Docker Einstein

Technical Guide

Name: Ben Coleman

Student Number: 15487202

Supervisor: Stephen Blott

Abstract

The goal of this project was the rebuild the *Einstein* script marker used within DCU to remove any dependencies and allow the application to be used anywhere, such as in other universities. The approach taken to develop the original app did not have a desired end product in mind, and was instead constantly updated and expanded. The aim of this project is to make Einstein a containerised, end to end product where scripts can be marked rapidly and efficiently against test cases that an administrator may set. Lots of steps were taken to remove any DCU specific dependencies, and ensure that the completed system was an extensible and deployable application.

Motivation

Since I did not have any dominant Final Year Project ideas, I consulted some lecturer ideas. This particular idea caught my attention due to its usage of *Docker* technology. During my INTRA work placement, I was introduced to *Docker* and experienced working on a containerised application within a *Docker* environment. Since I knew what *Docker* was and how containers/container orchestration tools worked, I thought that pursuing an application building on top of that would be even more beneficial.

I was also motivated by the fact that it was *Einstein*. Similar to most students of this course, I was already familiar with the file marking utility. Knowing both its purpose and its functionality, I could focus on design choices and the Docker aspect of the project. It was also motivating to know that remaking *Einstein* would involve web application development, which I had prior knowledge in.

Research

From a research perspective, I had two routes that needed to be explored deeper: *Docker* and the current *Einstein*.

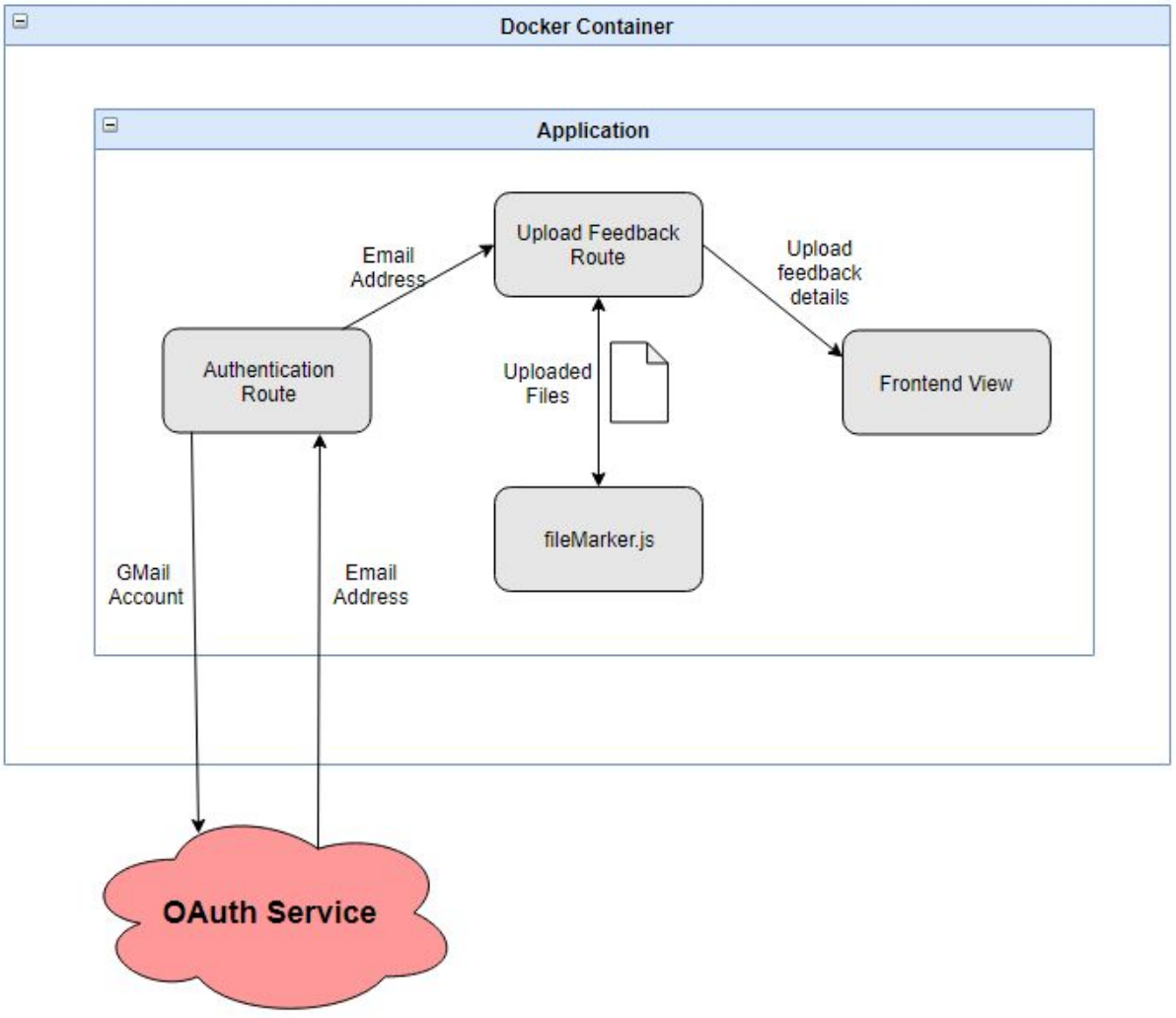
For *Docker*, I visited the company of my INTRA placement to seek any further guidance and tips on Docker development. Since I knew the project and direction by this point, I was able to ask questions regarding this specific development. These questions were aimed at covering the gaps I had in my knowledge. The biggest gap being that I had worked inside a Docker environment, whereas I was now looking to create one.

