START Testing Doc

Below is a high-level overview of how we created and carried out unit testing of our web application. We will mention how we made use of Jest and Unittest packages for testing our frontend and backend respectively. We will also briefly highlight our CI/CD operations also.

Frontend Testing

Frontend testing was carried out using Jest. Jest is a JavaScript testing framework designed for testing JavaScript code. Jest is a Node.js module and is installed via `npm install`. It is important to note that our node_modules are included in our .gitignore file, therefore Jest would need to be installed if running tests on a new device. Upon installing Jest in our backend folder, we used the package.json file to define certain configuration settings for Jest.

```
{
  "name": "2024-ca400-kellyn88-graya27",
  "version": "1.0.0",
  "scripts": {
    "test": "jest"
  },
  "type": "module",
  "jest": {
    "testEnvironment": "jest-environment-jsdom",
    "transform": {
      "^.+\\.jsx?$": "babel-jest"
    },
    "testMatch": [
      "**/tests/**/*.js"
    ]
  },
  "devDependencies": {
    "jest": "^29.7.0",
    "jest-environment-jsdom": "^29.7.0",
    "jsdom": "^24.0.0",
    "mock-local-storage": "^1.1.24"
  }
}
```

The ‘testMatch’ tag is used so Jest can search for the test files. In this case, it will use any JavaScript files in the ‘tests’ folder to run its tests. The ‘testEnvironment’ tag defines the environment in which Jest should run its tests. ‘Jest-environment-jsdom’ allows Jest to simulate a browser-like environment without the need for the tests to run an actual browser, which is useful for our tests.

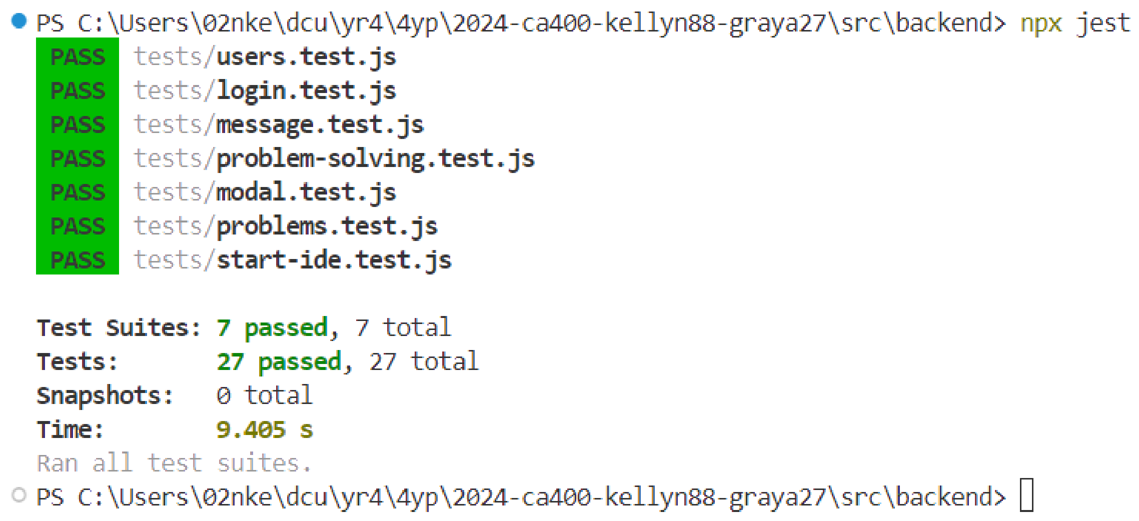
Our ‘tests’ folder includes seven Jest test files that tests 27 JavaScript units or functions across seven of our frontend JavaScript files located in our ‘static’ folder. Each Jest test sets up any necessary preconditions, then runs the target function and then a series of assertions to ensure the function carried out its expected actions. For example in this test, we set up a dummy ‘webpage’ for the ‘message’ function to target. We then run the function passing in arguments. We then run assertions to ensure the message has been added to the dummy page and that its content is correct.

```
test('message', () => {
    document.body.innerHTML = `
    <div id="wrapper"></div>
    `;

    message("success", "This is a success message");

    const messageElement = document.getElementById("message");
    expect(messageElement).not.toBeNull();
    expect(messageElement.classList.contains("message-success"))

    message("error", "This is an error message");
    const messageElement2 = document.getElementById("message");
    expect(messageElement2).not.toBeNull();
    expect(messageElement2.classList.contains("message-error"))

});
```

