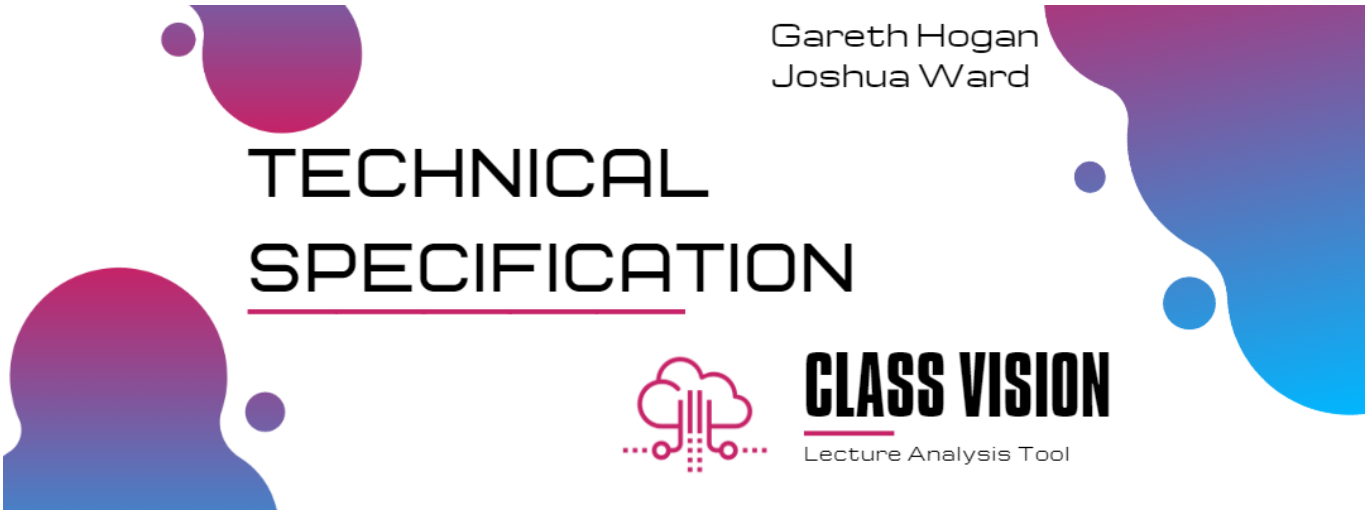
DCU CA326 - Third Year Project



Gareth Hogan 20379616 & Joshua Ward 20460854

Table of contents

Introduction

Overview

The system we have developed is called Class Vision. The purpose of this system is to use computer vision technology to analyze and gather data during lectures and present it back to users to help them improve their lectures, the system focuses on data points such as attention, attendance and emotion. The system can be used via a website frontend which allows users to register/login and manage their scheduled lectures and what modules they teach. Users can activate the analysis tool via the website, while the analysis tool is active it captures images from the users' camera, these images are analyzed using computer vision tools. The system detects faces, profiles (side-faces) and eyes, it also does further emotion analysis on the detected faces. Once the user ends the analysis the collected data is stored and a new report is generated on the website, users can view statistics and graphs in the report, and an aggregated report which combines all the individual lecture reports is also made for each module a user has created.

Glossary

Term	**Description**
Computer Vision	Computer Vision is a field of AI. It enables computers and systems to be able to see and observe, it allows them to derive information and data from visual inputs such as images and videos.
OpenCV	(Open Source Computer Vision Library) OpenCV is an open-source library full of functions mainly aimed at real-time computer vision and machine learning
Haar Cascade	A facial detection algorithm that uses edge detection features to identify faces or objects in an image or video. It is one of the oldest yet most powerful face-detection algorithms
Classifier/Model	These are large files of data points and weights, these are used alongside an object detection algorithm to define what it is the algorithm is looking for.

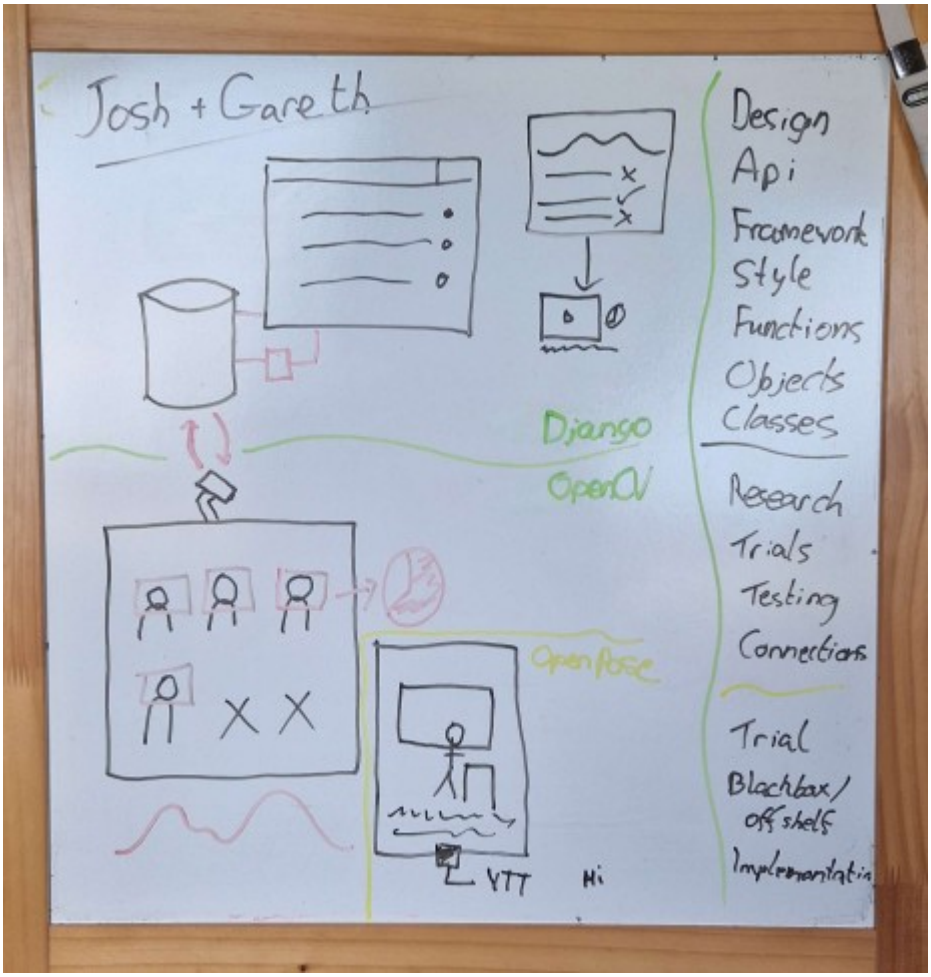
Motivation

Why did we choose this project? The initial plan for this project started with an idea on our supervisor's website, the idea was to develop some sort of classroom attendance tracker using a camera and rudimentary facial recognition. Below you can see the initial idea we read from.

CLASSROOM ATTENDANCE

The goal of this project is to create an app which shows an educator which students in their class are paying attention. The project will utilise a camera and a rudimentary facial recognition algorithm to convey to the educator which students are paying attention to the class. The results will be presented in a virtual classroom type manner.

We were initially quite interested in the facial recognition and camera aspect of the idea, it was something we had never done before, and we thought it would be very hard but upon a bit of research, we found that rudimentary facial recognition scripts weren't too difficult to make in Python. So we met with Michael and discussed the idea, over the course of two meetings we refined and expanded upon the idea and described a high-level overview of the system we wanted to make, we diagrammed the system on a whiteboard and discussed the various technologies and functions that would be involved in making it. Below you can see a refined version of our initial sketches on the whiteboard.



Our idea evolved from Michael's initial proposal, we moved away from rendering the results in a virtual environment, and we added a more traditional webpage and more data tracking, so rather than in-the-moment tracking of students' attention we provided post-lecture data and reporting/feedback.

Research

In this section we will describe some of the research we did for this project, providing some example links to our sources. A full list of links to sources can be found in /res/sources.txt

Microservice

The bulk of the research we did for this project was for the microservice which was responsible for the computer vision part of the project, analysing the camera input and collecting data, it was the experimental part of the project that neither of us had done before, and it required a lot of research and experimentation before we started developing the final implementation.

For the frontend and backend we did not do as much research, we had already decided on using a Django-Node base for our system and had done this before in CA298, the backend and frontend just needed a small bit of research while we were developing whenever we needed to do something we hadn't done before.

Using the camera and facial detection posed the biggest hurdle for us in our development. The research into openCV started early in the project with this tutorial. This tutorial was the first time we read about doing facial recognition, and this tutorial showed us that it was achievable (albeit in more than 25 lines of code).