For *Einstein,* there was lots of consultation with Stephen Blott in the first few weeks making sure I fully understood the application. I was then able to pick out what parts were essential to my project without compromising the generic "*work anywhere*" approach of the application. There was then some research into the most fitting programming languages or frameworks I could use to implement this application. I looked into various web frameworks for various languages, such as *Spring* for *Java,* but ultimately settled on *Node.js* with *Express* and *JavaScript.*

System Design

The fundamental layout of the web application directory structure was dictated by the *Express* framework used with *Node.js.* There was also multiple design choices that had to be made throughout development of both frontend and backend to ensure the completed application would deliver on the promise of working anywhere outside of DCU and have no dependency like the current system.

Application Structure



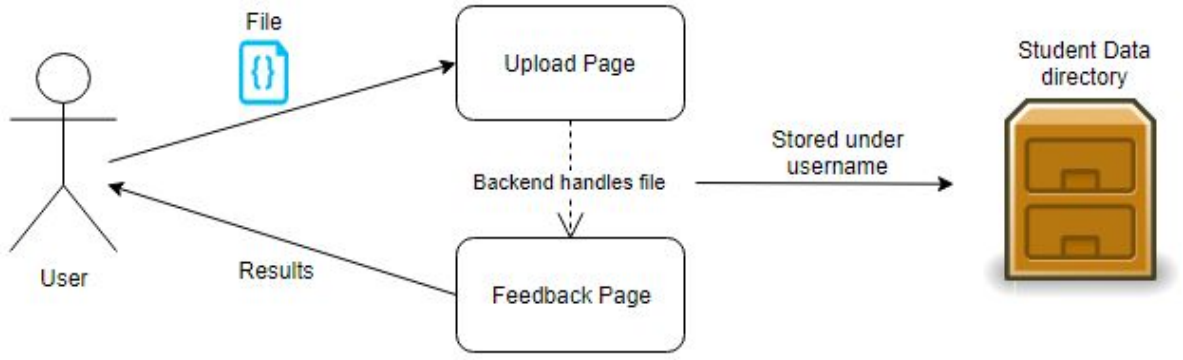
Backend

Handling Uploads

The application was designed to utilise a POST request for uploading a file to the server. The logic that handles the execution and marking of the file is done in another script ‘fileMarker.js’. This file still lives inside the application hierarchy and can be easily accessed from the route that handles the upload POST request. This design was chosen to separate file handling and execution code from file uploading and HTML response code.

