Technical Guide

Project Title: Intendi

Karl Duignan: 16105982

Luke Hebblethwaite: 17425212

Supervisor: Dr Michael Scriney

Date created: 02/05/2021

Abstract

This is a document written to cover all technical aspects of the Intendi web application. Intendi is a web application that allows students to watch uploaded lecture videos and provide anonymous and honest feedback, particularly focused on how much students were concentrating during the video. This is achieved through facial recognition and multiple other features and in return provides all the feedback and data to the lecturer in a well designed and easy to understand graphics report page.

1. Overview

Intendi is a web application built for a variety of supported web browsers. The application aims to help lecturers improve and create more engaging course material by receiving honest, anonymous feedback from students and examining what seemed to engage them and what they did not find of any interest. It will also aid in giving the lecturer ideas of when best to upload lectures and know what topics students may be having difficulty understanding. This will allow students to access more exciting and engaging content and create a much more in-depth learning environment.

The lecturer receives highly beneficial feedback from students through our web application that uses AWS Rekognition image processing to analyse their faces while watching the lecture videos, provided they have given permission. Our facial analysis process extracts valuable details such as estimated emotion, eye position, facial landmark direction and other useful facial features. Other data extraction includes determining when students are watching, if they are on the video player window, and if the volume is not muted. With all this, we use our own concentration algorithm to provide an estimated concentration level of the student.

Students' identities are kept completely anonymous. This application also takes away the need for the end of module feedback forms which, more often than not, are ignored by students. Instead, with the use of our web application, students can go about their regular college days of watching lectures whilst also providing critical feedback to their lecturer. This leads to lecturers receiving much more frequent feedback and also feedback for specific videos or topics.

Students can securely sign up with a valid DCU email address and log on to the web application and then join modules they are taking once provided with the module code and password from the lecturer. Once they join a module, they will see course videos that they can then watch. When they click on a video, they are prompted to provide permission for their webcam to capture. Once they accept and press the play button, the facial analysis will begin. The images are then analysed and the data is extracted. This is then reported to the relevant lecturer via the report page to see as anonymised data. The report page features different bar charts and graphs that generate a rich feedback report.

2. Motivation

The basis of the idea came from when we discussed how lecturers don't have the ability to tell whether or not students are interacting with the new pre-recorded lectures. For example, whether or not they are paying attention throughout the video or how easy students found it to stay engaged with these pre-recorded lectures. We then delved into the idea more and realised it could also be challenging for lecturers to know which part of the course interested students. We also noticed how students tend to either forget to fill in an end of semester feedback form provided by the lecturer or are simply unwilling to. Even when a student does fill in the form, they may not be able to provide specific details to a particular part of the module as they may have forgotten about it.

Our idea originally was to use live facial analysis on Zoom lectures. This became a less realistic idea as we soon discovered it would be near impossible to correctly and efficiently use the raw video data provided by Zoom as no access to it is provided by the Zoom API. We then met with our supervisor, Dr Michael Scriney, who helped us change and mould our idea into the one we have today. This involved us driving towards the concept of a space where lecturers upload pre-recorded lectures instead of live ones and then students watch within their own time and pace, as well as the ability to watch multiple times.

In terms of competitors, there is no evidence of any commercial company providing the same sort of functionality, particularly where the aim is to generate feedback from students for lecturers.

3. Research

Why Serverless

Serverless means it is a complete cloud provider which in turn is responsible for executing a piece of code by dynamically allocating the necessary resources. We decided to opt for a serverless architecture at the very beginning as we both agreed they offer great scalability and tend to be much cheaper. We felt this was ideal for the use case of our application if it were to be picked up by a university/college as we would need to ensure that the application had enough resources to handle a large number of users.

