An example of this code can be seen in figure 52.

```
var messageReceivedFlag = false;

for(var i = 0; i < proxyArray.length; i++)
{
    let url = proxyArray[i] + "?ConnectionString=TotalMarks" + "&mark1=" + mark_1 + "&mark2=" + mark_2
            + "&mark3=" + mark_3 + "&mark4=" + mark_4 + "&mark5=" + mark_5;

    console.log(url);

    await fetch(url, {
    method: 'GET',
    headers: {
    },
    }).then((response) => response.json())
    .then((data) => {
        messageReceivedFlag = true;
        displayTotal(data.message);
        proxyArray.reverse();
    }).catch(err => {
        console.log("Calling backup");
    });

    if(messageReceivedFlag)
    {
        break;
    }
}

if(!messageReceivedFlag)
{
    alert("All microservices are down");
    return;
}
```

Figure 52: Front End Failure Handler Loop for the Total Marks Microservice Call

Testing Details:

The testing for the front-end failure handler included bringing down the proxy ingress to force the webpage to use the backup. It could be seen in the console that it would be said the backup proxy was being called and it would take longer for the request to fulfil. This testing was done with all microservices including valid and invalid data (had to turn off the front-end validation check to allow this).

Brief Review of Success:

The testing as explained in the section above was successful with the application effectively calling the back-up proxy whenever the main proxy was down. Figure 53 shows a call to the classification microservice when the main proxy is down, the correct classification is retrieved and as seen in the networking tab, the first call times out and the backup proxy is called.

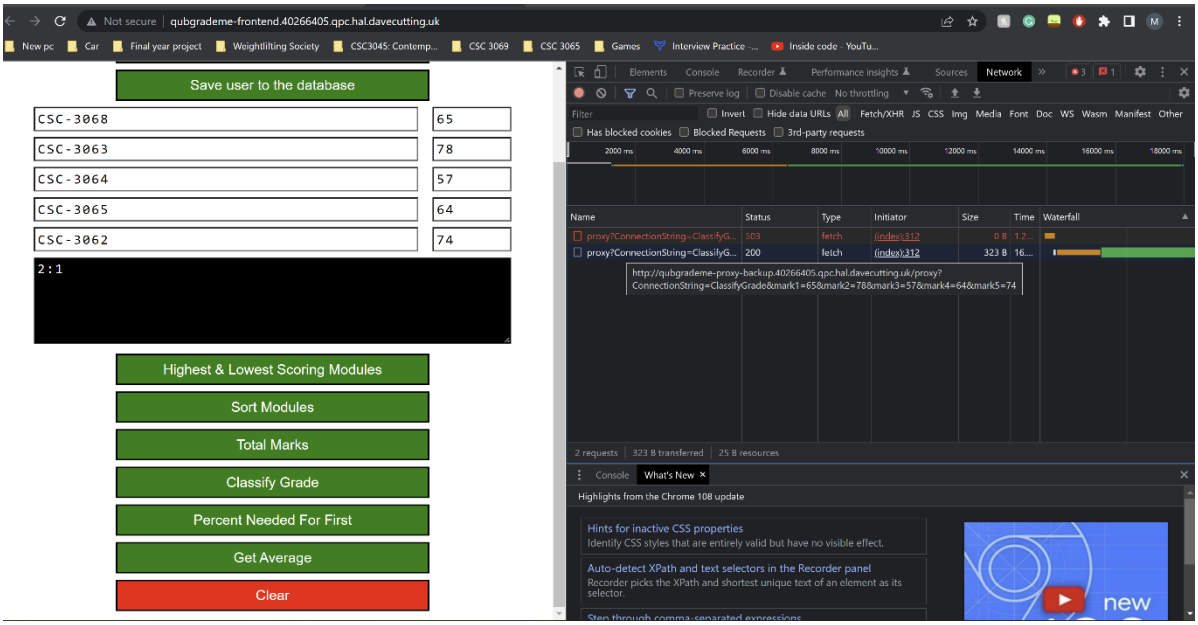


Figure 53: Front End Failure Handler Working and Calling the Backup Proxy

Anything else to highlight (Load Balancing):

Load balancing has been implemented into the front end failure handler by reversing the array of proxies every time a request has been fetched. This ensures that every time the proxy is called, it always uses a different instance of the proxy and if there is a lot of load coming from the front end, this load is balanced between multiple instances of the proxy.

Task E: Monitoring

Monitoring repository URL: http://gitlab.hal.davecutting.uk/40266405/qubgrademe-monitoring
Monitoring live URL: http://qubgrademe-monitoring1.40266405.qpc.hal.davecutting.uk/

The monitoring microservice is a HTML, CSS and JS page deployed to QPC rancher using the PaaS (Platform as a Service) paradigm. The microservice uses a configuration file to set all the URLs just like the front end and fetches this data into variables. A Dockerfile is used to containerise the application and the “php:7.2-apache” image is used.

Monitoring the necessary metrics

This microservice monitors all microservices using multiple metrics, firstly it checks to see if the service is up and running, calculates latency of the microservices and finally checks the correctness of results. This is done all in a single fetch statement for each microservice and it is shown how this is done in figure 54 for the save to database microservice.

```
var startSaveToDatabase = Date.now();
var saveToDblatency;
await fetch(qubgrademesave, {
method: 'GET',
headers: {
    // 'Content-Type': 'application/json',
    'mode': 'no-cors',
},
}).then((response) => response.json())
.then((data) => {
    var flag = false;
    console.log(data);
    if(data.Message == "The username Michal has been updated in the database")
    {
        flag = true;
    }
    saveToDblatency = Date.now()-startSaveToDatabase;
    proxyLatencies.push(saveToDblatency);
    displaySaveLatency(saveToDblatency, flag);
}).catch(err => {
    console.log("Couldn't connect to microservice " + err);
    document.getElementById("save-paragraph").innerHTML = "The Microservice is down";
    sendEmail();
});
```

Figure 54: Fetch Statement Used to Calculate Latency and Correctness of Results

If the microservice returns a HTTP code of 200 and the fetch statements moves into the ".then" section it is assumed that the microservice is up and running. The message returned from the microservice is then checked against a pre written string to ensure the correctness of results. The latency is finally calculated by getting the time at the start of the fetch statement, and taking that away from the current time. The latency and the flag to see if the results are correct is sent to the display function (which exists for each check) and it updates the data on screen for if all the checks have passed.