Storing Uploads

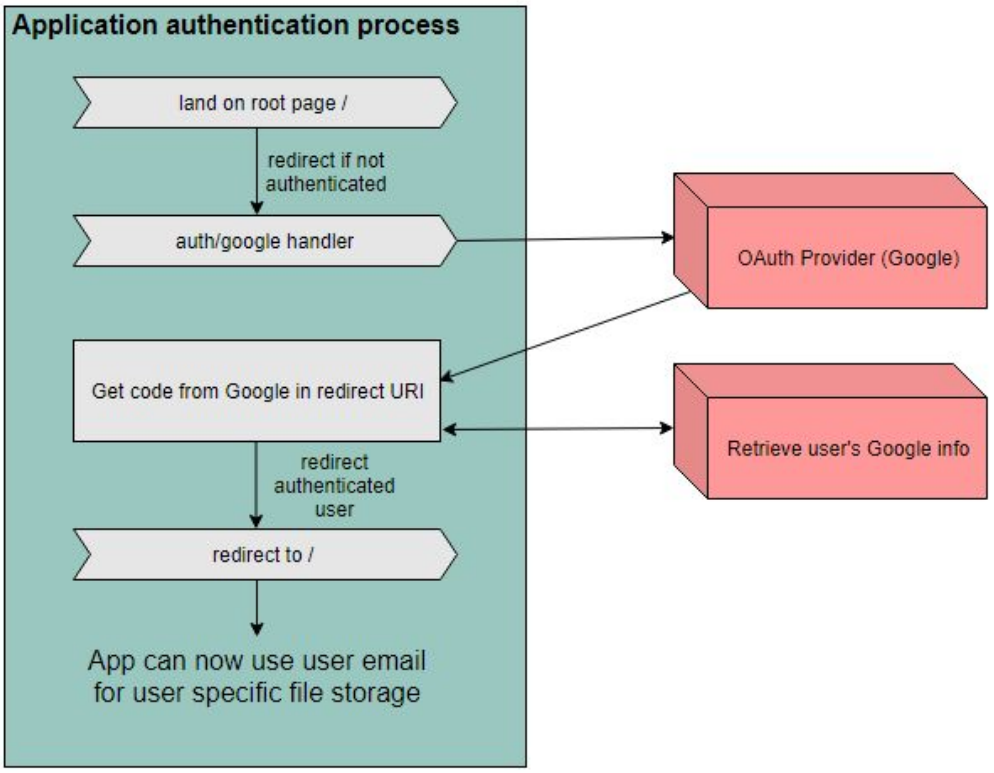
When thinking of how the storing uploads functionality could be designed, I knew a database was not suitable. Storing raw files is more appropriate within a file system, as is currently seen in Einstein. Therefore, I did not change this and opted for the same approach.



Authentication

The design process behind the authentication was simple. Allow OAuth to decide whether a user was valid and authentic or not. I did not need to store sensitive user information inside the application or hold onto a username longer than the session duration. This design was to ensure solid integrity and no security flaws within the application.

The following diagram shows the flow of control during authentication. This is the standard design of most systems using OAuth. If a user is not logged in they are redirected to a custom auth/google handler. A user is then redirected to Google’s sign in page to select a Google account. After selection, a code is returned in the redirect URI. This individual token is then used by my application to retrieve information from Google about the user, such as a profile or email. Once the user is logged in they can now access the root landing page.



Frontend

The front end was designed with extensibility in mind. The aim of this project was to make Einstein usable outside of DCU in any other university. This means that it should also be easy to use and accessible for any students outside of DCU. The front end is minimalistic to remove any unnecessary distractions. The application has a primary purpose of uploading files to be marked, so the simple design is intended to focus the user on that functionality. This simplistic approach can be seen within the coding/styling of the front end. There are no complicated HTML/CSS fragments that may cause cross browser compatibility issues, since the system may be used by any browser in other universities.

Accessibility

The colours of the front end are high contrast and easily viewable for any users with visual impairments. The entire application can also be interacted with entirely through the ‘return’ key. On the upload page, a user can press Enter to open the file explorer, press again to choose a file, and then press it while on the feedback page to return back to the file upload. The test case results are have alternating colours to easily.

Implementation

Throughout this section any sentence ending with [1] , [2] etc. has a code fragment in the ‘Sample Code’ section with the corresponding number.

Backend

File Uploads

An upload is made using a *DropZone* on the uploads page. This is the box at the centre of the screen that a user can drag and drop files onto, or click to browse. The POST method handler of the feedback route is what controls the file uploading. Saving the file to student data, and removing it from the temporary uploads directory is carried out inside this file. There is also timers in place to ensure a file is given enough time to be fully uploaded before any further logic happens, and to ensure an upload has been safely saved in the correct student directory before it is removed from the uploads folder. [1]

File Operations

The ‘fileMarker.js’ file was handles all operations to be carried out on an uploaded script. The POST request handler in the route file calls on the first *fileMarker* method, ‘checkForMarker’. This method does an initial check to see if the uploaded file name matches with a directory name in the marker directory. This means that files that do not need to be executed are not executed. If there is a marker directory i.e “*markers/hello-world.py*” for an upload “*hello-world.py*”, then the file is valid and can be handled further. [2]

There are a total of 3 functions within *‘fileMarker.js’* namely “*checkForMarker*”, “*executeFile*” and “*getResults*”. Between these 3, all operations on any uploads can be carried out. Once a file does not call back as invalid from *checkForMarker*, it is passed to *getResults* which uses a for loop to access the file’s corresponding marker directory, and execute the upload against the input/output inside these test cases. [3]

The *executeFile* method handles running the uploads and getting their output. It builds the path to where the upload lives and executes it using a *JavaScript* ‘child_process’. These executions are handled inside a promise so the result can be returned to *getResults* properly when it is called. [4] Currently, the system has Python and Shell script functionality. However, the nature in which these languages are implemented makes it very simple to add any further languages that are needed.