Why we used React

For a frontend, we needed a responsive and universal framework. We decided to use React for several reasons. It is widely used in the industry and has excellent documentation. It is also great in terms of performance, particularly as it efficiently updates the DOM. Karl already had some experience with React Native from his project last year and enjoyed using it immensely. Luke wanted to use React after hearing about Karl's experience last year and had read good things online about it. Also, its wide use means that it is a skill we could potentially use in our careers going forward. React also boasts multiple libraries that we found very beneficial to achieve the best user experience possible.

Why AWS

We decided to use Amazon Web Services for many reasons. It is easy to use, reliable and cost-effective. It was also a great way to make sure we had enough resources and options to make the web application completely serverless. Similar to React, it is also widely used in the industry and has extensive documentation. Once again, Karl used it for his third-year project last year and is something that he reported positive experiences with. Luke used Google Cloud for this third-year project, and while he enjoyed using it, he was eager to use AWS as he had not used it before.

AWS Lambda

AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers. Lambdas can be written in Node.js, Python, Ruby, Java, Go and C#. We decided to use Python as it is the language we are most familiar with, and we felt that it was a choice that would make implementation quicker and easier.

Why DynamoDB

DynamoDB is a NoSQL database offered by Amazon Web Services that supports key-value pairs. AWS provides several databases but DynamoDB seemed to be the best match due to the fact that it supports key-value pairs and could be easily integrated with our application.

Why REST API

We chose a REST API throughout the project as we knew we wished to return JSON formatted data, whereas SOAP returns XML. REST API's are also very user friendly and easy to understand. REST API is also ideal for scalability as it creates a separation from client and server. We also felt that REST fits our serverless architecture a lot more due to it being stateless/serverless and seemed like a natural fit. It is also great for interoperability which was useful for any future development.

Why Cognito

Cognito allows you to add user sign-up, sign-in, and access control to your web and mobile apps quickly and easily. We decided to use AWS Cognito as we realised pretty quickly in order to use some AWS services such as invoking API Gateways, the user had to be using a Cognito authenticated token. As we knew that we wished to use AWS Lambdas, we concluded we had to also use Cognito. We were happy with this decision regardless as Cognito handles a considerable portion of the user management process for you, including verification and updating the user pool.

Why Recharts

When creating the charts for the data report page there were many options to pick from, the main two being Recharts and Google Charts. The reason we opted for Recharts was that it was a library made solely for use with React. We also looked at examples from both libraries and together, we concluded that we preferred the look of Recharts as it suited our web application style more and also provided more graphs which we thought looked better and better suited the data and the way in which we wished to display it.

Concentration Algorithm

We first had to research some academic papers to see what defines focused concentration for implementing a concentration algorithm. We found an excellent paper which explains that a good way to calculate concentration is by analysing attention indicators observed by humans and matching them with the observable behaviours, activities, gestures, etc., of the students through screenshots or video (Zaletelj and Košir, 2017). We then decided to work alongside our supervisor in order to determine some common characteristics of a student losing concentration. This included not looking at a screen, muting the video, not being on the same tab as the video, and the direction of multiple facial features.