If all assertions pass, the test succeeds. Other tests are more complex as they test functions which call endpoints in the backend of the application. When we run Jest tests the backend is not active so we will not receive any responses when we call the endpoints. For this reason we create mock objects. For example in the below example we create a mock XMLHttpRequest object, with mock return values. For example now if an XMLHttpRequest object is used within our test the mock object will be used. `xhrMock.open` returns `jest.fn()` which is function that does nothing. `xhrMock.status` returns 200. With this implementation we can ensure the JavaScript function `getSessions()` works and calls the correct endpoint, without initialising our backend.

```
const sessionStorageMock = {
    getItem: jest.fn(),
    setItem: jest.fn()
};

test('getSessions', () => {
    Object.defineProperty(global.document, 'cookie', {
        writable: true,
        value: 'csrftoken=token'
    });

    const xhrMock = {
        open: jest.fn(),
        setRequestHeader: jest.fn(),
        send: jest.fn(),
        status: 200,
        responseText: JSON.stringify({'session': 'session'}),
    };
    global.XMLHttpRequest = jest.fn(() => xhrMock);
    global.sessionStorage = sessionStorageMock;
    global.getCookie = jest.fn(() => 'token');

    getSessions();

    expect(XMLHttpRequest).toHaveBeenCalled();
    expect(XMLHttpRequest().open).toHaveBeenCalledWith("GET", "
    expect(XMLHttpRequest().setRequestHeader).toHaveBeenCalledWi
    expect(XMLHttpRequest().send).toHaveBeenCalled();
});
```

Our Jest tests are run using the `npx jest` command.



```
PS C:\Users\02nke\dcu\yr4\4yp\2024-ca400-kellyn88-graya27\src\backend> npx jest
 PASS  tests/users.test.js
 PASS  tests/login.test.js
 PASS  tests/message.test.js
 PASS  tests/problem-solving.test.js
 PASS  tests/modal.test.js
 PASS  tests/problems.test.js
 PASS  tests/start-ide.test.js

Test Suites: 7 passed, 7 total
Tests:       27 passed, 27 total
Snapshots:   0 total
Time:        9.405 s
Ran all test suites.
PS C:\Users\02nke\dcu\yr4\4yp\2024-ca400-kellyn88-graya27\src\backend>
```

Backend Testing

Our backend tests are implemented using Django's built-in unit testing functionality. We can write functions within the `tests.py` file which can be used to test each of the functions in the backend `views.py` file.

Within the `tests.py` file there is a `setUp` function. This function is run before any other tests and creates any objects or variables that are required for the following tests. Our `setUp` function creates a `Client()` instance. This `Client()` object simulates a web browser and allows us to make requests to our Django views as if they were being accessed by a real client.

```
def setUp(self):
    self.client = Client()
```