For each execution of a file against each test case belonging to the file’s marker, three values are added to a results array; ‘Correct/Incorrect’ depending on whether a single test case was correct or not, the actual output of the file for that test case, and the expected marker output for that test case. Therefore at the end of *getResults* when a file has been run against every test case, I have a results array filled with triplets of information about each test case that can be used by the frontend to display feedback. [3] Before this goes to the frontend, the flow jumps back to where it was initially in *checkForMarker* when *getResults* was first called. Now that I have returned with the full array of test case information, I can call

back to the route handler with the relevant information such as upload status, test case pass rate, and outputs. [5]

File Storage

User uploads are stored in a file system structure. A database was not suitable since it was raw files being saved, so I opted for a file system where I could specify the path to a storage location based on the user's login credentials. Any files that are not valid uploads are not executed and also not saved, meaning a user's student data directory will only be relevant uploads.

The process of user specific data storage without a database was achieved using the authentication system. Once a user has authenticated, or logged in, their email address is saved inside *Node.js'* inbuilt local variable holder *'app.locals*'. By doing this, I could then access their email address from inside the POST request handler when an upload was made, and use it to construct the path to their own specific directory inside the student data file system. This was further suitable because no sensitive information was being saved or exposed inside the application. The email address was all that was required to create a user specific directory. For example, my email address 'ben.coleman5@mail.dcu.ie' would be used to make a directory 'bencoleman5' which maintains student uniqueness but is also easily identifiable. [6]

Frontend

The front end was implemented through the *Node.js* views folder linked to each of the routes. 'Pug' was the template engine used to build dynamic HTML inside of each view. One layout template contained all stylesheets, scripts and the navbar. I could then inherit from this inside each individual template and just focus on purely content rather than linking files again. *Pug* allows you to have variables in the HTML that can be passed in when the template is being rendered back inside the route file. This was how I passed the feedback information from route handler to frontend.

Authentication

The authentication system was implemented using a *Node.js* module called '*Passport*'. This allows you to select a strategy, which in this case was 'OAuth 2.0'. The application is then set to use passport with the specified strategy which has options that can be set depending on your requirements. [7]

The unique Google client ID and client secret are tokens given on the Google developer console that allow the application to link to the OAuth service. These tokens are saved inside 'keys.js' which is not committed to GitLab. This means that from the code, an outsider cannot find the unique credentials to this specific OAuth service and cannot access the keys file as it is not public.

The 'auth.js' route is how the application on my end ties in with the above *Passport* and Google logic. The route file calls on Passport to initiate this authentication process, and can execute code once it has returned. In my case, I grab the user email from the Passport response, save it in a *Node.js* local variable as mentioned above, and then redirect back to the homepage which should now display the upload screen since the user is authenticated. [8]

Sample Code

[1] File upload and file copied timeouts

```
var uploadTimeout = setInterval(fileExists, 1000);

//Check if file has been uploaded
function fileExists() {

    var fileLocation = file.path;

    //Create directory in student data
    var studentDir = path.join(__dirname, '../studentdata/');
    var userSaveDir = studentDir + userName + '/';

    //If file is uploaded, render feedback page and clear timeout
    if (fs.existsSync(fileLocation)) {
```

```
var fileCopiedRetry = setInterval(fileCopied, 1000);

//Wait until file has been copied
function fileCopied(){

    //When file has been copied, delete from uploads
    if (fs.existsSync(userSavePath)){

        fs.unlinkSync(fileLocation);

        clearInterval(fileCopiedRetry);
        return res.status(200).end('File has been uploaded');
    };
}
```

[2] Validity check

```
//Check if marker exists before continuing
if(fs.existsSync(markerDir)){...
}

else {
    checkForMarkerCb('Invalid upload file name');
}
```