OpenCV was the obvious choice of library to use since we read lots of tutorials online that used it, however, we did also consider another which was YOLO. YOLO was initially suggested by Michael and we did some prototyping with it (see */res/prototyping/yolo*) however it was suited more to object detection, so we continued with OpenCV, below are some more links to articles and posts we read on OpenCV:

- Guide to OpenCV - this gave us an overview of OpenCV and its basic functions, and how to install it.
- Webcam Face Detection - this was the sequel to the first tutorial which used a webcam instead of still images.
- OpenCV Github - the OpenCV GitHub was an important source specifically for the haar cascade models they provide we used in our detection algorithms.
- OpenCV Camera Use - this documentation was all about getting a video stream from our camera using OpenCV
- Javascript getUserMedia - this was from when I started using javascript to get the camera and images instead of the openCV functions

The final implementation of our system uses flask alone, but for a long time during research and prototyping we planned to also dockerize the flask app, here is some of the research we did for that plan:

- Github Flask-OpenCV-Docker Repo - This github was a good example we thought of what we aimed for, the combination of three technologies OpenCV, Flask and Docker
- Flask OpenCV tutorial - This was a very short but great example of using flask and OpenCV together, displaying the video stream captured using OpenCV on the html page rendered by flask

- Dockerizing a webcam Flask App - This gave us some ideas on how to build our flask app, specifically around the capture of images using a button on the flask page

After initial development and upon suggestions from Michael we also started to look into emotion detection, here are the links for research we did for that function:

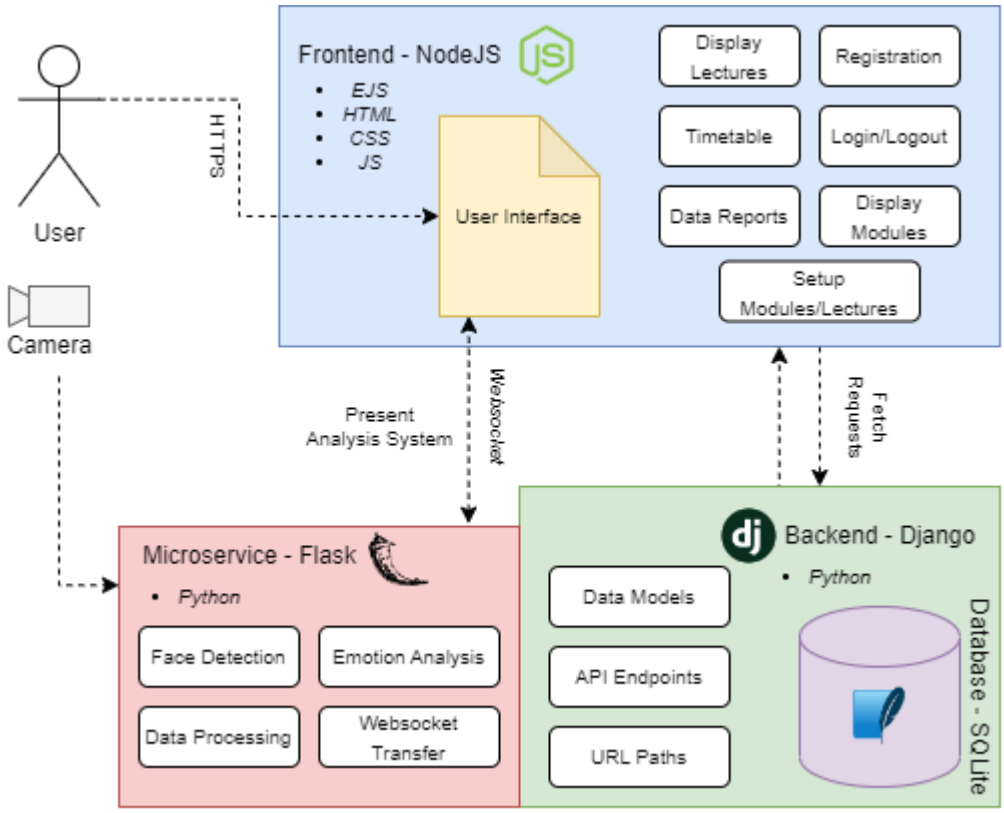
- Emotion Detection in Python - This was the first guide we found on the subject, the method displayed inside it didn't work well for us and we eventually started over
- How to detect emotions in Python - This is the second guide we followed, this introduces us to deepface, the library used in the final system for emotion detection

These are just a few links to show off some of the more important sources of our research early on for openCV, flask and the Microservice system as a whole, the more we used it and got used to the tech the easier development became. We wouldn't be able to link them all here but we also have to thank a lot of StackOverflow forums, no matter what strange error OpenCV gave us, someone had normally already experienced it and posted a solution.

Design

What are we building?
This section will go over the high-level design of our system, how we planned it out into different components and what each is responsible for. We will walk you through some diagrams we have for the system design.

System Architecture



Above you can see our **System Architecture Diagram.** This shows you a high-level overview of what the system design is, how we separated the functionalities into different components

exactly what each component is going to contain. It also contains some small implementation details such as links between components and what technologies we were going to use, we will talk more about the implementation in the next section. In this section, we are going to give an overview of how we designed our system and the individual components. You can see our system is separated into three main components;

- Frontend
- Backend
- Microservice

The role of the **frontend** component is to interface with the user, render the webpage and accept inputs from the user. The frontend presents the visual interface that the end user will see and interact with and is the only access point for users. Through the frontend the user can interact with the system processes and data contained in the backend, the frontend will take input from the user and send it back into the backend to be processed and it will also query the backend for data when the user requests it or it is needed to load a new page.

The **backend** is where most of the systems core processes and logic are located, it has no visual interface and the user interacts with it through the options presented on the frontend, its main responsibilities are to process and store the data in our database, this will include things like user account data, lecture and module models and then data reports which are gathered from the microservice analysis.

The **microservice** is contained within the backend, it is a small self-contained subsystem. Unlike the frontend and backend, it will not be running at all times the system in online, the microservice will be activated only when the user activates the analysis system via the frontend, the microservice will then present the user with a new UI which will capture images from the user's camera, the microservice will then analyze and record data from those images using computer vision models when the user ends analysis the microservice will bundle up the results and send them into the database through the backend.

Below we will go into more detail about each component, what their purpose is, what processes/functions each are responsible for, how they work and some diagrams representing each part.

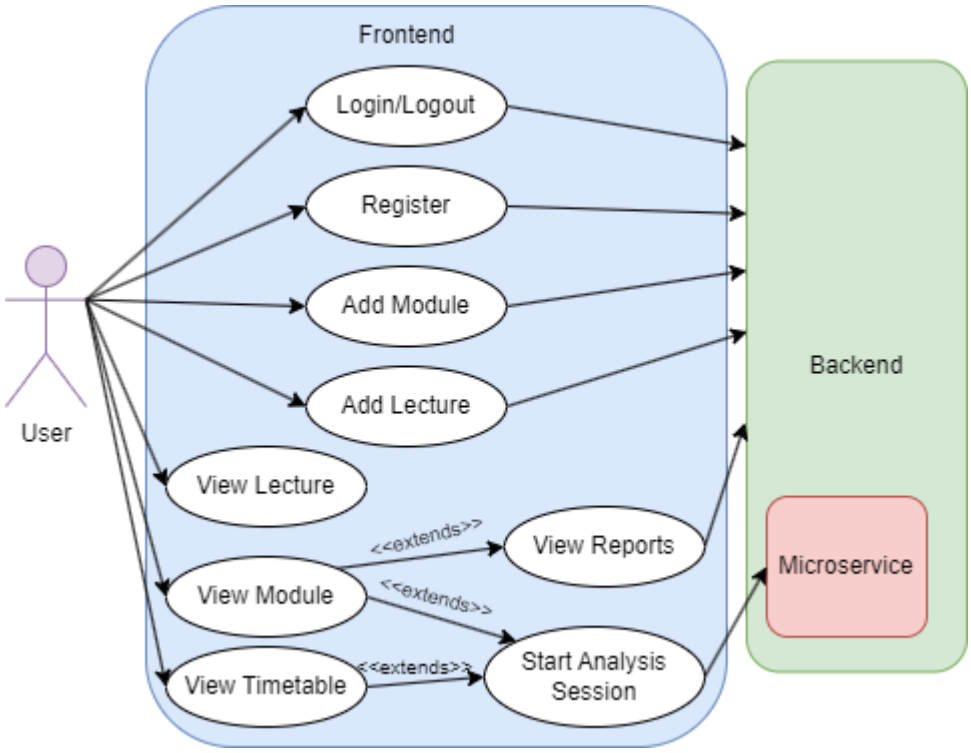
Frontend Architecture

The frontend is responsible for almost everything the user sees and interacts with, it will be generating and styling the webpage in part statically and in part dynamically using data retrieved from the backend. The main responsibilities of the frontend are to be the user interface, to display the system to the user and allow them to interact with it via links, and buttons and be a facade for the user to interact with the backend.
The core responsibilities of the frontend are to:

- Present the webpage to the user on a variety of pages
- Allow the user to access backend functions such as login
- Collect data from users who want to create new modules or lectures
- Send data into the backend for storage

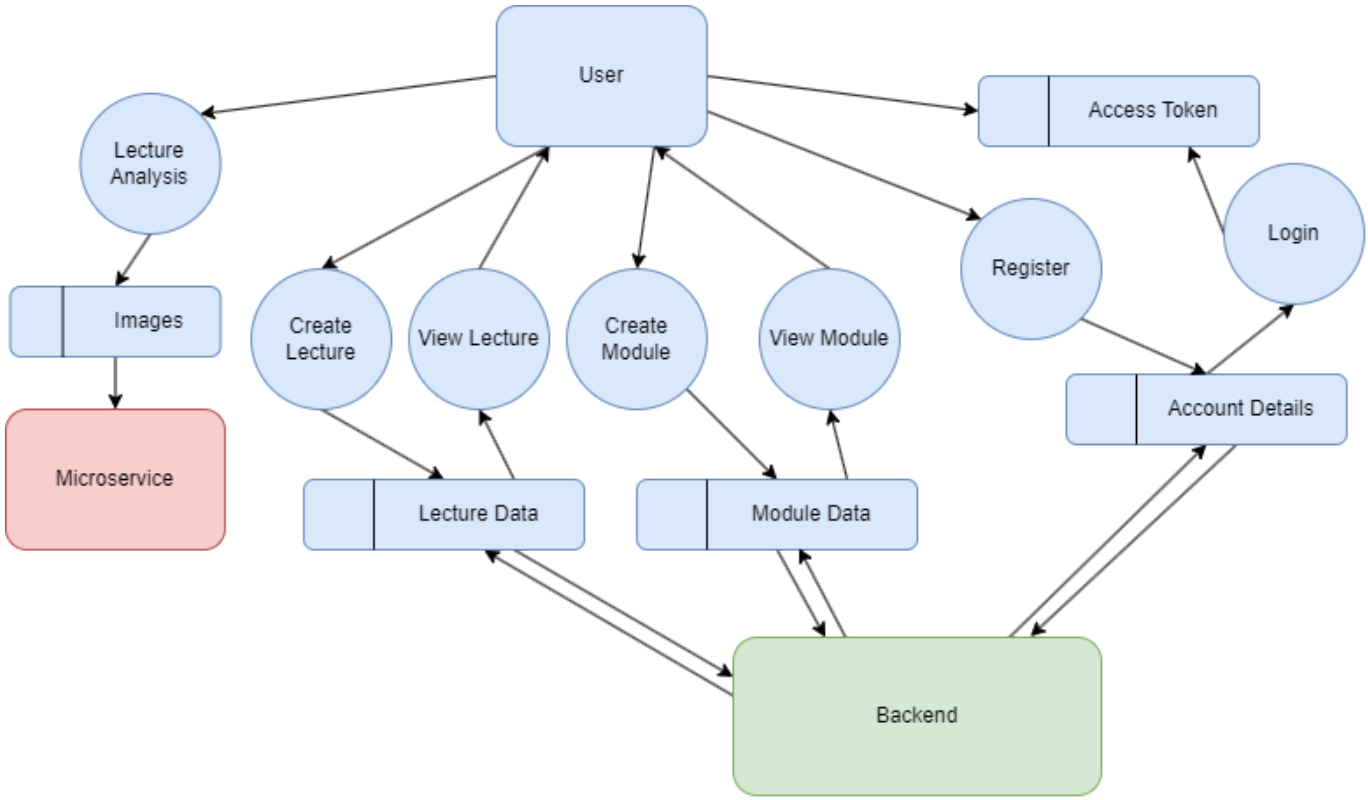
- Dynamically render data from the backend on the page in the form of modules, lectures or data reports to allow users to view them

Below you can view a use case model we made for the frontend, which shows you what functions the user can access.