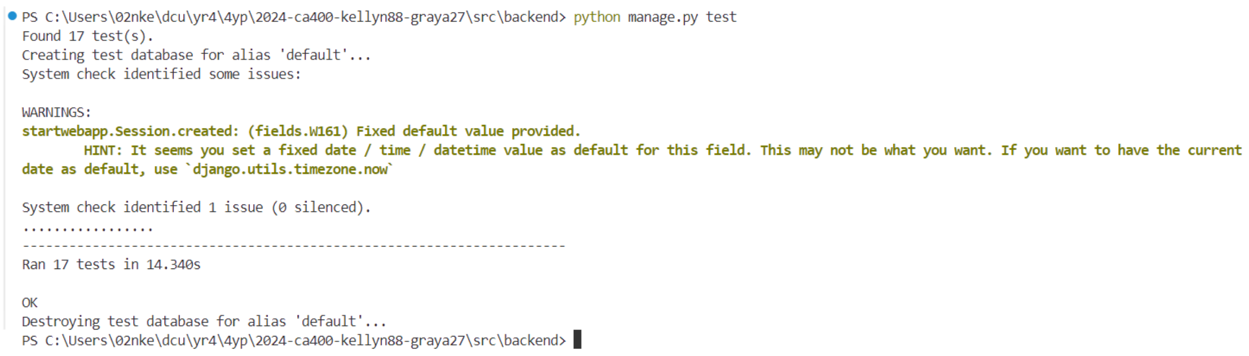
Each test is responsible for simulating a particular backend function. For example the below test tests the login function with valid credentials (there is also an identical test with invalid credentials). The test creates a user and commits it to a false database. It is important to mention that when Django runs these tests it creates a mock database which mimics the models of our actual database, however, when the tests finish this database is erased. When the user is created it uses the mock client to post a request to the 'login' url. When this completes, several assertions are ran to ensure the expected actions take place.

```
def test_login_user_valid_credentials(self):

  user = User.objects.create_user(username='testuser', password=

  data = {
      'username': 'testuser',
      'password': 'testpassword'
  }

  response = self.client.post(reverse('login'), data)
  user = authenticate(username='testuser', password='testpasswo

  self.assertEqual(response.status_code, 302)
  self.assertEqual(response.url, reverse('home'))
  self.assertIsNotNone(user)
```

More complex tests, such as our code cancellation function require mocks. For this we use the Python module unittest's built-in function 'patch'. Patch allows us to not run certain commands if they are happened across within the code being testing. For example, the cancel code feature attempts to kill a running jar process, however when these unit tests are running, no such jar execution is taking place. For this reason, we patch the `os.kill` function. This means when this function is found in the code being tested, its operations will be simulated, but will not actually happen. We can also assign these patches to mock objects, such as `psutil.Process` in the below example. By assigning the patch to a mock object we can then run assertions to ensure that the function is called in the code with the correct arguments.

```
@patch('os.kill')
def test_cancel_code_POST(self, mock_os_kill):
    data = {
        'process': '123',
        'uuid': '456'
    }

    # create files for user-files directory
    working_dir = os.getcwd()
    user_files_dir = os.path.join(working_dir, 'user-files')
    os.makedirs(user_files_dir, exist_ok=True)
    with open(os.path.join(user_files_dir, 'input456.txt'), 'w'
        f.write('some content')
    with open(os.path.join(user_files_dir, 'breakpoints456.txt'
        f.write('some content')
    with open(os.path.join(user_files_dir, 'memory456.csv'), 'w
        f.write('some content')
    with open(os.path.join(user_files_dir, 'instruct456.txt'),
        f.write('some content')

    with patch('psutil.pid_exists', return_value=True):
        with patch('psutil.Process') as mock_process:
            mock_process.return_value.is_running.return_value =
            mock_process.return_value.cmdline.return_value = ['
            response = self.client.post(reverse('cancel-code'),

            self.assertEqual(response.status_code, 200)
            self.assertJSONEqual(response.content, {'message':
            self.assertFalse(os.path.exists(os.path.join(user_f
            self.assertFalse(os.path.exists(os.path.join(user_f
            self.assertFalse(os.path.exists(os.path.join(user_f
            self.assertFalse(os.path.exists(os.path.join(user_f
            mock_os_kill.assert_called_once_with(123, 15)
```