[3] GetResults for loop

```
for(var i=1; i<=fileLength; i++) {
    var sampleOut = markerDir + '/test' + i + '/stdout.txt';
    var sampleIn = markerDir + '/test' + i + '/stdin.txt';

    //Grab marker stdout.txt
    var markerOut = fs.readFileSync(sampleOut, 'utf-8', function(err){
        if (err){
            throw err;
        };
    });

    var stdin = "";
    //Check if test case has stdin
    if (fs.existsSync(sampleIn)) {
        stdin = sampleIn;
    }

    //Run the file and wait for the output
    var fileOut = await executeFile(filename,stdin,user);

    if (markerOut == fileOut){
        resultsArr.push(['Correct', fileOut, markerOut]);
    }

    else {
        resultsArr.push(['Incorrect', fileOut, markerOut]);
    }

}

if (resultsArr.length == fileLength) {
    return resultsArr;
}
```

[4] File execution - Python block

```
return new Promise(function(resolve,reject){

   //If Python file
   if (fileExt == 'py'){

       //Run test with input
       if (stdin != ""){
           const child = execFile('python', [fileUpload, stdin], (err,stdout,stderr) => {
               if (err) reject (err);

               else resolve(stdout);
           });
       }

       //Run test without input
       else{
           const child = execFile('python', [fileUpload], (err,stdout,stderr) => {
               if (err) reject (err);

               else resolve(stdout);
           });
       }

   }
```

[5] Results array operations before calling back to route file

```
var resultsArr = await getResults(file.name, files.length, markerDir, user);

if (resultsArr){

    var pass = 0;
    for(var i=0;i<resultsArr.length;i++) {

        //Check status
        if (resultsArr[i][0] == "Correct"){
            pass += 1;
        }
    }

    var passRate = pass + '/' + files.length;

    if (pass == resultsArr.length){
        checkForMarkerCb('Correct', passRate, resultsArr);
    }

    else {
        checkForMarkerCb('Incorrect', passRate, resultsArr);
    }

}
```

[6] User specific directories

```
//Get authenticated users name
var email = req.app.locals.emailAddr
var user = email.substring(0, email.lastIndexOf('@'));
var names = user.split('.');
var userName = names[0] + names[1];
```

```
tmpDir = path.join(__dirname, '../uploads/' + userName + '/');
if(!fs.existsSync(tmpDir)){
    fs.mkdirSync(tmpDir);
}

//Set path of file to ../uploads/user/filename
file.path = tmpDir + file.name;
```

[7] Passport options

```
passport.use(
    new GoogleStrategy({
        //Options for the strategy
        callbackURL:'/auth/google/redirect',
        clientID:keys.google.clientID,
        clientSecret:keys.google.clientSecret
    }, (accessToken, refreshToken, email, done) => {
        //Passport callback function
        done(null, email)
    })
)
```

[8] Auth.js route

```
//Authentication with Google
router.get('/google', passport.authenticate('google', {
    scope: ['email'],
    accessType: 'offline',
    approvalPrompt: 'force'
}));

//Callback route for Google redirect
router.get('/google/redirect', passport.authenticate('google'), (req,res) => {
    //Save user email in req.app.locals to use in feedback route
    req.app.locals.emailAddr = req.user.emails[0].value;
    res.redirect('/')
});
```

Problems Solved

Throughout the development lifecycle several problems were encountered. These were a mixture of design choices, technology conflicts or bugs. Solving these issues required different approaches depending on the type of problem. Some required a new approach, alternative design or extra code, while others were solved by letting the problem incubate for a few days to figure out an appropriate solution.

Bugs

In the very early stages of development, I found that files being uploaded to the server were not actually being uploaded, even though the response from the server indicated they had been. After some research I found that the encoding type was not set correctly to allow the server to receive the files, which wasn't enough of an issue to throw any errors.

Technology Stack

The application is built using *Node.js* alongside *ExpressJS*, inside *Docker*. All files were written in *JavaScript*, with some uploads requiring a child process of *Python* or *Shell* to be

executed. The authentication was implemented through Google OAuth and the frontend through *'Pug'* templating engine rather than plain HTML. Because of this range of different technologies interacting together in a narrow, closely linked scope, some problems were introduced.

One problem was that there is not a lot of documentation or resources on projects that use a similar range of technology. For example, I can get help on a particular *Node.js* issue, or a suspected *Docker* error with ease. However, I cannot always find resources on a *Docker Node.js* combined error that may be because of the two technologies clashing. I solved this by trying to make sure to implement code in isolation so that I could minimise the chance of bugs being technology related.