You can see in the above diagram the multitude of things a user can access via the frontend, some of the functions are basic admin functions such as login and register, and then there are custom creational functions for setting up new modules and lectures. All of these functions interface with the backend for the user, sending data to where it needs to go without the user having access to the raw data and processes in the backend.
There are then viewing functions to display the lecture, module, timetable or report elements where the frontend will generate the elements using data fetched from the backend database. One of the most important functions is to start an analysis session, this function allows the user to access our microservice, and the function itself activates an endpoint in our backend to start up and display the microservice UI.

Next, you can see a data flow diagram for the frontend, this diagram helps to visualise the kinds of data going through the frontend and where it will end up and what functions use it.



At the top of the diagram, you can see the user. And near the bottom, you can see the other two components of the system the backend and microservice.
The circles represent processes and you may notice that they are all shared with the use case diagram from before, these are the things that the user will be accessing to do *something*, to achieve a goal.
Then we can see the data going around between the user and the other two components of the system;

- Images are sent from the user into the microservice during lecture analysis
- Lecture/Module data is either sent into the backend (creating) or retrieved from the backend (viewing)
- Account details are sent into the backend when a new user registers an account, this includes their username and email
- Account details are fetched when a user logs in, and an access token is given to the user during authentication

This is quite a short simple diagram which helps show what types of data is being moved around in the frontend, a lot more data is being processed and stored in the backend but what we can see on the diagram is important because it is what is presented to the user on the webpage.
Also here is a breakdown of what each type of data will typically contain;

Data	**What is in it**
Images	An image captured from the users' camera
Lecture Data	Lecture ID, Lecture Timeslot, Lecture Module, Number of Students registered

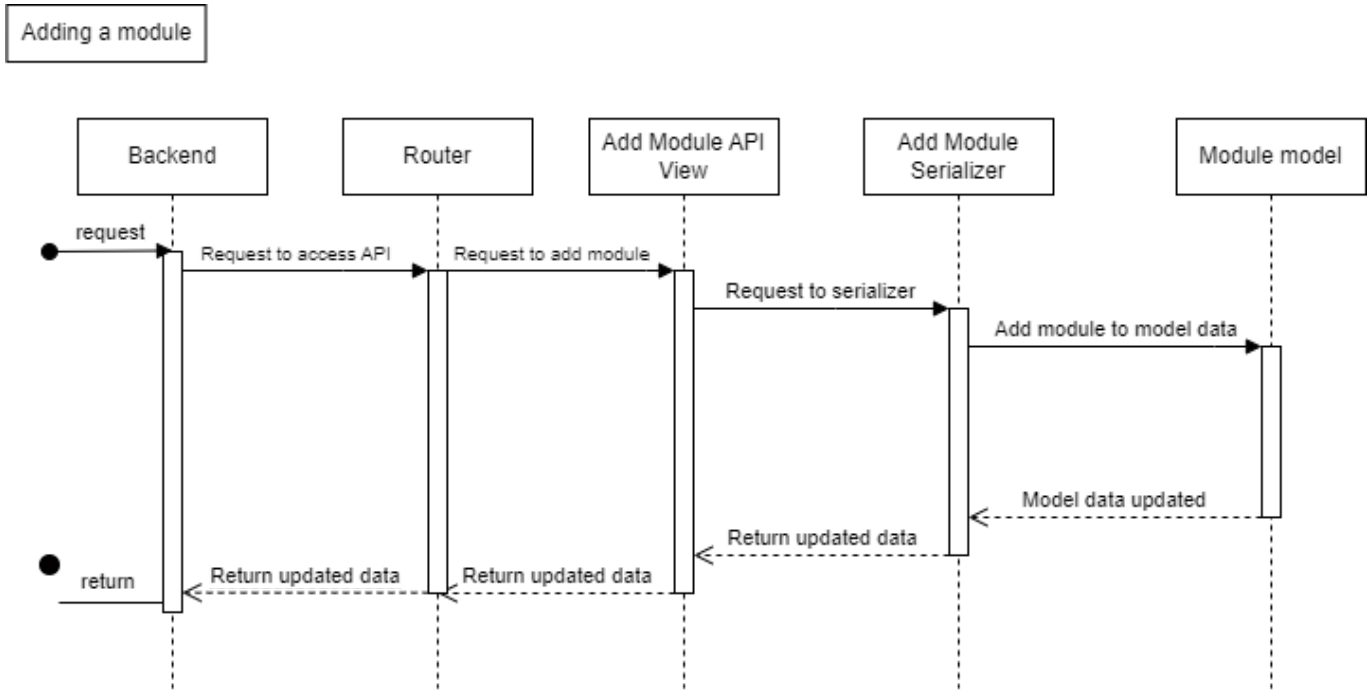
Data	What is in it
Module Data	Module Code, Module Name, Module Description, Number of Students registered
Account Details	Username, Email, Account ID

Backend Architecture

The backend is responsible for handling all the data that the user will be interacting with and changing, while also containing the microservice. It takes in requests from the user accessing the frontend and makes relevant changes to the data stored with regards to that, while also sending along any data requested by the backend. The core responsibilities of the backend are to:

- Store and update upon request the data for users, lectures, modules, and lecture analysis records
- Manage the login, logout, and registration functions
- Add new lectures or modules
- Remove lectures or modules
- Support access to the microservice

Below you can look through a sequence diagram showcasing the backend.



The sequence diagram showcases the order of options when a user requests to add a module to the system's database. This process is triggered upon a user clicking on the 'Add Module' button on the frontend, before filling in the module details and submitting them.

Description of the flow:

- Module details enter the backend through fetch request to the API

- Module details are passed to the API view function, which checks permissions before passing the details to the serializer
- The serializer takes in the details and validates the data it has been sent, before creating a new module object
- The module object is created and added to the model's data, before returning the updated data
- This updated data is passed back until it is sent back to the frontend.

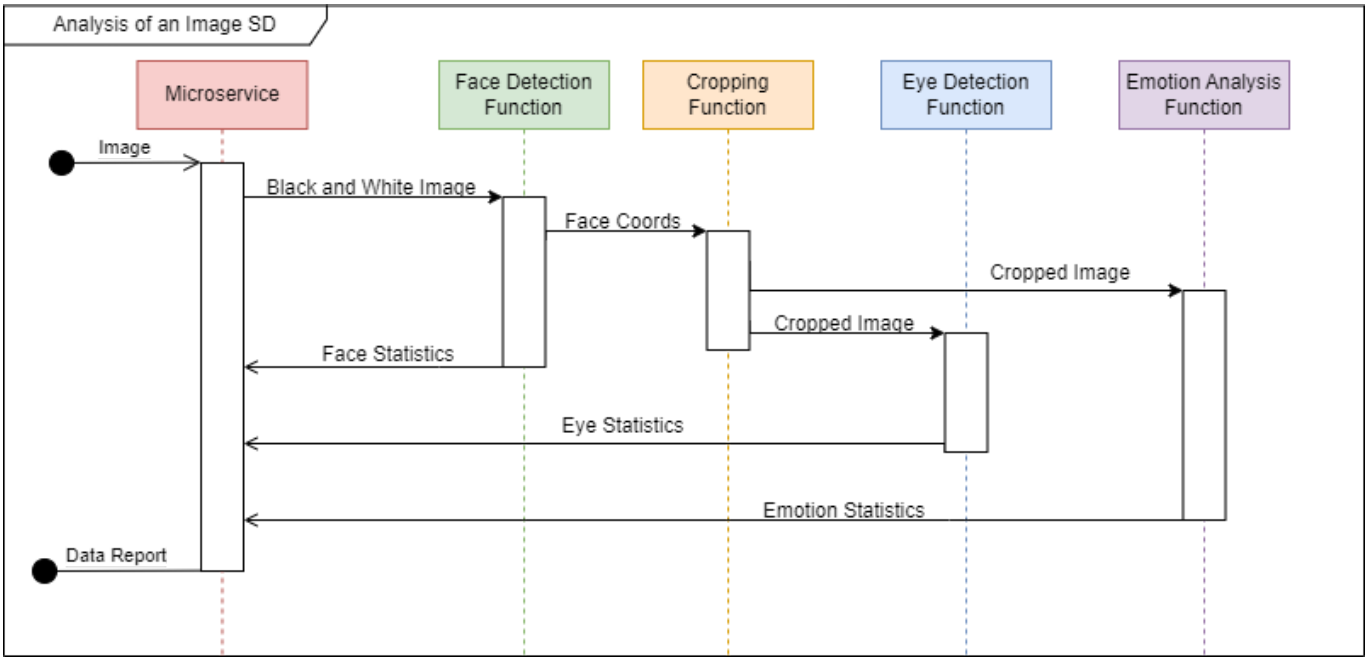
Microservice Architecture

The microservice is responsible for the actual Computer Vision part of the system. It is a core part of the system we designed and it will have several core functions:

- Capture images from the user's camera
- Analyse images using computer vision
- Collect data from each individual image
- Record data overtime of the lecture
- Bundle and send data to storage upon shutdown

The microservice will be started as a subprocess in the backend whenever a user requests to start an analysis session. The microservice's main input then will be images coming from the user, it will have to manipulate these images and run several detection algorithms on them. It will then store the results into a JSON bundle which will be assembled and sent to storage when the microservice is shut down and the analysis session is ended.