The display latency functions are very similar for every microservice check, they display the latency, and dependent if the microservice is returning the correct result, it outputs the necessary message. Figure 55 shows how this is done.

```
function displayRetrieveLatency(latency, flag)
{
    console.log('Retrieve from database latency: ' + latency.toString() + 'ms');
    document.getElementById('latency-retrieve').value = 'Latency: ' + latency.toString() + 'ms';
    document.getElementById('retrieve-paragraph').innerHTML = 'The service is up and running';
    if(flag)
    {
        document.getElementById('test-retrieve').innerHTML = 'The microservice returns the correct value';
    }
    else {
        document.getElementById('test-retrieve').innerHTML = 'The microservice is returning an incorrect value';
    }
}
```

Figure 55: Function to Display the Latency and Correctness of Results

Before any of the results are fetched from the microservices, a wait message is displayed to let the user know that no results have been checked yet. Figure 56 shows how this webpage looks whenever the monitoring service boots up.

Figure 56: Monitoring Service Before the Results are Retrieved

Once the results have been retrieved, as seen in figure 57 all the boxes are updated with the latency and two messages to say if the microservice is up and running and if the data being returned from the microservice is correct.

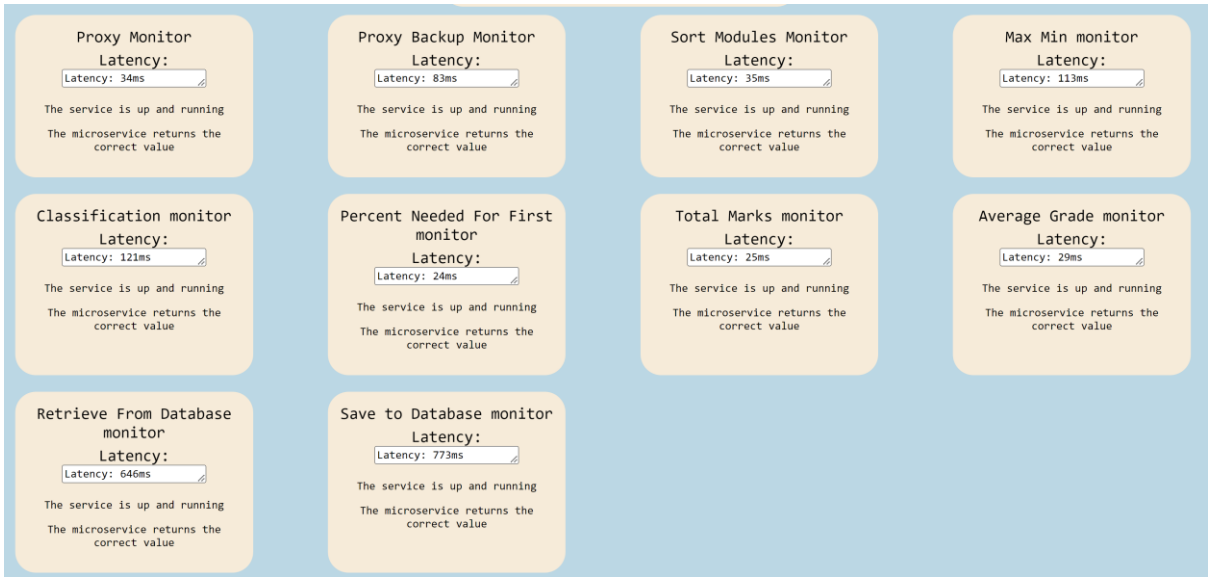


Figure 57: Monitoring Service Once the Data Has Been Retrieved

Periodically fetching new results

The action of retrieving this data is done periodically every ten minutes to ensure that these latencies are updated regularly. This is possible to achieve because getting all the data from the microservices is inside one function. It is called using the JavaScript set interval function which retrieves the data infinitely every 10 minutes. Figure 58 shows how this is done in the code.

```
setInterval(UpdateLatency, 600000);
```

Figure 58: Periodically Sending Requests to Retrieve Data

Fetching Results on Demand

There is a button in the monitoring service to allow the system administrator to run these tests on demand if there are any issues occurring at that time. Figure 58 shows this button with the networking tab open to show all the requests being run consecutively.

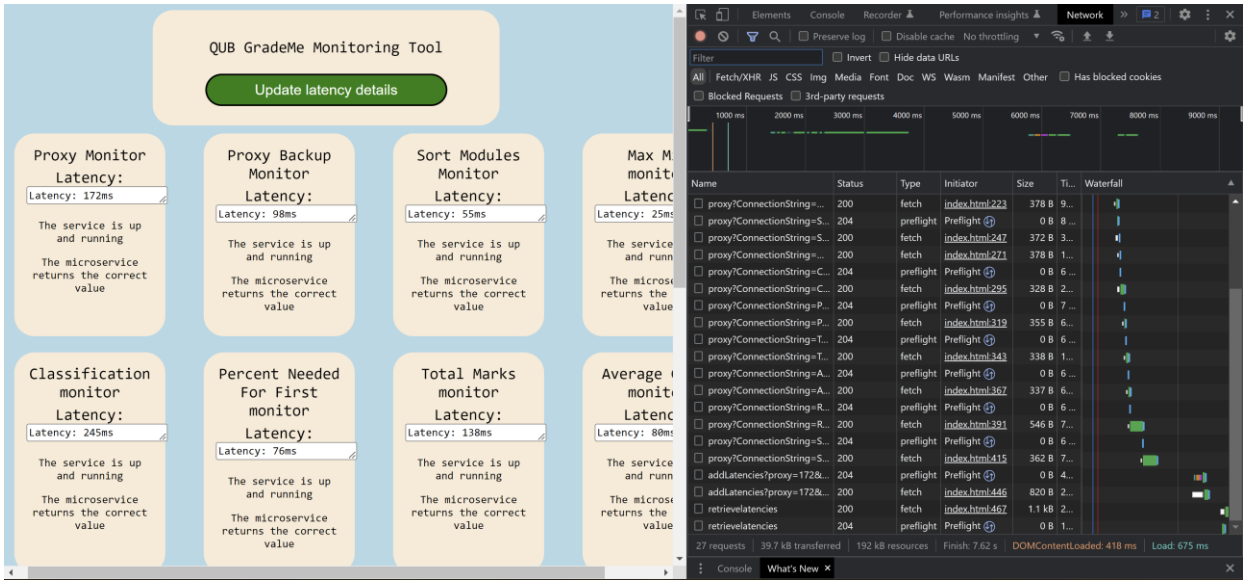


Figure 59: All Network Traffic After Pressing the Update Latency Details Button

If an error is thrown by any microservice or the monitoring system cannot connect to one of the microservices (a HTTP 400 code, bad request counts as an error), the exception message is logged in the console and an email is sent to the system administrator to alert them of this occurrence. Figure 60 shows how this would look on the webpage for the user.