Asynchronous Execution

Node.js is "an asynchronous event driven JavaScript runtime". This can cause some issues within a project where asynchronous execution is not suitable. While there are times the code can continue to run while previous operations happen in the background, there are other times when the flow must wait for one block to finish execution.

An example of this is checking file output against a particular test case. The test case subdirectories can be accessed and read at any time, but once a file has been called to execute, the script must wait for it to finish before checking its output against the test case, or else the output simply won't exist. To solve this I utilised the asynchronous execution as much as possible where I could, but used the *'await'* keyword in special cases where it was required.

Results

The result of this project is an operational, extensibile and deployable remake of the Einstein file correction utility. The system is portable and completely containerised without any external requirements or files that must first be installed on a host system. To achieve this, there had to be slightly less customizability on the backend. However the flow and interaction to a user on the frontend is largely unchanged and still simple, operational and fast.

The whole system was developed with native *JavaScript* as a *Node.js* project. The code is thoroughly commented/documented throughout, meaning that while the application itself is already extensible from a deployment standpoint, it is also extensible by the fact that it can be further developed, customised or adjusted to suit a host customer's needs.

As a software developer, the result was extensive experience designing, planning and implementing an end to end project alone. Aside from supervisor consultation, all decisions were made alone with only myself to praise or blame for the respective success or failure. This experience is really significant starting on a career path of software engineering/development, as it helped me to make advised and well thought decisions and stand by them. This project also helped me to act on those decisions and solve all the many problems that came with delving into the unknown and figuring out how everything was going to come together.

There was also the resulting knowledge gain in web application development, *Docker* technology, and *JavaScript* coding. With web applications already becoming a worthy

alternative to stand alone OS specific apps, it is highly beneficial to get further experience developing projects like this. The same can be said about *Dockers*, and their growing popularity among even large software firms. The information I have learnt through the research will without question be useful to me wherever I end up as an engineer. Lastly the *JavaScript* coding was worthwhile, due to it not being taught on the course. The experience I have in JavaScript is now close to the level achieved in *Python* and *Java* due to them being huge factors in the course.

Future Work

Regarding future work, it may be interesting to do some research into the differences between this containerised version of Einstein and the currently implemented system in DCU. The two systems could be compared under criteria such as server performance under load, efficiency of file uploading and executing, or usability/functionality. This wasn't a possible step within the scope of this project since I do not have access to the current Einstein system backend. However, if this project was being considered for implementation into a real user environment it may be worth considering these factors.

Building on from this, it would be worthwhile to do some alternative server testing. The approach to this project was to prove that it works anywhere by showing that there is no reason it should not. I avoided any dependencies or relation to DCU specifically and ensured that every design step was implemented in a manner that did not impact extensibility. Even with this approach, it would still be interesting to test some real server usage of the application either on DCU's server or another university.

Future work may also include further functionality or adding some customisation to the application relating to where it is being hosted. The system as it stands is a blank canvas application that has all functionality required to work, but does not go overboard to a point where extensibility is lost. The option is there for code to be added to give the base Docker Einstein created in this project a particular twist for smoothing merging with how some university operates. Maybe adding module code recognition, or integrating the file upload system with some internal plagiarism checker that may be used in other places of a university's infrastructure.

Guidelines on Testing

Testing Type

The scope of testing required for full coverage of this system is quite narrow. The primary functionality & user interface can be tested fully without the need for long test suites. The web application itself uses end-to-end *Nightwatch.js* browser testing to ensure proper functionality and interactions that a user may have. The unit tests are designed to ensure the web application is operational and responding properly to any requests made to it.

Ad Hoc Testing

Throughout the development of this system there was frequent ad hoc testing undertaken. Whenever a new feature was being implemented I would restart the system and ensure each step taken was progressing the system in the right direction, without any obvious conflicts. The reason this was ad hoc testing is because it was not documented, nor was it compliant with any rigid test case structure, but was more from a developer point of view to check the system service continuity as changes were being made.

While building the system, I maintained a ‘test data’ directory that contained files that I could upload to the server expecting various responses. There was a file in *Python* and a matching file in *Shell* to test the multi language functionality and ensure this was never compromised at any stage. There was a file to test a completely incorrect script, and another to test a partially incorrect script. These were to show the response on the feedback page and if the system could still run test cases and display results appropriately. There was also a file ‘sampletext.txt’ that had a random name and file extension that the system did not yet have a marker for. This file allowed me to test if files that were not supposed to be executed were continually not being executed. I could also use this to test the file storage facilities and ensure invalid files were not being saved to student directories.