Below you can look through a sequence diagram modelled around the microservice.



The sequence diagram depicts the flow of the program whilst an image is being analyzed, this flow will occur each time an image is sent into the microservice from the user which will be one image every second as long as the analysis session is in progress.

You can see that the input to the microservice is an image and the output at the end is a packet of data gathered from the image.

Description of the flow:

- Image enters the microservice
- Image is converted into a black and white image (most detection algorithms work only on black and white images)
- This image is sent into the face detection function which uses facial rec to establish where in the image faces are detected
    - This function also records how many faces are seen and what type, this is returned to the microservice as data
- Then the coordinates of faces are sent into a cropping function, which isolates and crops smaller images just of each face for use in further analysis
- These smaller images which contain just one face each are then fed further into the flow
- First, they enter an eye detection function similar to face detection but with a different model, this function looks for eyes in the images, records where they are and how many and returns this data to the microservice
- The cropped images then also are fed into emotion analysis, this algorithm will analyse the face to determine what emotion the face is feeling, and the results are returned as data
- After all the functions are finished the total data packet is constructed for that image and stored.

Implementation

How did we build it?
The design section above describes what our system should be able to do, and a high-level view of how it would do things. This section describes how we implemented our system and the physical details of how we made each component according to our design. We will go through the libraries and languages we used and give some code samples throughout.

Frontend Implementation

The frontend is responsible for the user interface and rendering of our webpage. We implemented this component using NodeJS. We chose NodeJS because we both had some experience using it before, and knew we could integrate it into a Django backend.

> You can find our frontend in /src/frontend

Within our NodeJS server, a lot of work went mostly into the actual webpage rendering and styling, we kept the Node server relatively simple and easy to use as we didn't want to overcomplicate this component of the system.

In the current edition of our system, the frontend server is hosted through localhost on port 3000, during the demo and if you run it yourself you can navigate to localhost:3000 to see our webpage once the Node server is running.

Within the frontend we used EJS along with regular Javascript and CSS to create our webpage, the EJS and Javascript together account for about 45% of the code in our project according to Gitlab. We chose EJS because a lot of our webpage has to be generated dynamically as data

comes in from the backend, EJS helps us do this as well as offers the basic static elements that HTML does.

Overall we created four main pages within our component, we originally had more but we chose to combine a few of the smaller pages, for example, we had login and register pages that just held the forms for logging in, but we later integrated both of those into the index page using a function and hidden divs, the user could press the login button to swap out the default description on the page to view and use the login form. You can see that function below.

```
// div1 = desc, div2 = one we want to display, div3 is the other one
function SwapDivsWithClick(div1,div2,div3)
{
    d1 = document.getElementById(div1);
    d2 = document.getElementById(div2);
    d3 = document.getElementById(div3);

    if(d2.style.display == "block"){
        // if the block we want is already there, we toggle it off
        d2.style.display = "none";
        d1.style.display = "block";
    }
    else if(d1.style.display == "none"){
        // if the base desc if not there, we know div3 is displayed
        d3.style.display = "none";
        d2.style.display = "block";
    }
    else{ // if base description is there
        d1.style.display = "none";
        d2.style.display = "block";
    }
}
```

The function above allows our index page to display three different divs in the same space, toggling between them at the click of a button.

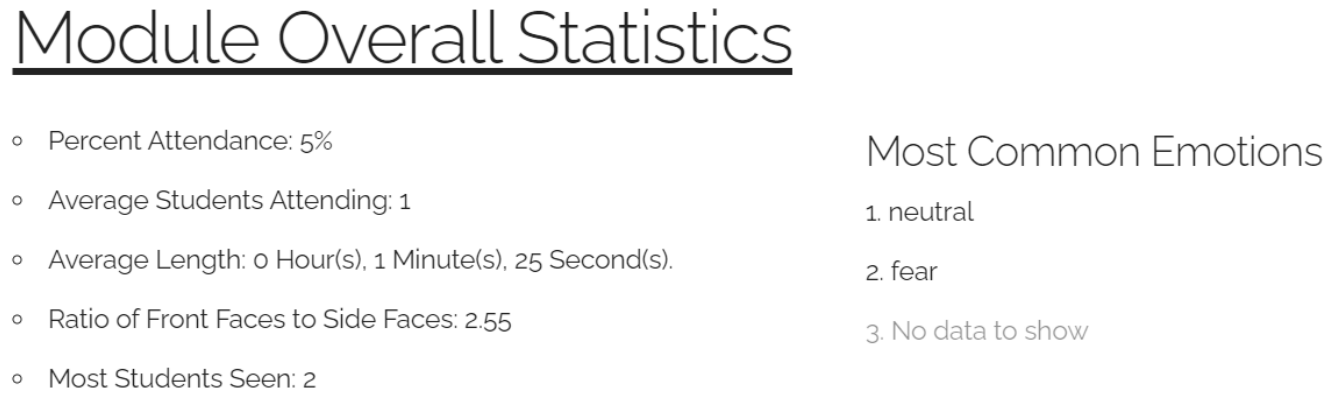
A lot of the work involved in dynamically rendering our pages was done using Javascript, we used fetch requests to retrieve data from the backend to load onto the screen in the form of styled html elements. Below you can see an example of one of our fetches.

```
let req = "http://127.0.0.1:8000/api/lecturerecord";
let index = 0;
fetch(req)
.then(response=>response.json())
.then(data => data.forEach(element =>
{
    let report_data = JSON.parse(element['report_data']);
    if(report_data['mcode'] == code){
        constructReport(report_data, index);
        index += 1;

        total_students += report_data['average_faces'];
        aggregated_stats['attendance_overtime'].push(Math.round(report_data['average_faces']));

        time = report_data['lecture_length'].split(':');
        total_length += parseInt(time[0]) * 3600; //hours
        total_length += parseInt(time[1]) * 60;//mins
        total_length += parseInt(time[2]); //secs

        total_percentage += report_data['average_faces'] / report_data['attendance'];

        total_ratio += report_data['total_faces'] / report_data['total_profiles'];

        emotion_buffer.push(report_data['dominant_emotion']);

        // aggregate results
        if(report_data['max_faces'] > aggregated_stats['max_students']){
            aggregated_stats['max_students'] = report_data['max_faces'];
        }
    }
```

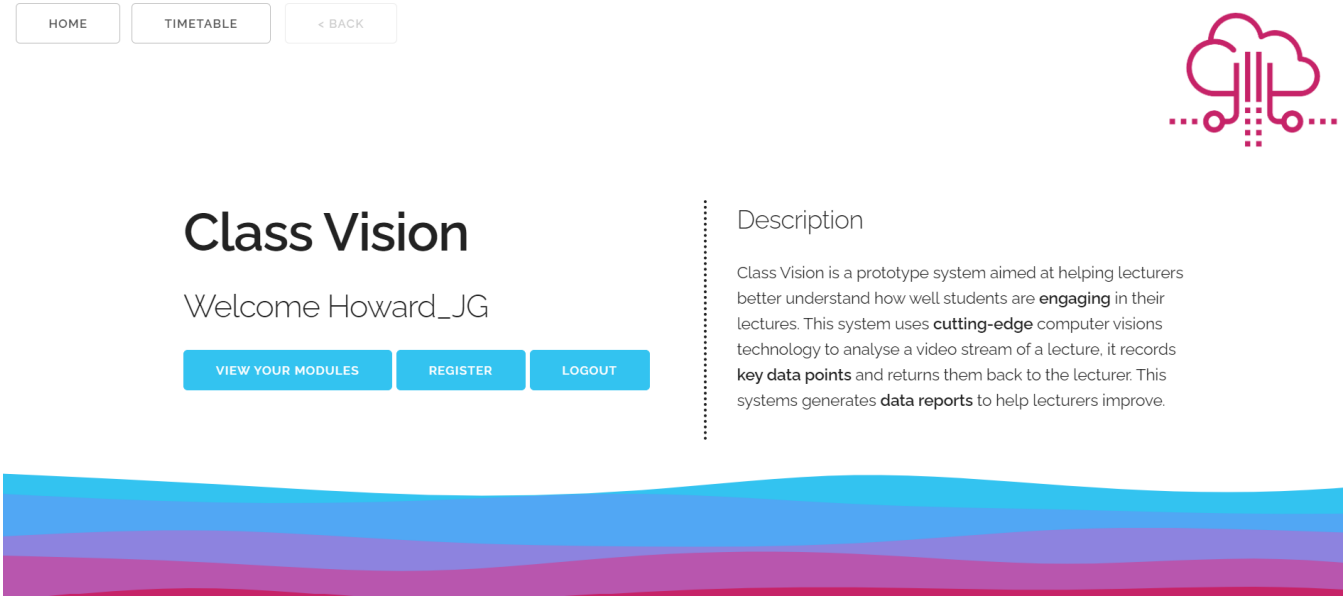
The fetch above is just a small part of our function to display the data reports generated by our Analysis system. The code retrieves lecture records from the database, the data is stored in the form of a JSON object with different data points within it, and the parsed data is passed to a different constructing function called *constructReport()*

> Fact: the constructReport() function is the biggest Javascript function we have, it is in charge of dynamically creating an entire Modal in HTML and plotting a graph using PlotlyJS. This function can be found in *module.ejs*

The rest of the function is about recording the modules' stats overall, combining every individual lecture report into an aggregated stats object which is used to display Module stats and another Graph in our webpage. You can see an example of those stats below.