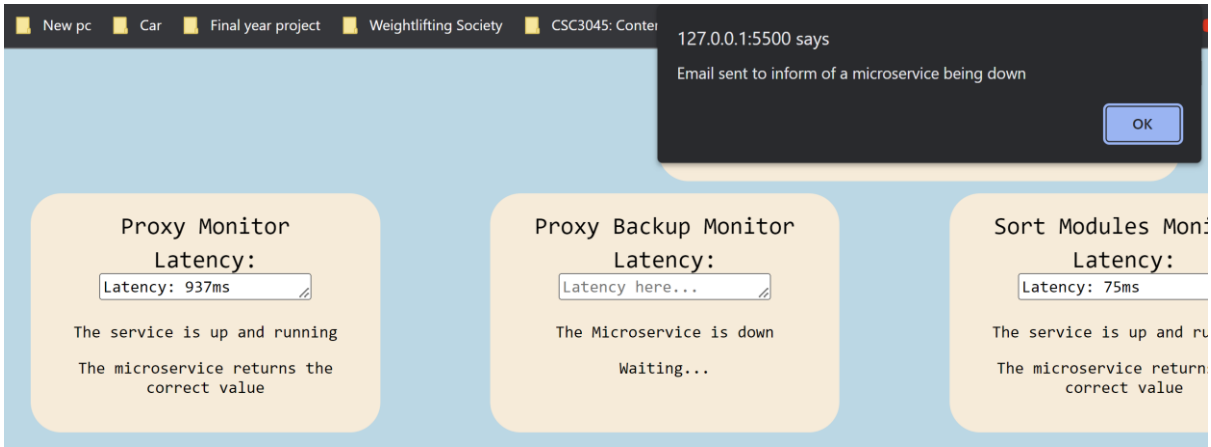


Figure 60: Example of What Happens When the Proxy Backup is not Responding

Alerting system to send email on failure

An account has been created on elasticemail.com so that its SMTP server can be used to relay emails onto the system administrator. The password is currently stored in the code, but in the future this should be stored in an encrypted configuration file or on Azure Key Vault. As seen in figure 61, the host used is “smtp.elasticemail.com” and the username and password combo was generated by elastic email. An account had to be created on elastic email to use their SMTP server and the emails are sent over that host. “Smtpjs” was used for the library to send emails to the user and an alert is shown to the screen whenever an email is sent.

```
<script src="https://smtpjs.com/v3/smtp.js"></script>
Email.send({
    Host : "smtp.elasticemail.com",
    Port : "2525",
    Username : "michalguzyburner@gmail.com",
    Password : "88E1E3A2316941AEE104630725459DD31BE1",
    To : 'michalguzyburner@gmail.com',
    From : "michalguzyburner@gmail.com",
    Subject : "A Microservice is down",
    Body : "<html><<strong>Please check the monitoring microservice, one of the microservices is down</strong><br></br><em></em></html>"
}).then(
message => alert("Email sent to inform of a microservice being down")
);
```

Figure 61: Code to Send an Email to the System Administrator

An example of an email which has been sent can be seen in figure 62.

Figure 62: Example Email Sent to the System Administrator

Storage of latencies in the database and graphing them on the monitoring microservice

Once all the results and latencies have been retrieved, the results are then collated and saved to the database which is hosted on Azure. A fetch statement is used to call the stateful saving web API which is hosted on Azure and explained in further detail in section F. Once the data is saved in the database, all the latencies for the given day are retrieved using another fetch statement and the data is collated into arrays of latencies for each microservices as seen in figure 63.

```
// Labels for the times
var labels = [];
var proxyData = [];
var proxyBackupData = [];
var averageGradeData = [];
var maxMinData = [];
var percentNeededForFirstData = [];
var retrieveFromDbData = [];
var saveToDbData = [];
var sortedModulesData = [];
var totalMarksData = [];

// Generating the arrays for the graph
for(var i = 0; i < arrayOfLatencies.length; i++)
{
    labels.push(arrayOfLatencies[i].dateSaved);
    proxyData.push(arrayOfLatencies[i].proxy);
    proxyBackupData.push(arrayOfLatencies[i].proxyBackup);
    averageGradeData.push(arrayOfLatencies[i].averageGrade);
    maxMinData.push(arrayOfLatencies[i].maxMin);
    percentNeededForFirstData.push(arrayOfLatencies[i].percentNeededForFirst);
    retrieveFromDbData.push(arrayOfLatencies[i].retrieve);
    saveToDbData.push(arrayOfLatencies[i].saveToDb);
    sortedModulesData.push(arrayOfLatencies[i].sortedModules);
    totalMarksData.push(arrayOfLatencies[i].totalMarks);
}
```

Figure 63: Collating all the Latencies Stored for that Day into an Array for Each Microservice

These arrays are then used to generate a chart on the monitoring web service which illustrates all the latencies for the given day. The Chart.JS library is used to accomplish this task and figure 64 shows a snippet for a part of the code of how this is done.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.5.0/Chart.min.js"></script>

// Drawing the chart
new Chart(document.getElementById("line-chart"), {
    type: 'line',
    data: {
        labels: labels,
        datasets: [{
            data: proxyData,
            label: "Proxy",
            borderColor: "#3e95cd",
            fill: false
        }, {
            data: proxyBackupData,
            label: "Proxy Backup",
            borderColor: "#8e5ea2",
            fill: false
        }, {
            data: averageGradeData,
            label: "Average Grade",
            borderColor: "#3cba9f",
            fill: false
        }, {
            data: maxMinData,
            label: "Max Min",
            borderColor: "#e8c3b9",
            fill: false