System testing documentation

Unit Tests

The unit tests for this application are written with *Mocha* test framework alongside *Chai* assertion library. They are used to simply ensure correct operation and response from the application server.

```
var expect  = require('chai').expect;
var request = require('request');


describe('Status and Content', function() {
    describe('Upload page', function() {
        it('checks status', function(done){
            request('http://localhost:3000' , function(err, res, body) {
                expect(res.statusCode).to.equal(200);
                done();
            });
        });

        it('checks content', function(done) {
            request('http://localhost:3000' , function(error, res, body) {
                expect(body).to.contain("Upload Page");
                done();
            });
        });

    });

    describe('Feedback page', function() {
        it('checks status', function(done){
            request('http://localhost:3000/feedback', function(err, res ,body){
                expect(res.statusCode).to.equal(200);
                done();
            });
        });

        it('checks redirect to /', function(done){
            request('http://localhost:3000/feedback', function(err, res ,body){
                expect(body).to.contain('Upload Page');
                done();
            });
        });
    });
});
```

This is an example of such unit test. This checks the status and content of both pages of the application once a user has authenticated. The feedback page testing ensures that the system redirects to the upload page if a user tries to make a GET request to /feedback without uploading a file.

```
var expect  = require('chai').expect;
var request = require('request');


describe('Upload Page', function() {
    describe('Navbar', function() {
        it('checks navbar content', function(done){
            request('http://localhost:3000' , function(err, res, body) {
                expect(body).to.contain('Docker Einstein');
                done();
            });
        });
    });

    describe('Body', function() {
        it('checks body content', function(done){
            request('http://localhost:3000/feedback', function(err, res ,body){
                expect(body).to.contain('Upload Page');
                done();
            });
        });

        it('checks DropZone', function(done){
            request('http://localhost:3000/feedback', function(err, res ,body){
                expect(body).to.contain('dropzone');
                done();
            });
        });
    });
});
```

This unit test is similar to the above but is used to check the rendering of the upload page and ensure all primary elements are displayed properly. Critical sections of the page such as the navbar and upload dropzone must be visible on page load to allow user interaction with the system.

The result of these unit tests can be seen below:

```
benjyc@BenThinkPad:~/Programming/4YP/2019-ca400-colemab5/src/app$ npm test

> app@1.0.0 test /home/benjyc/Programming/4YP/2019-ca400-colemab5/src/app
> find app/test/unit -name '*.js' | xargs mocha -R spec



  Upload Page
    Navbar
      ✓ checks navbar content (53ms)
    Body
      ✓ checks body content
      ✓ checks DropZone

  Status and Content
    Upload page
      ✓ checks status
      ✓ checks content
    Feedback page
      ✓ checks status
      ✓ checks redirect to /


  7 passing (227ms)
```

Integration Tests

The integration testing was carried out using *Nightwatch.js* end-to-end testing framework. This works alongside *Mocha* to carry out *Selenium* based browser testing. Again, the scope of this testing was quite limited based on the low complexity of the frontend. However, some test cases were written to handle unseen and potentially bug inducing events.

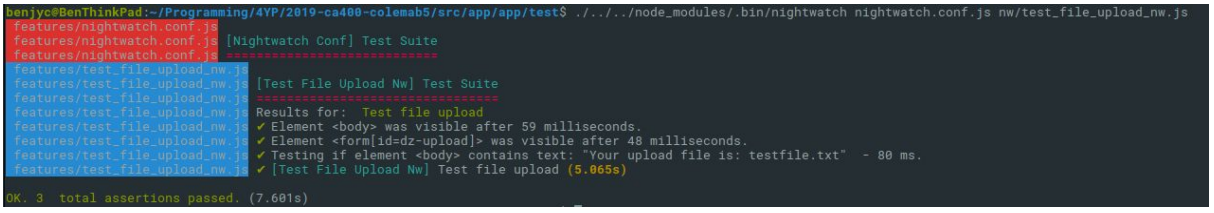
Such test cases are coded in the style seen below. These use web drivers to open the browser and carry out various operations. In this case, the test case uploads a dummy file and ensures the resulting feedback page is seen and the file uploaded was recognised.

```
const assert = require('assert');
const path = require('path');

module.exports = {

  'Test file upload' : function(browser) {
      browser
        .url("http://localhost:3000")
        .waitForElementVisible('body', 1000)
        .waitForElementVisible('form[id=dz-upload]', 1000)
        .pause(1000)
        .execute("document.querySelectorAll('input[type=file]')[0].style.display = 'block';")
        .setValue('input[type=file]', path.resolve(__dirname + '/testfile.txt'))
        .pause(3000)
        .assert.containsText('body', 'Your upload file is: testfile.txt')
        .end();
  }
};
```