We have a ton of Javascript in the frontend, it makes up the bulk of what is done in the frontend as defined in our designed use cases, it manages the forms for login and register, it dynamically renders pages for modules and lectures and it also links to our Microservice using a button on the timetable.
If you would like to view the rest of the pages they are all found in */frontend/views/*



A quick note on styling
We decided to keep our website clean and simple, we didn't want to worry about designing a complex UI and styling it, so we kept the page number low and the CSS styling simple.
To help with our styling we decided to use a framework so we didn't start from scratch, we imported the skeleton CSS framework to use, it is a small and very simple framework that doesn't add too much clutter but it makes life a lot easier, it adds a fantastic grid system which helped us align and move things around our page easily. You can view the two CSS files from skeleton (normalize.css & skeleton.css) in */frontend/public/stylesheets*, our own custom styles can be seen in style.css

> You can read more about skeleton here: Skeleton You can see an example of skeleton here: Skeleton Example

Backend Implementation

The backend is responsible for handling all of our data and the functions related to managing this. We implemented this component using Django, as we had experience using it before, and it is well documented online allowing for a wide variety of resources if we needed them.

> You can find our backend in /src/backend

The main focus of our Django backend was handling the data and creating the different functions to interact with it, whether directly through accessing the API or through the use of the frontend website and fetch requests to the backend API.

As we were using Django, our language of choice for the backend was Python. This was a language we had a lot of familiarity with, but also gave us a lot of resources that we could use for it such as tutorials and documentation we could refer to if we ever struggled. Django also has widespread use, which meant that any issues we ran into had a reasonable chance of having been solved already by someone else, cutting down on the time needed to develop a brand-new solution.

Our backend has four main files which were worked on to handle the data being used. These were models.py, which was the basic storage of the different types of data and their details; serializers.py, which validated the data being passed through the system and creates or deletes an object for the relevant model; views.py, which ensures that any access from the API to the serializers has the correct authorisation; and urls.py, which manages the different links of our API and which views should be assigned to each link.

What does the models.py file do? The models.py file handles the different types of data we need to store on our service, and the details associated with each. These are the user data, the module data, the lecture data, and the lecture record data for the microservice analysis. The module model has fields for the module ID, code, module name, lecturer, description, and student count. The lecture model has fields for the lecture ID, the respective module ID, date, day, time, and attendance count for that lecture. Finally, the lecture record data only has fields for the lecture record ID, the respective lecture ID, and a text field for storing the results of the report for filtering later.

```python
class APIUser(AbstractUser):
    pass

class Module(models.Model):
    id = models.AutoField(primary_key=True)
    code = models.CharField(max_length=5, null=False)
    name = models.CharField(max_length=100, null=False)
    lecturer = models.ForeignKey(APIUser, on_delete=models.CASCADE, default="1")
    description = models.CharField(max_length=500, default="There is currently no description for this module")
    student_count = models.IntegerField(default=1)

class Lecture(models.Model):
    id = models.AutoField(primary_key=True)
    module_id = models.ForeignKey(Module, on_delete=models.CASCADE, default="1")
    date = models.DateField(default='2020-01-01')
    day = models.CharField(max_length=10, default="Monday")
    time = models.TimeField(default='11:00')
    attendance_count = models.IntegerField(default=1)

class LectureRecord(models.Model):
    id = models.AutoField(primary_key=True)
    lecture_id = models.ForeignKey(Lecture, on_delete=models.CASCADE, null = False)
    report_data = models.TextField(null=False)
```

What does the serializers.py file do? The serializer.py file handles the interaction with the models in models.py, by validating the data being passed to it before performing the requested function, whether it be simply passing the data back to the frontend or adding or deleting an object in a model. An example of this is the AddModuleSerializer shown below, which validates each of the relevant details for a Module model, before creating a new object in the model with this data.

```python
class AddModuleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Module
        fields = ['id', 'code', 'name', 'lecturer', 'description', 'student_count']

    def create(self, validated_data):
        code = validated_data['code']
        name = validated_data['name']
        lecturer = validated_data['lecturer']
        description = validated_data['description']
        student_count = validated_data['student_count']
        request = self.context.get('request', None)
        if request:
            new_module_item = Module.objects.create(code=code, name=name, lecturer=lecturer, description=description,
            student_count=student_count)
            return new_module_item
        else:
            return None
```

What does the views.py file do? The views.py file handles the permissions of different views associated with different links, such as displaying data, adding new data, or deleting data. It assigns a serializer for each view and can set a permission class for each, which restricts who can access the specified view. In the example views shown below, it can be seen that while anyone can register an account, a user must be authenticated and logged in to be able to add a new lecture.

```
class UserRegistrationAPIView(generics.CreateAPIView):
    serializer_class = UserRegistrationSerializer
    permission_classes = [AllowAny] #No login is needed to access this route
    queryset = queryset = APIUser.objects.all()

class AddLectureAPIView(generics.CreateAPIView):
    serializer_class = AddLectureSerializer
    permission_classes = [IsAuthenticated]
    queryset = queryset = APIUser.objects.all()
```

What does the urls.py file do? The urls.py file handles the different links of our API system, which the frontend interacts with, while also handling the link for our microservice in the backend. The API has links for viewing the different models, registering a new account, and adding or deleting lectures, modules, and lecture records. It also has a link for our microservice, which runs the flask app and redirects the user to a link for accessing that on the backend server.

```
router = routers.DefaultRouter()
router.register(r'users', views.APIUserViewSet)
router.register(r'module', views.ModuleViewSet)
router.register(r'lecture', views.LectureViewSet)
router.register(r'lecturerecord', views.LectureRecordViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
    path('apiregister/', views.UserRegistrationAPIView.as_view(), name="api_register"),
    path('apilecture/', views.AddLectureAPIView.as_view(), name="api_lecture"),
    path('apimodule/', views.AddModuleAPIView.as_view(), name="api_module"),
    path('apiremove/module/', views.RemoveModuleAPIView.as_view(), name="api_remove_module"),
    path('apiremove/lecture/', views.RemoveLectureAPIView.as_view(), name="api_remove_lecture"),
    path('apirecord/', views.AddLectureRecordAPIView.as_view(), name="api_record"),
    path('flask/<str:mcode>/<str:lecture_id>/<str:date>/<str:time>/<str:attendance>/<str:day>', views.open_flask,
    name="start_flask")
]
```

Microservice Implementation

The microservice is responsible for the key part of our system, the facial recognition and analysis component. We designed a lot of the system around this component as it comprised the core part of our design and systems functions. We started development and prototyping of this component very early on the project and spent a lot of time researching it.

We decided to use **Python** as our language of choice and to build the Microservice as a **Flask** app. We chose Python because it is the language we are most familiar with but it also is a very popular language and during our research, we found a lot of examples and tutorials using Python for basic facial recognition scripts. We chose to build our program into a flask app because we knew Flask was easy to learn and great for creating small contained apps and is very flexible and easy to use compared to a larger framework like Django. Flask also allowed us to make a small HTML page separate from our frontend website, this page could be presented to our User during analysis and it could send images back to the flask app for processing.

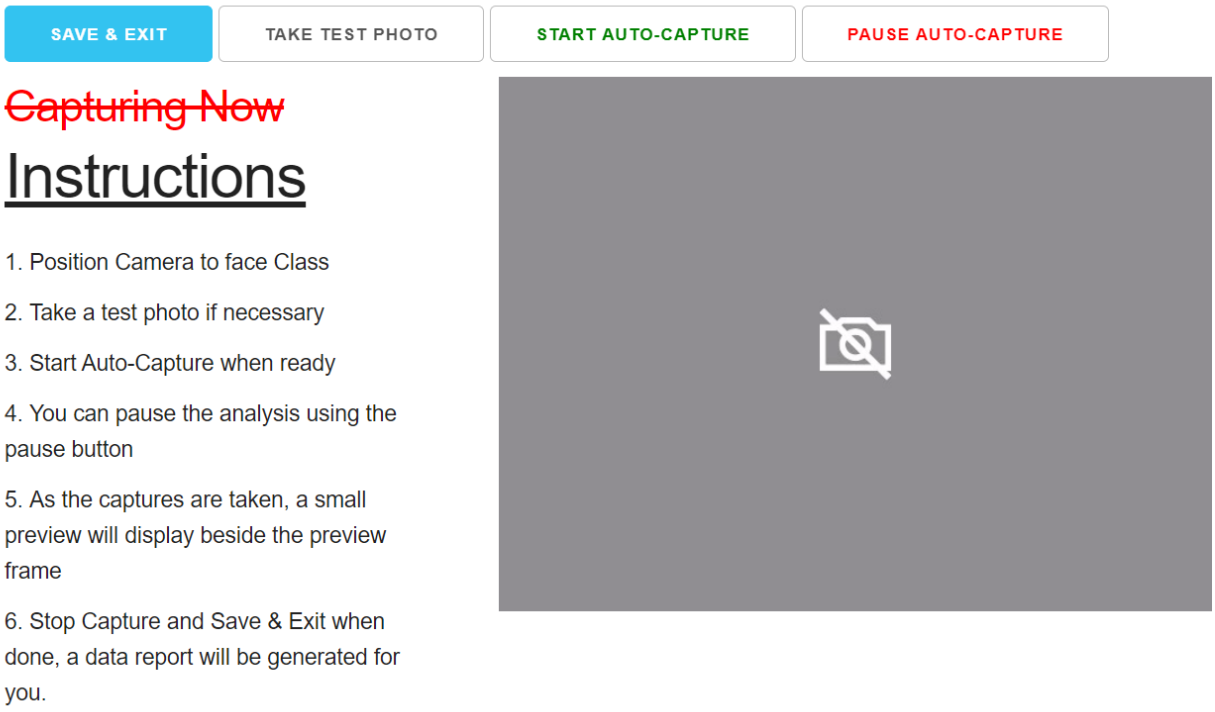
Where is the flask app?
For most of the development of our project, we were prototyping the flask app separately from the webpage components, so when it came time for integration an important decision had to be made about where to put the flask app. The flask app sits within its directory in the backend

/src/backend/flask-app, everything involved in the flask app is self-contained within this directory. This includes the app itself, *app.py*, our static css and js files, the webpage *index.html* and also importantly our three haar cascades used during our computer vision functions.