<canvas id="line-chart" width="550" height="250"></canvas>
```

Figure 64: Snippet of code for Generating the Chart to Illustrate Latencies

To ensure that data is not misread from the chart, whenever the microservice is booted up the chart is not shown. Once all the data from that day has been retrieved from the database then the chart appears on the screen with a different coloured line for every microservice that is checked by the monitoring microservice. The chart also updates every time that new latencies are retrieved so the system admin can see the newest data live. Figure 65 illustrates the graph on the monitoring screen and how it would look to the user once quite a lot of data points have been retrieved. If it is the first time booting up the monitoring microservice for that day then the chart will be empty, you need to either wait until it updates after multiple ten minute intervals or press the check latency button multiple times and the graph will update in real time.

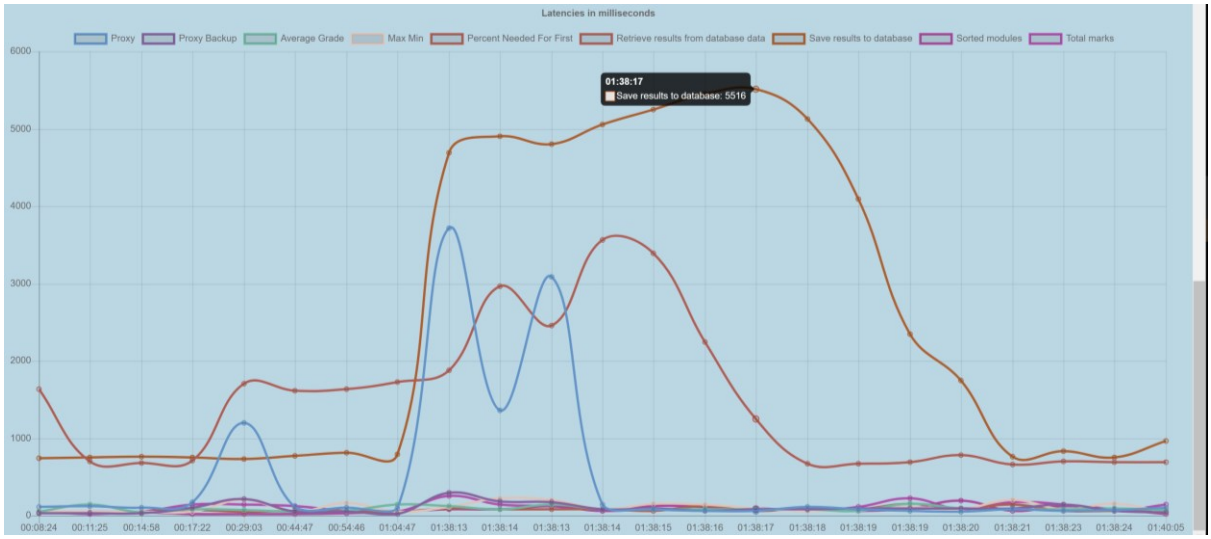


Figure 65: Monitoring Data Illustrated as a Chart in the Monitoring Microservice

Continuous deployment is implemented on the monitoring service to ensure that every time new code is pushed to the GitLab repository a new docker image is built and pushed to the container registry. Figure 66 shows the ".gitlab-ci.yml" which has been written to use the "docker:latest" image and figure 67 shows the pipeline working on the QPC GitLab.

```
# Deploying the new image to the QPC container registry
deploy:
  image: docker:latest
  stage: deploy
  tags:
    - "dind"
  script:
    - docker build -t registry.hal.davecutting.uk/40266405/qubgrademe-monitoring .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker push registry.hal.davecutting.uk/40266405/qubgrademe-monitoring
```

Figure 66: CD File for the Monitoring Microservice

```
latest: digest: sha256:6e71c023028020e4587e3a0eba8ebba5c08f34f7a269f4f3e37da3ff597a093e size: 3450
Job succeeded
```

Pipeline #82012 passed with stage in 9 seconds

Figure 67: Deploy Pipeline Results for the Monitoring Microservice

Task F: Stateful Saving

Link for stateful saving API: https://qubgrademe-savetodatabase20221128232120.azurewebsites.net/
Link for stateful saving code repository:
http://gitlab.hal.davecutting.uk/40266405/qubgrademe-savetodatabase
Link to azure database:
https://portal.azure.com/#@qubstudentcloud.onmicrosoft.com/resource/subscriptions/eb6309b4-c27d-4cbd-b993-e1c6c90d6ef0/resourceGroups/qubgrademedatabase/providers/Microsoft.Sql/servers/qubgrademe-database/databases/qubgrademedatabase/overview

Stateful saving has been implemented with two main components, a C# Web API deployed as an app service to Microsoft Azure and an Azure SQL Database which is also hosted on Microsoft Azure. The app service uses the ASP.Net Core framework with .Net version 6.0 using the PaaS (Platform as a Service) paradigm and the database uses T-SQL statements to create, read, update and delete data. Saving and retrieving has been implemented for the results in the front end, and the latencies for the monitoring part of the system.

The stateful saving app service has five endpoints used for all the logic of the system and the first endpoint can be retrieved by calling the URL + "/". This welcomes the user onto the app service with a message to let them know they have the correct URL and always returns a status code of 200 HTTP Ok.

Storing and retrieving results data

The next two endpoints are used to retrieve and save data from the front end, all marks, modules and results are retrieved whenever this endpoint is called. The data is displayed to the user in a user friendly manner and a username that is typed in by the user is used as an identifier for retrieving results. Figure 68 shows how these results are displayed on the front end, when they are retrieved the marks and modules all go into the input fields for the user and the results are nicely shown in the black results box in the middle of the screen. The username is also displayed to let the user know that the correct results have been retrieved.

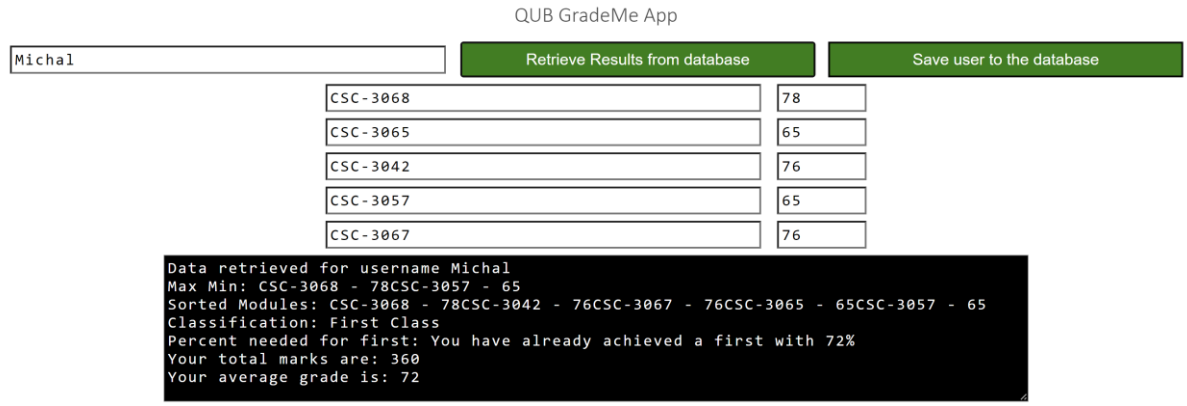


Figure 68: Data Retrieved From the Database Being Shown on the Front End

On the back-end side of retrieving results from the user the retrieve endpoint is called by calling the URL + "/retrieveresults" with the username as a parameter in the format of "?username= <username>". An example of a valid URL call for this has been shown below:

https://qubgrademe-savetodatabase20221128232120.azurewebsites.net/retrieveresults?username=Michal

If the username does not yet have any data in the database, a Bad Request HTTP code of 400 is returned to the user with a customised message saying that said username does not yet exist. An example of a URL that would return this message has been shown below:

https://qubgrademe-savetodatabase20221128232120.azurewebsites.net/retrieveresults?username=Caitlyn123

This happens because, as seen in figure 69 the SQL query to retrieve data from the database selects all columns where the username is equal to the username in the parameter. So, if no results are returned the bad request message can be returned to the user. An SQLConnection object has been used to connect to the database using a pre-written connection string retrieved from the Azure Portal, and the password is stored in the configuration for security reasons. If the SQLConnection object throws an exception, for example the connection to the database is refused then this error message is returned as a Bad Request HTTP code 400 and relayed back to the user that called the API. Finally, if all these checks pass then an Ok HTTP Code 200 is returned to the user with all the details stored in the database for that username.