The results on the backend of running this test can be seen below:



There were some integration tests at the earlier stages of the development lifecycle that were later removed due to refactoring or design changes that made them redundant. An example of this is a test to catch no file being uploaded to the system. This was a test case seen when the system used a ‘choose file’ and ‘submit’ button and was to ensure continuous service if a user clicked submit without a file. The implementation of the DropZone area seen on the upload page has removed the need for this test. There is no longer a submit button, and the uploading process can only be activated when a file is chosen from the user’s file explorer.

User Testing

Recruiting Users

For user testing of this system, I allowed a group size of 5 to interact with the application. This was undertaken at a single point in the development process when the first thin system had been fully implemented. These 5 users had different levels of computing knowledge and background, and contained 2 members from the course and 3 from outside of the course.

Facilitating Testing

For the purposes of this testing, I did not use authentication and supplied users with the sample test data directory that contained my own sample scripts. This was to ensure anonymity and allow users to judge the design and accessibility of the application without ethical responsibility. For the duration of any user interaction with the system I remained separate and did not lead their perception of the system.

- To facilitate proper user testing of the application as a file correction utility, I gave the users **preliminary information on what Docker Einstein is**. The decision to release this information was that any users who are to interact with this system wherever it is hosted will already know its purpose. The aim of the system is not to appeal to the general public, but to serve as a tool for people who know when and why it is required.

User	Design (1-5) (higher is better)	Complexity (1-5) (lower is better)	Ease of use (1-5) (higher is better)	Comments
1 (Computing)	2	1	4	System too basic but easy to interact with.
2 (Computing)	3	1	5	System is simple, does what it's supposed to.
3 (Non-computing)	2	2	3	Self-explanatory, figured out relatively fast.
4 (Non-computing)	3	1	4	Likes the minimalistic design, not very fance but easy to follow.
5 (Non-computing)	4	2	2	Confused on purpose due to lack of computing knowledge but keen on design.

Objectives

This short, concise table shows the summarised data taken from the group of users. I asked them to fill in a sheet at the end which contained the 3 criteria seen below (Design, Complexity, Ease of use) and then made my own notes on their comments throughout. The comments above are a summarised version of the comments made by users.

Analyzing Results

The general consensus of this simple round of user testing was that the system was simple and usable as intended, but was slightly too basic by design. This is why it was beneficial to carry out this testing after the first initial system was implemented and not later in the development lifecycle. I was able to build on these comments and add some further styling to the frontend that would make the design more appealing to users who want more than a blank page with a file upload field.

Accessibility Testing

Accessibility testing was significant for this system due to its “deployable and usable” anywhere design ideology. While the system is designed to have a backend that will not conflict with any servers or host systems outside of DCU, it is also designed to have a front end that can facilitate any students that may be interacting with it.

The following checklist was consulted throughout development after the initial thin system was built and was maintained until project completion.

1. Are there keyboard equivalents for all mouse operations? Yes.
2. Are instructions provided as part of a user manual or documentation? Is the system easy to understand/operate using the documentation? Yes.
3. Whether all labels are written correctly in the application? Yes.
4. Whether images or icons are used appropriately for users to easily understand? No images used, and icons are attached to wording to emphasise meaning i.e file icon beside the “Upload Page” heading.
5. Whether user can adjust or disable flashing/rotating/moving displays? Display is static with no unnecessary animations.
6. Check to ensure colour-coding is never used as the **only** means of conveying information? The colour-coding of the file status on the feedback page is superficial and does not hide the information itself (correct/incorrect).
7. Whether colour of application is flexible for all users? High contrast colour scheme for readability.

Vision Impairment

Users suffering from visual impairment such as poor vision can still interact with the application due to its simple design. The upload page can be used just with the ‘Enter’ key on the keyboard, if preferred. The feedback page is a static text representation of how the uploaded file performed against test cases. The colour scheme is high contrast to increase readability and could be read with a screen reader if needed.

Colour blindness is also catered for due to no information relying solely on colour. The colouring of the individual feedback data does not stop a colour blind user from still seeing the content of this data.

A user with severe to full vision disability may require a screen reader when using web services. This application is primarily text based with an upload function that can be used with the keyboard and does not require specific element locating and clicking. This means that the feedback page can be read by a screen reader and conveyed to the user is a means suited to them.