How does it run? The flask app is set up to not be running constantly, it is turned on every time a user requests an analysis session. On the frontend a user presses a button which triggers an endpoint in our backend, this calls one of our Django views to run the flask app as a python subprocess. You can see the view in the image below.

```
def open_flask(request, mcode, lecture_id, date, time, attendance, day):
    subprocess.Popen(["python3", "flask-app/app.py", mcode, lecture_id, date, time, attendance, day])
    return redirect('http://127.0.0.1:5000')
```

In this Django view we take in lots of data points about the lecture we are going to analyse, these are fed into the app and will become a part of the data report returned at the end of the analysis.
We then call the flask app as a subprocess, passing in all the data points. We need to start it as a subprocess because otherwise it would block our Django backend from working until it ended. As a subprocess, the flask can run in the background while Django still functions. We then return a redirect to a new URL, this URL is where the flask app will be presented, our backend server runs on 127.0.0.1 and the flask app is hosted on port 5000. Below you can see the UI presented to our user by Flask. (The camera is disabled in the below example)



The UI is generated by our html, js and css in the flask app, it presents instructions for the user to be able to use the system. Once the user starts auto-capture the javascript begins calling our *takePicture()* function once per second. Below you can see that function.

```
function takepicture() {
    // tracks how many pictures have been taken
    n_captures += 1;

    const context = canvas.getContext("2d");
    // if the video is there
    if (width && height) {
        canvas.width = width;
        canvas.height = height;
        context.drawImage(video, 0, 0, width, height);
        // takes an image from the video and draws it to a canvas

        const image_src = canvas.toDataURL("image/png");
        // converts the canvas to an image dataUrl
        socket.send(image_src)
        //sends the dataURL via websocket
    }
    else {
        clearphoto();
    }
}
```

The function above is responsible for taking a picture from the user's camera, during normal processing it will be called once a second, sending back the image in the form of a DataURL, the image is sent over a WebSocket connection between the webpage and the python program in our flask app.

How do we process the image

The bulk of the work in the flask app is done by the python script, as seen in the design section, it takes in an image, runs several detection functions on the image and packages up the data into a JSON object which will be sent to the database when the flask app is shut down. **What do we use**

In the python script, we use two core libraries which allow us to do facial recognition, OpenCV and Deepface. OpenCV is the main library we use, we have developed the whole of our system using OpenCV as the base, it provides us with several important functions which allow us to construct our facial recognition algorithm and use the haar cascades on our images. Below you can see the key part of our face detection function.

```
img = cv2.imread(path)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) #change to grayscale

faces = faceCascade.detectMultiScale( #experiment with different numbers in here
    gray,
    # the closer to one, the slower but better, away from 1 is faster but less accurate
    scaleFactor=1.05,
    minNeighbors=5,
    minSize=(20, 20),
    # smallest size object that can be detected,
    # i.e. for eyes we need smaller as they are smaller, maybe reduce this to get faces further away
)
```

In the above code, we first read in our image using openCV, this allows us to work with the image using OpenCV. Then we convert the image from colour to grayscale, this is important as it needs to be black and white for the detection algorithms to work.
Next is the important part, the **detectMultiScale** function is what allows us to apply our haar Cascade to image and detect faces, it takes in the grey image and several parameters. It returns to us an array of coordinates, these coordinates are where in the image faces have been detected. In this code example, we use the faceCascade which detects the front of faces, this code is repeated twice more in the app to detect profiles (side of faces) and then eyes using our two other haar cascades. Next, we draw boxes around the faces and crop them into smaller images, you can see that part of the function below.

```
i = 0
faces_array = []
for (x, y, w, h) in faces: # draw rectangles around faces
    cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)

    crop_path = "./flask-app/faces/crop_face" + str(i) + ".png"

    crop = img[y-10:y+h+10, x-10:x+w+10].copy()
    try:
        cv2.imwrite(crop_path, crop)
        faces_array.append(crop_path)
        i += 1
    except:
        pass
```

The coordinates we received are in form of x, y, width and height. In this part of the function, we use OpenCV to draw a green rectangle around each detected face. We also then crop out each individual face and save them, these cropped images will be used in the emotion detection and eye detection functions instead of the large image.
Below you can see an example of what the image looks like when faces are detected, this example is from the early prototyping days when we fed still images such as this one of Abba into the script.

All this time while the images are being analysed we are collecting data, at the start of the flask app we define a large dictionary for all the data points to be entered into (see below), as the images are processed this data dictionary is filled in. When the flask app is shut down this dictionary is converted to JSON and posted into the backend database for storage.

```
data_captured = {
    "mcode" : mcode,
    "day" : day,
    "lecture_id" : lecture_id,
    "attendance" : attendance,
    "date" : date,
    "start_time" : time,
    "lecture_length" : "",
    "max_faces" : "",
    "min_faces" : "",
    "average_faces" : "",
    "faces_overtime" : "",
    "eyes_overtime" : "",
    "profiles_overtime" : "",
    "dominant_emotion" : "",
    "total_faces" : "",
    "total_eyes" : "",
    "total_profiles" : ""
}
```

What is Deepface Deepface is the other main facial recognition library we used specifically for Emotion detection. Deepface is a facial recognition system created by people at Facebook, Deepface offers a lot easier access to facial recognition, with the use of one function we could have analyzed an image for faces, emotions and more using Deepface however we wanted to learn how to do it ourselves so we used the lower level tools of OpenCV to construct our functions. But we did use deepface for emotion detection as you can see in the code below.

```
def check_emotions(array):
    emotions = []
    for img in array:
        try:
            face_analysis = DeepFace.analyze(img_path = img, enforce_detection=False, actions = ['emotion'])

            emotions.append(face_analysis[0]["dominant_emotion"])
        except Exception as e:
            print(str(e))
    return emotions
```

In this function, we take in an array of images (our cropped faces from before). For each face we use Deepfaces' *analyze* function, to check what the dominant emotion detected is, we then add this emotion to a list and repeat. This list of emotions will eventually be used to display an overall emotion for the whole class.

Shutdown of the App Our flask app will keep running and processing images as long as the user wants, but what happens when the user clicks save & exit. We used flask app routes to handle the shutdown and packaging up of data. In our flask app, we only have three routes that do things you can see the routes in the image below.

```
@app.route('/')
def index():
    return render_template('index.html')

@app.route('/results')
def results():
    print("Results preparing to Send...")
    report_data = prepare_data()
    url = "http://127.0.0.1:8000/apirecord/"

    myobj = {
        'lecture_id': lecture_id,
        'report_data': json.dumps(report_data)
    }

    x = requests.post(url, json=myobj)
    print(x.text)

    return report_data

@app.route('/shutdown')
def shutdown():
    print("Shut Down Initiating...")
    os.kill(os.getpid(), signal.SIGINT)
    return "Done"
```

The default route is the index route, which just loads up the webpage, this is what the user sees when they load up the analysis session. The other two are important for the data collection and shutdown of the flask app. When the user presses *Save & Exit* both /results and /shutdown are accessed one after the other. First is /results, this route is what sends the data into the database, it calls a function *prepare_data()* which finalises the data dictionary we showed earlier, adding the last points of data such as lecture length to it and then returns it to use, then we create a JSON object containing the data and post it off to the backend to be recorded. The shutdown route is then called a second after the results. Shutting down the flask app from inside itself proved to be a bit annoying to do but we figure out this way to do it, with the os module we could get the process ID of the flask subprocess and interrupt it, thus shutting it down.

More Detail The above describes the general flow and important functions of our flask app, the app in total is around 300 lines of code so we couldn't go over everything, you can see the code itself and read our code comments in */src/backend/flask-app/app.py*

Problems and Solutions

Here we are going to talk about some of the problems we encountered during the development of our project, what happened and how we eventually managed to create a solution or work around the problem.

Working with a Camera

Member(s) Affected: Gareth **Problem:** At the very start of the project when I first started researching and setting up OpenCV I had quite a lot of difficulties getting everything working together. I chose to use OpenCV as it was a very popular choice for learning about computer Vision and creating simple programs, there were a lot of different examples and tutorials, and they all suggested it was pretty simple to set up and use, but as often happens, it wasn’t as easy to get it working the first time. The problem I was facing was very hard to understand and I was struggling to understand the errors and forums I was reading to try to fix the problem, I would later learn that the problem I was facing was that WSL (Windows Subsystem for Linux) did not support GUI apps or processes. In my initial tests, my OpenCV code was trying to open a popup window to show its results but WSL couldn’t handle it and the program would crash.

Solution: I went through several iterations of development environments to fix this problem as the development continued, at the very beginning I switched from prototyping on WSL, to programming within a pure Linux virtual machine, so I could use the GUI elements. However, using a VM was cumbersome and a bit slow so I then developed a docker container to run my flask app inside so I could bring it out of the Linux VM, I believed that we would need this docker container in the final system but we didn’t in the end, because with the docker container I had to hardcode in the camera from my machine, so running this on any machine wouldn’t work. My final solution which is implemented in the final system was to use Javascript, with JavaScript I could grab the user's camera via getUserMedia() through the browser. I chose this final solution because it made sure the system would work on any user machine, the Javascript is universal and can grab the user's camera from any browser (as long as they have a camera). This solution

also fit well into the system as then I was able to use WebSockets to easily send the images captured from the frontend browser into the Flask app for analysis.

Deleting Lectures and Modules

Member(s) Affected: Josh **Problem:** I designed buttons on the page for viewing modules and viewing lectures designed to delete a specified lecture or module, through the use of a DELETE fetch call to our API. I tested this during development and found the functionality satisfactory. However, a later test proved that the function was not working as intended in all circumstances. After some more extensive testing, I discovered the discrepancy was that the function was working while logged in as the superuser, while not working for any other accounts that may be created. An HTTP 301 status code would be sent into the console which stated that the link for the required fetch to delete the data was not available, before redirecting the user to another link which sent up an HTTP 403 status code which meant that the link was forbidden unless they had the increased permissions of the superuser.

Solution: Before attempting to engineer a solution to this, I searched the internet to see if anyone had experienced a similar issue in the past. Unfortunately, while I found the same status codes coming up often, it was always concerning an unrelated problem regarding trailing slashes at the end of links. I did some testing but quickly confirmed this was not the issue with the system. The next solution I attempted was changing the permissions for our views to make them more accessible, in the hopes, this would highlight a specific part of the program to narrow in on. This ultimately proved fruitless as well, so I decided the best route would instead be to look into other ways people implemented a similar feature. I realised that the majority of people implementing a similar feature of deleting an object in a Django model would use a PUSH fetch request and simply remove an object through that fetch, rather than attempting to use a DELETE fetch request. After changing my fetch request to PUSH and implementing editing the model through that, I found that the options for deleting modules and lectures now worked on all accounts, regardless of whether they are the superuser with the corresponding permissions.