```
        using (SqlConnection connection = new SqlConnection(connectionString))
        {

            String sql = $"SELECT * FROM [dbo].[Users] WHERE [UserName]= '{request.userName}'";

            using (SqlCommand command = new SqlCommand(sql, connection))
            {
                connection.Open();
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        userFromDb.userName = reader.GetString(1);
                        userFromDb.module1 = reader.GetString(2);
                        userFromDb.module2 = reader.GetString(3);
                        userFromDb.module3 = reader.GetString(4);
                        userFromDb.module4 = reader.GetString(5);
                        userFromDb.module5 = reader.GetString(6);
                        userFromDb.mark1 = reader.GetString(7);
                        userFromDb.mark2 = reader.GetString(8);
                        userFromDb.mark3 = reader.GetString(9);
                        userFromDb.mark4 = reader.GetString(10);
                        userFromDb.mark5 = reader.GetString(11);
                        userFromDb.results = reader.GetString(12);
                    }
                }
            }
        }
    catch (SqlException e)
    {
        Console.WriteLine(e.ToString());
        return BadRequest(e.ToString());
    }

    if(userFromDb.userName == "")
    {
        return BadRequest("{\"Message\":\"The username " + request.userName +" doesn't have any values in the database\"}");
    }

    return Ok(userFromDb);
```

Figure 69: Retrieving Results for a Username in the C# App Service

Saving the results into the database also includes validation to ensure that bad data is not saved into the database. The username is validated to check if it is empty or not. The modules are validated to check if they are empty or not, the marks are checked to see if they are empty, an integer or in the range of 0 to 100, and finally the results are checked to see if they are empty or null. Figure 70 shows how this validation is done in the C# app service for each of the types of parameters.

```
for(int i = 0; i < listOfModules.Count(); i++)
{
    if (listOfModules[i].Trim() == "" || listOfModules[i] == null)
    {
        return BadRequest($"Module {i + 1} is either null or blank and can't be empty");
    }
}

// Checking if the marks are integers
if (!Int32.TryParse(request.mark1, out intmark1))
{
    return BadRequest("{\"Message\": \"Module 1 needs to be an integer \"}");
}
// Checking if the values are >=0 and <=100
if (intmark1 < 0 || intmark1 > 100)
{
    return BadRequest("{\"Message\": \"Module 1 needs to be <= 100 or >=0 \"}");
}
for(int i = 0; i < resultsList.Count; i++)
{
    if (resultsList[i].Trim() == "" || resultsList[i] == null)
    {
        return BadRequest($"Result {i + 1} is either null or blank and can't be empty");
    }
}
```

Figure 70: Validation for All Parameters Before Saving to the Database

The results are stored in the table under one column as a string separated with “\n” so that when they are displayed for the user in the front end the results are well formatted. Figure 71 shows a part of this result string being generated.

```
// Generating the results string
var resultsString = "Max Min: " + request.result1 + "\n" + "Sorted Modules: " + request.result2 + "\n" + "Classification: " + request.result3
```

Figure 71: A Part of the Result String Being Saved into the Database Being Generated

Once all the parameters have been validated it is checked to see if the username already exists in the database in the same way that it is checked in fetching the results, if the username already exists then the details are updated for the user as seen in figure 72. If the username does not have a record already in the database a new record is created for the user, and this is illustrated in figure 73.

```
// If the username already exists in the database, update the record that is there
if (userFromDb.userName != "")
{
    try
    {
        string connectionString = $"Server=tcp:qubgrademe-database.database.windows.net,1433;Initial Catalog=qubgrademedatabase;Persist Security Info=False;User ID=qubgrademe-database;Password={_confi

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            // command.CommandText = "UPDATE Student SET Address = @add, City = @cit Where FirstName = @fn and LastName = @add";

            String sql = $"UPDATE [dbo].[Users] SET [UserName] = '{request.userName}', [Module1] = '{request.module1}', [Module2] = '{request.module2}', [Module3] = '{request.module3}', [Module4] = '{

            using (SqlCommand command = new SqlCommand(sql, connection))
            {

                connection.Open();
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        _logger.LogInformation("{0} {1}", reader.GetString(0), reader.GetString(1));
                    }
                }
            }
        }
    }
    catch (SqlException e)
    {
        Console.WriteLine(e.ToString());
        return BadRequest(e.ToString());
    }

    return Ok("{\"Message\":\"The username " + request.userName + " has been updated in the database\"}");
}
```

Figure 72: Updating a User's Details in the Database on the Username

```
try
{
    string connectionString = $"Server=tcp:qubgrademe-database.database.windows.net,1433;Initial Catalog=qubgrademedatabase;Persist Security Info=False;User ID=qubgrademe-database;Password={_configuration[

    using (SqlConnection connection = new SqlConnection(connectionString))
    {

        String sql = $"INSERT INTO[dbo].[Users] ([UserName], [Module1], [Module2], [Module3], [Module4], [Module5], [Mark1], [Mark2], [Mark3], [Mark4], [Mark5], [ResultString]) VALUES('{request.userName}',

        using (SqlCommand command = new SqlCommand(sql, connection))
        {

            connection.Open();
            using (SqlDataReader reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    _logger.LogInformation("{0} {1}", reader.GetString(0), reader.GetString(1));
                }
            }
        }
    }
}
catch (SqlException e)
{
    Console.WriteLine(e.ToString());
    return BadRequest(e.ToString());
}

return Ok("{\"Message\":\"The username " + request.userName +" Saved user to the database\"}");
```

Figure 73: Inserting a New User Into the Database

Dependent on what occurs, a bad request is returned if an exception is thrown with the exception details, and an ok response is returned if everything succeeded to tell the user if the record has been updated or a new one has been created.

Storing and retrieving latencies data

Retrieving the data for latencies is done in a similar way as retrieving data for the user results, but instead of the username, the select statement checks against the date that the latency was saved in

the database and it only retrieves the values for that same day. The retrieve latencies function can be accessed with the URL + "/retrievelatencies" and an example URL has been shown below: (The return from this will be empty if the monitoring microservice has not been running yet on that day as it only retrieves the data from that day)

https://qubgrademe-savetodatabase20221128232120.azurewebsites.net/retrievelatencies

Figure 74 shows how the retrieving of these latencies is done in C#, there is a latencies object which stores each instance of all the latencies and all of them are added to an array of latency objects, which stores the date saved, the time saved and all the latencies for every microservice. This array is then returned to the front end so that the data can be shown in a graph as seen in figure 65.