We trigger the backend unit testing with `python manage.py test.`



```
PS C:\Users\02nke\dcu\yr4\4yp\2024-ca400-kellyn88-graya27\src\backend> python manage.py test
Found 17 test(s).
Creating test database for alias 'default'...
System check identified some issues:

WARNINGS:
startwebapp.Session.created: (fields.W161) Fixed default value provided.
        HINT: It seems you set a fixed date / time / datetime value as default for this field. This may not be what you want. If you want to have the current
date as default, use `django.utils.timezone.now`

System check identified 1 issue (0 silenced).
.................
----------------------------------------------------------------------
Ran 17 tests in 14.340s

OK
Destroying test database for alias 'default'...
PS C:\Users\02nke\dcu\yr4\4yp\2024-ca400-kellyn88-graya27\src\backend>
```

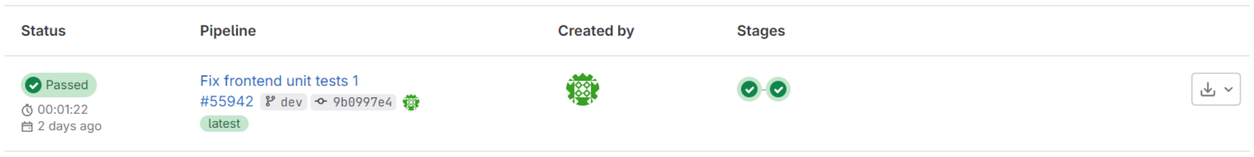
CI/CD

A major part of CI/CD we incorporated into this project was code reviews and merge requests. When we were attempting to merge any work work done on a local branch into either the dev or master branch, it was mandatory that we create a merge request. This allowed us to review our code and examine it for any underlying issues before merging our work. Only when we were satisfied with the work that had been done would we approve a merge into dev/master.

Another aspect of our CI/CD was our Gitlab pipeline. This was a series of tests that were ran when either dev or master branches were updated to ensure there integrity. These test commands were outlined in a yml file located in our repository, which was ran for every update by a Gitlab Runner located on one of our machines (see technical guide). Once we were satisfied that all backend and frontend code was operating as expected with passing unit tests, we added the unit tests to our Gitlab pipeline.

```
stages:
  - build
  - test

django-build:
  stage: build
  only:
  - master
  - dev
  image: python:3.11
  before_script:
    - pip install Django
    - pip install requests
    - cd src/backend
    - echo "Starting Django server..."
    - Start-Process python -ArgumentList "manage.py runserver 80
    - sleep 20
  script:
    - echo "Full server statup process succesful."
    - echo "Shutting down server."
    - $process = Get-Process -Id (Get-NetTCPConnection -LocalPor
    - Stop-Process -ID $process.Id -Force
    - echo "Server shut down."

django-test:
  stage: test
  only:
  - master
  - dev
  image: python:3.11
  script:
    - echo "Starting backend test for Django server..."
    - cd src/backend
    - python manage.py test
    - echo "Backend test finished."
    - echo "Starting frontend test for Django server..."
    - npm install
    - npx jest
    - echo "Frontend test finished."

start-complete-test:
  stage: test
  only:
  - master
  - dev
  image: java:18
  script:
    - echo "Starting test for standard jar..."
    - cd src/start/CI-CD-testing/standard
    - ./test.sh
    - echo "Standard test finished."

    - echo "Starting test for debug jar..."
    - cd ../debug
    - ./test.sh
    - echo "Debug test finished."
```

The yml file was broken into two stages: build and test. The build stage was responsible for booting up our server, ensuring that no errors were present during the start-up procedure. If and only if this stage completed successfully could the test stage begin. The test stage was broken into two stages. One stage was responsible for running our frontend and backend unit tests highlighted above. The other was responsible for testing our START jar file, to ensure that our language was still operating as expected. These two stages ran simultaneously during the test stage. Once the build and test stages had passed, the pipeline passes successfully. Here is an example of a pipeline run, with both stages passing.



Status	Pipeline	Created by	Stages
✓ Passed 00:01:22 2 days ago	Fix frontend unit tests 1 #55942 dev 9b0997e4 latest		✓ ✓