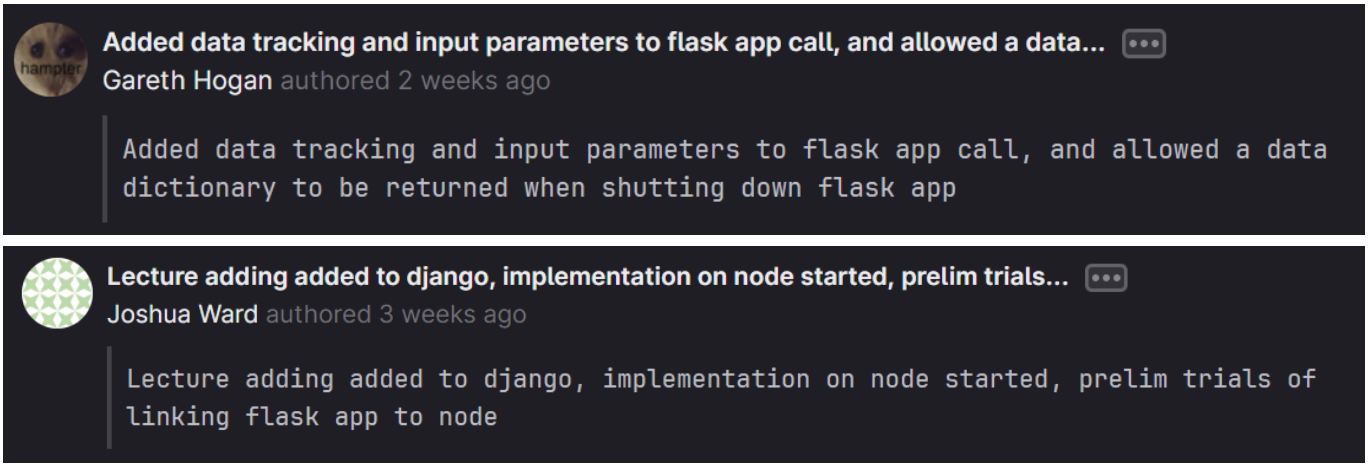
Testing

This section contains an overview of the testing we conducted on our system throughout its development and on the finished system at the end of our project. You can also read about the user testing we did near the end of this section and how we improved the system after the tests.

Git + CI/CD

Throughout our project, we used Gitlab to store and share our code. Every time we completed a new feature or finished work for the day we would bundle up our work and commit it to the repo along with a complete commit message to describe what we had done so that our partner could understand what we had done and if we needed to go back back several commits we understood what was added in each push.
Pushing our work each day helped us as partners work together effectively and understand what we had each done. Gareth tends to work from the morning into the afternoon, whereas

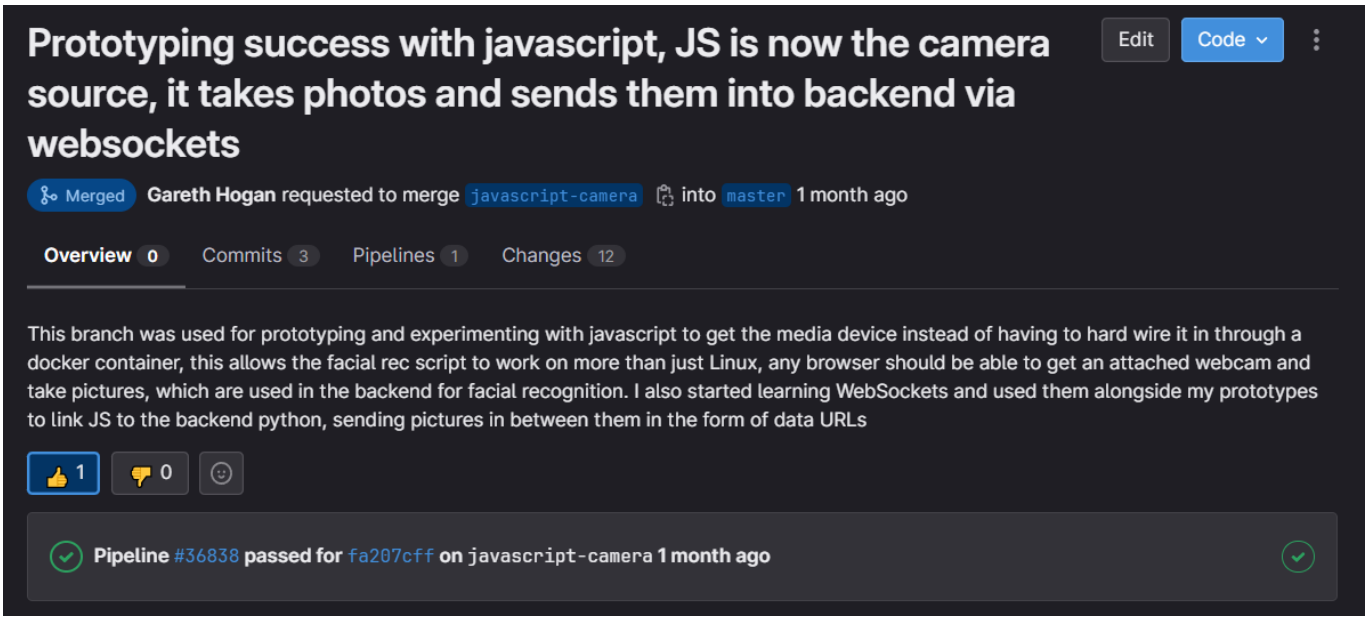
Josh works from the evening into the night so every time we got set up to start individually working we could easily check what had been done this morning or the night before and continue from that point. This tempo was especially important near the end of the project when we both were working on the same files and area of the system.
Below are two examples of commits into our repo.



Git also allowed us to use branches for prototyping and experimentation, this was especially useful for Gareth in the early development of the Camera Computer Vision component. With the Camera component, we had never made something like that before, and a lot of research and prototyping/experimentation was needed so we used branches to experiment with adding new features before merging them to master if we were happy with them and wanted to continue with that idea.

> You can view the remnants of those branches and the prototyping files in our repo *res/prototyping*

One example of a branch we used was the **javascript-camera** branch. This branch was a very important location for prototyping and developing what would become the final solution to the camera problem (Described above in the Problems section).
This branch was made after I had successfully made a docker container which held our prototype flask app which had the computer vision script within it. In this branch we started experimenting with getting the user's camera and taking pictures using Javascript instead of doing it with Python and OpenCV, this proved to be a much easier way to do it and worked very well alongside websockets in JS to send the images back for processing. We were very happy with this solution so we merged the branch into master and you can see in the final system that we still use the solution made in this branch.
Below you can see the merge request to bring this branch into master.



Unit Testing

During our development we didn't do automated unit tests. We didn't do it automated because the setup of our system did not allow it, because our system is mainly about user interaction and runs on two servers.

Instead of automated unit tests, we both made sure to test the components under development at all points during development. Every time there was a change or additional feature added to our code we always ran the servers on our machines and visited the webpage to validate our changes, we filled in as sample users.

For example, while Gareth was developing the flask app and facial recognition scripts he was constantly testing it as he tweaked values and added features into the microservice trialling the script out on himself with his laptop webcam, making sure the algorithm detected his face, eyes and recorded the data correctly.

We also used Postman when testing the backend, specifically when testing our GET and POST requests to see if they were working and that the database was storing out data correctly.

See the image below for some formal descriptions of an endpoint we tested manually using postman during unit tests. During this test we were trying to figure out the exact syntax we need to create our JSON object so that we could post new lecture objects and add them to a module.

Test Name: POST Lecture Details						
Test Author: Joshua Ward						
Pre-condition: Module related to module_id setup in DB already						
Dependencies: Django Server Running						
Priority: High						
Test Case #	Test Summary	Test Steps	Test Data	Expected Result	Actual Result	Status

1	Failed due to wrong JSON syntax, line 3	1. Make JSON Object to post,
		2. Post JSON to http://127.0.0.1:8000/api/lecture/
		3. Observe Results
			{
			"module_id": 1,
			"date": 2022-1-1,
			"day": Wednesday,
			"time": 10:00,
			"attendance_count": 20
			}
				Status Code 201	Status Code 400	Fail

2	Failed due to wrong JSON syntax, line 4	1. Make JSON Object to post, 2. Post JSON to http://127.0.0.1:8000/api/lecture/	{ "module_id": 1, "date": "2022-1-1",	Status Code 201	Status Code 400	Fail
3	Failed due to wrong JSON syntax, line 5	1. Make JSON Object to post, 2. Post JSON to http://127.0.0.1:8000/api/lecture/ 3. Observe Results	{ "module_id": 1, "date": "2022-1-1",	Status Code 201	Status Code 400	Fail
4	Failed due to module ID expecting URL not Int	1. Make JSON Object to post, 2. Post JSON to	{ "module_id": 1,	Status Code 201	Status Code 400	Fail
5	Success	1. Make JSON Object to post, 2. Post JSON to http://127.0.0.1:8000/api/lecture/	{ "module_id": "http://127.0.0.1:80("date": "2022-1-1",	Status Code 201	Status Code 201	Success

System Testing

We didn't have the ability to do automated system testing because of the nature of our sytem, instead, after we had brought all three components together we again filled in as users and did manual system tests by trying out every possible thing on the system that we could, we were trying to prod the system until it broke or encountered a bug.

This did help us spot a lot of bugs, just by going through the page every few days, adding lectures, starting the flask app and pressing buttons we did manage to find and fix several bugs we might have missed.

For example, a major bug we nearly did not find was to do with permissions, while we tested the webpage we both normally were logged into a superuser account we used to test and access the admin Django page, but at one point we registered a new user and tried out some functions, we realised then that non-superusers did not have permission to send Delete requests, so the delete module or lecture buttons were not working. We then were able to fix that bug in the backend.