```
var listOfLatencies = new List<AddLatencyRequest>();

try
{
    string connectionString = $"Server=tcp:qubgrademe-database.database.windows.net,1433;Initial Catalog=qubgrademedatabase;Persist S

    using (SqlConnection connection = new SqlConnection(connectionString))
    {

        String sql = $"SELECT * FROM [dbo].[Latencies] WHERE [DateSaved] = '{DateTime.Now.ToString("MM-dd-yyyy")}';";

        using (SqlCommand command = new SqlCommand(sql, connection))
        {
            connection.Open();
            using (SqlDataReader reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    var latenciesFromDb = new AddLatencyRequest();
                    latenciesFromDb.Proxy = reader.GetInt32(1);
                    latenciesFromDb.ProxyBackup = reader.GetInt32(2);
                    latenciesFromDb.MaxMin = reader.GetInt32(3);
                    latenciesFromDb.SortedModules = reader.GetInt32(4);
                    latenciesFromDb.ClassifyGrade = reader.GetInt32(5);
                    latenciesFromDb.PercentNeededForFirst = reader.GetInt32(6);
                    latenciesFromDb.TotalMarks = reader.GetInt32(7);
                    latenciesFromDb.AverageGrade = reader.GetInt32(8);
                    latenciesFromDb.Retrieve = reader.GetInt32(9);
                    latenciesFromDb.SaveToDb = reader.GetInt32(10);
                    latenciesFromDb.dateSaved = reader.GetString(12);
                    listOfLatencies.Add(latenciesFromDb);
                }
            }
        }
    }
}
catch (SqlException e)
{
    Console.WriteLine(e.ToString());
    return BadRequest(e.ToString());
}

return Ok(listOfLatencies);
```

Figure 74: Retrieving the Latencies for the Current Day as a List

Saving the latencies to the database is also done in a very similar way as saving results, the only things that change are the parameters which are saved in the insert statement. This can be seen being done in figure 75. If saving the latencies fails, then the exception is returned as a bad request to the user and if the latencies are saved correctly a message is returned with a HTTP Status code of 200. The endpoint for saving the latencies into the database can be written using the URL + "/addLatencies" and all the latencies as parameters.

```
// Endpoint for saving monitoring latencies
[Microsoft.AspNetCore.Mvc.HttpGet("/addLatencies", Name = "AddLatencies")]
[ProducesResponseType((int)HttpStatusCode.OK)]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
0 references
public ActionResult<string> AddLatencies([FromQuery] AddLatencyRequest request)
{
    try
    {
        string connectionString = $"Server=tcp:qubgrademe-database.database.windows.net,1433;Initial Catalog=qubgrademedatabase;Persist Security Info=False;User ID=

        using (SqlConnection connection = new SqlConnection(connectionString))
        {

            String sql = $"INSERT INTO[dbo].[Latencies] ([Proxy], [ProxyBackup], [MaxMin], [SortedModules], [ClassifyGrade], [PercentNeededForFirst], [TotalMarks],

            using (SqlCommand command = new SqlCommand(sql, connection))
            {

                connection.Open();
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        _logger.LogInformation("{0} {1}", reader.GetString(0), reader.GetString(1));
                    }
                }
            }
        }
    }
    catch (SqlException e)
    {
        Console.WriteLine(e.ToString());
        return BadRequest(e.ToString());
    }