User Testing

> *Resources and files relating to testing, including results and consent forms can be found in /res/testing-results*

We did do some small formal user tests at the end of our development to test our system and get users' opinions of how it looks and functions.
We did two different user tests:

- Camera/System Functionality Testing
- User Interface Design Feedback

Camera/System Functionality Testing

For the camera testing, we wanted to try out the analysis session with multiple faces in an environment closer to what a lecture would be like. We recruited Gareths' family to help out with

this test. For the test we set up the system with the webcam facing the 4 participants, we then started the analysis session and let it run for around 15 mins. Below you can see the report generated from this "lecture" analysis.

Lecture Statistics

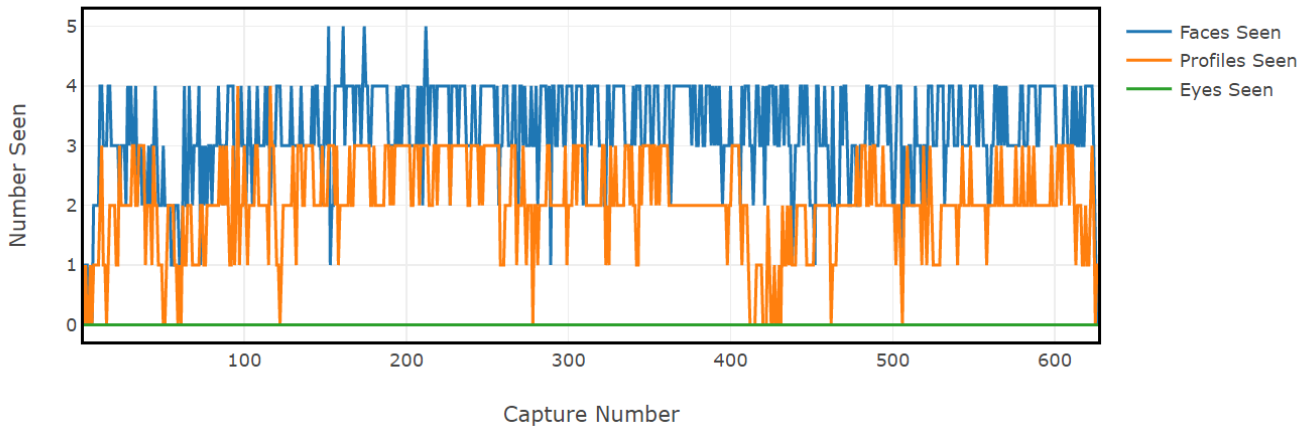
- Lecture Day: Wednesday
- Scheduled Time: 19:00
- Length of analysis: 0:18:54
- Registered Students: 4
- Maximum Faces in one capture: 5
- Average Faces seen: 3.25
- Total faces seen throughout lecture: 2035
- Total Profile Faces seen: 1285
- Total Eyes seen: 0

- Above you can see the raw statistics extracted from your lecture
- The faces seen will not be 100% accurate to student numbers, due to unseen faces or duplicate detections
- An image to analyse is captured every one second whilst analysis is ongoing
- Shown Below is a graph which shows the number of faces, profiles (side-faces) and eyes seen over time

Dominant Emotion = Sad

The dominant emotion is the most frequently recognised emotion from faces captured during analysis. This emotion detection is not 100% accurate and you may interpret the results and patterns in whatever way you feel is right.

Data Points throughout Lecture



We noticed several initial points of interest while setting up the test and immediately from the results.

- The camera quality wasn't the best
- The camera angle was very small and hard to fit everyone in the frame
- The eye detection did not work because of the quality and distance from the participants

These were very easy to spot upon first looking at the results of the analysis. 0 eyes were detected throughout the lecture, we believe this is because the webcam image quality was not great and this combined with how far the participants' faces were away meant that the eye detection did not work. In light of that if we were to continue developing further we would need

to have a look again at the eye detection system and our method, and think about how we could fix the issue.
The hardware issues are kind of out of scope for this project but do present an interesting problem to think about how we would circumvent the distance people sit away from the camera.

Apart from the issues and questions shown above we were quite happy with the results, this was the longest time the system had been continuously running and analysing and it seemed to cope very well, the stats were recorded as expected and given the sample size of 4, we were very happy to see that the average number of faces detected was 3.25 meaning the facial detection was working well.

UI Testing

For this test, we wanted to get opinions from people about the design of the system and how well the user interface functioned. We recruited several of our college friends for this part.

During this test we first presented our system to them, showed them our UI and explained how the system worked and what they would be able to do while using it. During this part, we also let them ask questions and ask about how things work.

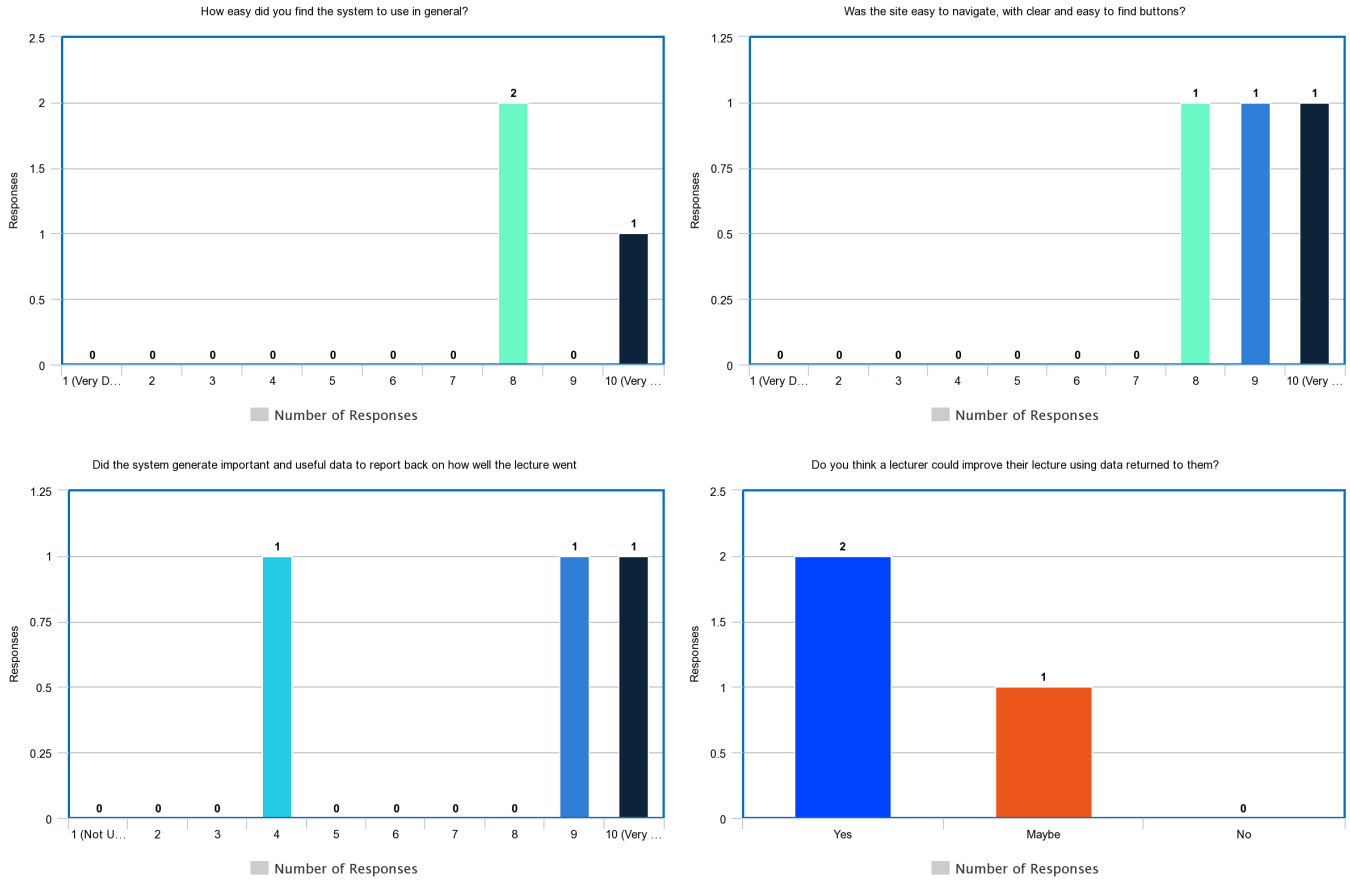
After the presentation, we asked them to fill out our anonymous survey. The results of this form can be found in */res/testing-results*.

Results We were quite happy with the results and feedback we got from this testing, overall for the design and styling of the website, all the participants noted that it looked very nice, the simple website was easy to view and navigate and wasn't too overwhelming to view.

On the functionality side, we got several interesting suggestions on how to make things clearer or to get rid of ambiguity from the UI. For example here are three suggested changes we went through and implemented in the last stages of development:

- Removed the ability for users to register while already logged in, we disable the button like how the "view modules" button is disabled when logged out
- Timetable styling, we improved the styling on the timetable page to make it clearer and easier to read, also linking having buttons to link from a lecture on the timetable to the module page of that lecture.
- Statistic clarity, improved the display of some stats in the reports and module statistics, specifically ones like attendance percentage to better highlight why they were low, for example, the attendance is 5% because only 1/20 students were at the lecture.

Below you can view some of the results we got from our google form, if you want to view the full results file you can find it in */res/testing-results*



Installation Guide

If you would like to try out the system for yourself please follow these instructions:

> *These instructions were made and tested in a fresh Ubuntu environment, your system may already have some of these libraries installed*

1. First make sure you have Git installed (sudo apt install git)
2. Clone our repo
3. Navigate to /src in your terminal
4. For the backend: a. First make sure you have the pip installer **sudo apt install python3-pip**
   b. navigate to /src/backend
   c. use pip to install the requirements **pip install -r requirements.txt**
   d. start the Django backend with the command **python3 manage.py runserver**
5. For the frontend: (Open a second terminal window) a. Make sure you have NPM installed - **sudo apt install npm**
   b. navigate to /src/frontend
   c. Install Nodemon **npm install nodemon**
   d. Start the Node server with the command **npm start**
6. Once both servers are running head to *localhost:3000* to view the webpage

> *Note about flask: when you start the flask service it takes a moment to start, the new tab will say failed to connect briefly before it will display the UI*