    return Ok("{\"Message\":\"The latencies have been saved into the database\"}");
}
```

Figure 75: Adding Latencies into the Database

An example of a valid request for adding latencies has been shown below:

https://qubgrademe-savetodatabase20221128232120.azurewebsites.net/addLatencies?proxy=95&proxyBackup=62&maxMin=47&sortedModules=22&classifyGrade=279&percentNeededForFirst=55&totalMarks=853&averageGrade=55&retrieve=751&saveToDb=874&dateSaved=empty

How the database has been created

The database has been created using the Azure Portal and has been added into a "qubgrademedatabase" resource group. It uses SQL Authentication for signing in which works as a connection string as the one seen in figure 75. The Azure portal query editor was used to create both the Users table and the Latencies table within the database.

The users table includes a UserId and an identifier (username) as seen in figure 76 with all the rest of the necessary records. The latencies table uses the primary key (LatencyId) as the identifier and has values for each one of the microservices to store the latencies as seen in figure 77. The date and time saved is also stored to help query the database and show the correct data on the graph.

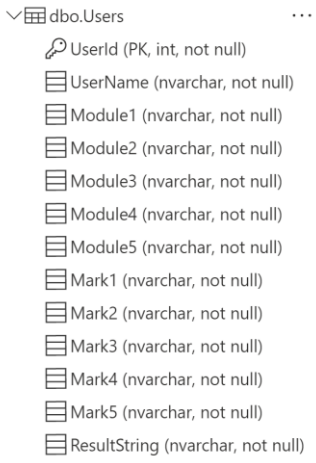


Figure 76: Users Table As Seen in the Azure Portal

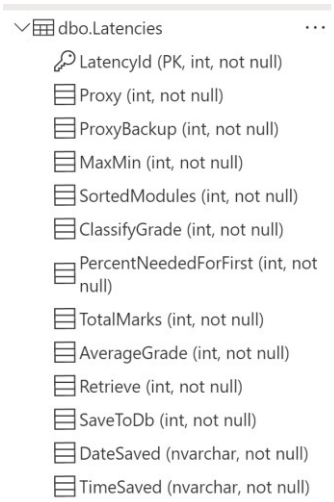


Figure 77: Latencies Table as seen in the Azure Portal

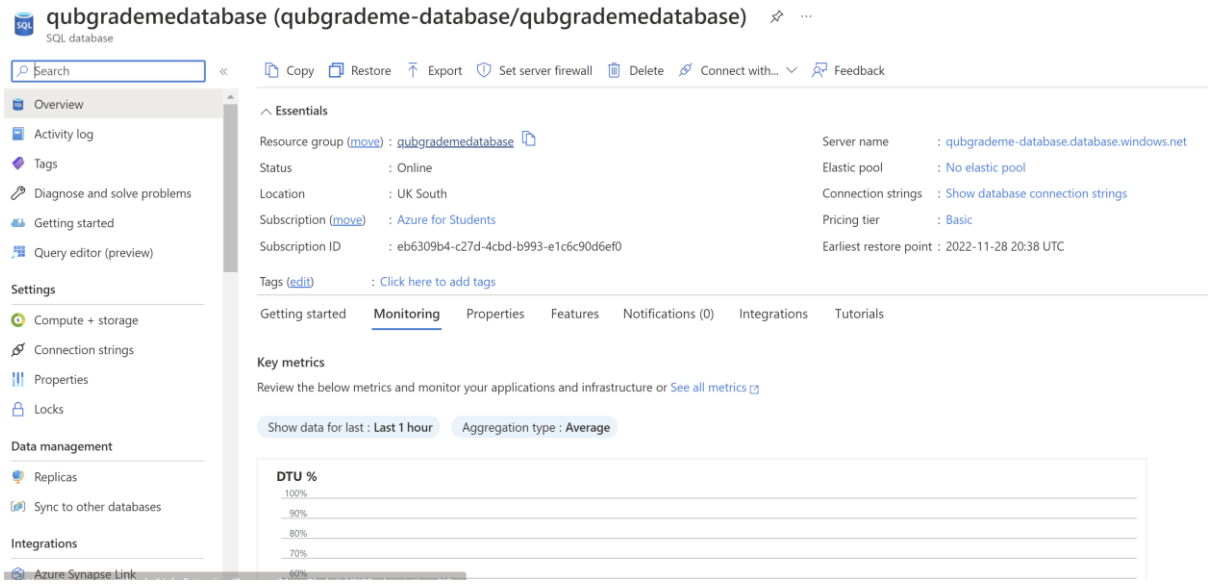


Figure 78: QUBGradeMe Database on Azure Portal

Task G: Design

Initial implementation using multi-cloud

The current QUBGradeMe application is a multi-cloud solution using both the Queen's Private Cloud (QPC) and Microsoft Azure. There are only two paradigms used within the current system and they are Function as a Service (FaaS) and Platform as a Service (PaaS). The classify grade function app deployed on Microsoft Azure makes use of the FaaS paradigm whereas the rest of the microservices make use of the PaaS paradigm. The main issue with on-premises cloud deployments like QPC is that even when there are no computing tasks running, electricity is still being used to power the servers which must be running constantly. The more microservices that are running on QPC, the more electricity is being used as more applications will be constantly listening for requests.

The QUBGradeMe application would have specific times like exam season in university where there would be much more load on the system, it would be extremely difficult to scale the system up and down automatically and to know when to scale the system up to ensure that the system is not overwhelmed by the exam season. The issue with this is that if the system was kept at full compute capability the whole time then a lot of electricity would be used by the system even though there would not be much load.

Figure 79 illustrates the current implementation and how the QPC is used with Microsoft Azure to implement the solution. All traffic from the front end is routed through the proxy which is shown by the black arrow coming from the front end. The green arrows show the proxy fetching requests from all the microservices, whereas the orange arrows illustrate the microservices updating the service registry with new URLs. The final connection is between the app service on Azure and the database, as only that app service can connect to the database. If the front end would like to retrieve any data it would have to go through the proxy, and then through the app service to get to the database.

All requests which enter Microsoft Azure must be routed through the Azure Firewall to ensure that the requests are valid and coming from a valid source, this drastically slows down the application and makes the user wait much longer for their requests to be fulfilled.

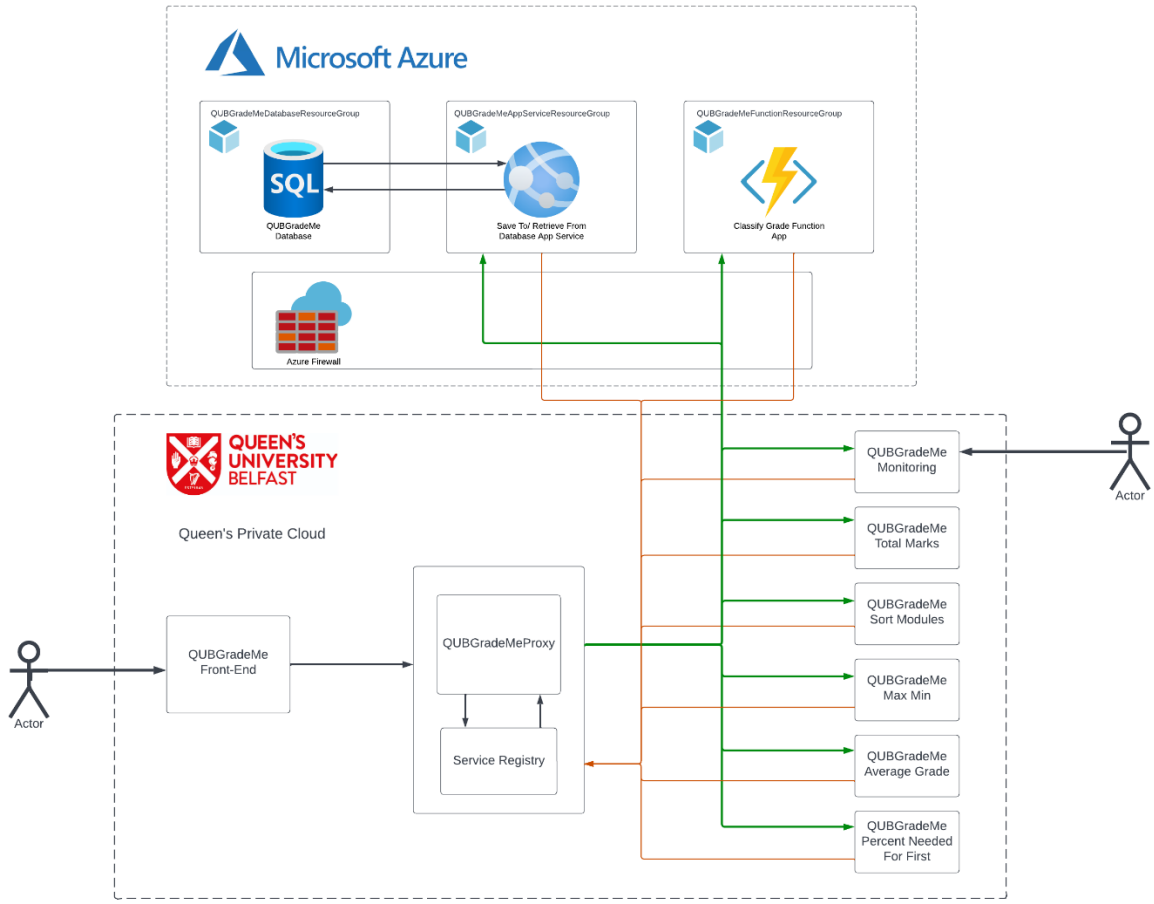


Figure 79: QUBGradeMe Initial Application Architecture Diagram

New Implementation for the QUBGradeMe Application

The new Implementation for the QUBGradeMe application has been illustrated as an architecture diagram in figure 80, the application has been moved completely off-premises onto the public cloud due to a multitude of reasons explained in this section. All the calculation services have been re written to make use of the FaaS paradigm and all the rest of the microservices now use Azure App services for their deployments. The service registry has also been changed into a database and the initial database has been split up into two databases for the latencies and user results.

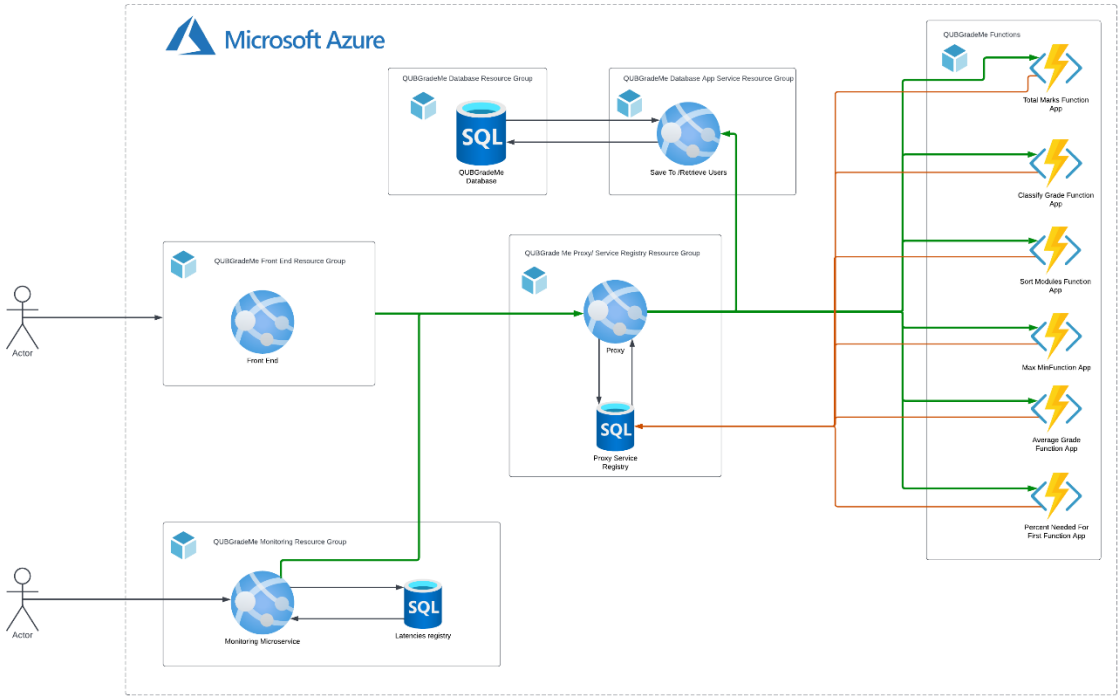


Figure 80: Architecture Diagram for the New Sustainable Implementation

Benefits of the new implementation

Moving the complete application completely onto the public cloud ensures that no initial investment must be made to create a private cloud and all infrastructure management has now been passed onto Microsoft so that the team can fully focus on implementing a robust application. Having all the microservices on Microsoft Azure also ensures that all requests must only go through the Firewall once and there is a much smaller chance of a man in the middle attack for requests between QPC and Azure.

Having all the applications on Azure also means that they can be scaled down drastically or completely turned off whenever students are not in university during reading weeks and breaks, if the microservices were deployed on premises then those servers would still be running the whole time using a lot more energy than if they were deployed on a part of a server in Microsoft Azure. Buying a large amount of physical hardware to run the servers on premises would also involve a lot of transportation of heavy goods which would release a lot of carbon dioxide and not be very sustainable.

Using FaaS instead of PaaS is more sustainable due to FaaS only running on a trigger basis. Container applications running on QPC are constantly running and listening for requests from users, this uses a

lot of electricity compared to functions. Functions completely stop if they have no requests, they only boot back up again once they are triggerd.

Microsoft Azure autoscaling allows for the application to always be available for any load necessary, but also using as little energy as possible whenever the application does not need to handle as much load. This would be complex and difficult to do with QPC if the programmers also had to manage their own infrastructure, but as Azure does this for the programmer this is one less worry.

Azure app services are used for all the HTML, CSS, JS webpages and the C# web API's including the proxy and stateful saving API. App services have been chosen as in this case the programmer will no longer need to worry about containerisation and all the applications will still act in the same way.

Azure key vault can be used to store secrets in an encrypted manner instead of having them in environment variables and/or configuration files, centralising all secrets makes them much easier to find and much easier to store securely.

Drawbacks of the new implementation

One of the main drawbacks to making a more sustainable application is the increase in latency between the user and the system. This means that the user would have to wait much longer for a response from the server. The average user of this application would be in or around QUB, so having the application run on-premises means the data has a much shorter distance to travel and the overall usability of the application becomes much better. But, when the application is deployed on Azure the closest server location would be in Cardiff (UK West), increasing the distance that the data must travel drastically.

Using FaaS instead of PaaS also decreases the performance of the application due to using triggers instead of constantly listening for requests. Even if the delay is very short, there is still a delay between a function receiving a request and being able to fulfil it because the function needs to boot back up between every request.

If the application was constantly running at full capacity, it would always be ready to serve as many requests as it needed to and would never have any performance drops. However, this is extremely unsustainable and would use a large amount of electricity. The new implementation uses the automatic Azure auto-scaler to scale down the application whenever load is not as large, this makes the application much more sustainable but if load drastically increases this may cause performance drops and the application may not be able to handle all the students using it at the same time.

Splitting up the databases increases security by making it more difficult to retrieve all the data in one go, but this also decreases the sustainability of the application as having more database instances would use more energy than having just one database instance.

Due to the new implementation using a lot of new paradigms and being completely on Microsoft Azure, a lot of code needs to be re written so that it works on Azure Functions. Completely different languages would have to be used for the PHP microservices as Azure Functions do not support PHP. This would make the move to this new implementation be very expensive and not worthwhile in an industry scenario.

Using Microsoft Azure also stops the programmers from being able to use a different cloud provider in the future as the code will only run on Azure correctly, a more cloud agnostic approach has been highlighted in the next section.

Using a more cloud agnostic approach

There are three main Azure components that would need to be replaced in the new implementation to develop a more cloud agnostic approach, the App Services, the Azure SQL Databases and the Azure Functions. The App Services can be very easily replaced with containerised applications which would be able to run on nearly any cloud provider, the Azure SQL database could be replaced for a containerised MySQL database instance. This would work with the code straight away so no issues would occur there. However, a lot of work would have to go into either re writing the functions to work with a new cloud provider, or a technology like Open Faas could be used to deploy the functions with most cloud providers being able to host such functions.