Video vs Screenshots

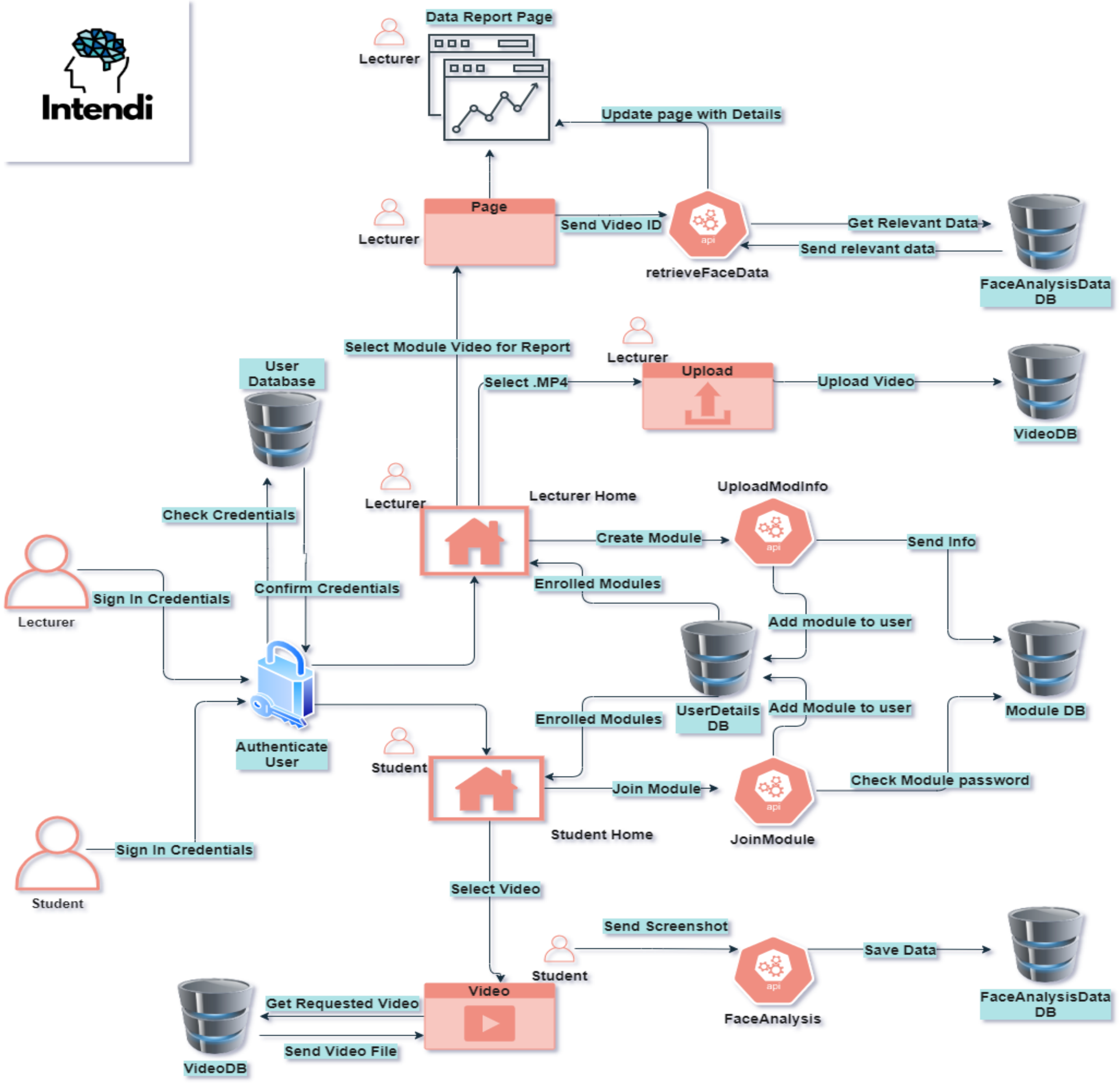
When we first began to think about how we imagined capturing the students' faces, we initially thought the best approach would be to either record them as a full video or even live stream the raw video into an S3 bucket. We quickly realised both of these are very cost-ineffective due to file size and also less efficient in terms of processing time. This is something we wanted to keep to a minimum. We, therefore, decided to opt to take a screenshot every 10 seconds. We felt that an image would suffice for the data we needed. We also settled for intervals of 10 seconds as we wished to minimise costs while also calculating valuable data. We believed that very little in terms of concentration can happen within 10 seconds.

4. Design

The design of Intendi was to make it as simple for both students and lecturers to interact with and create an utterly serverless web application to minimise costs and allow for scalability if it was ever to be brought into practice within a college/university. This section will give an overview of each component and the expected role it plays in the web application at a high level. We will provide necessary information and diagrams for each component to explain its purpose and how it is intended to work.

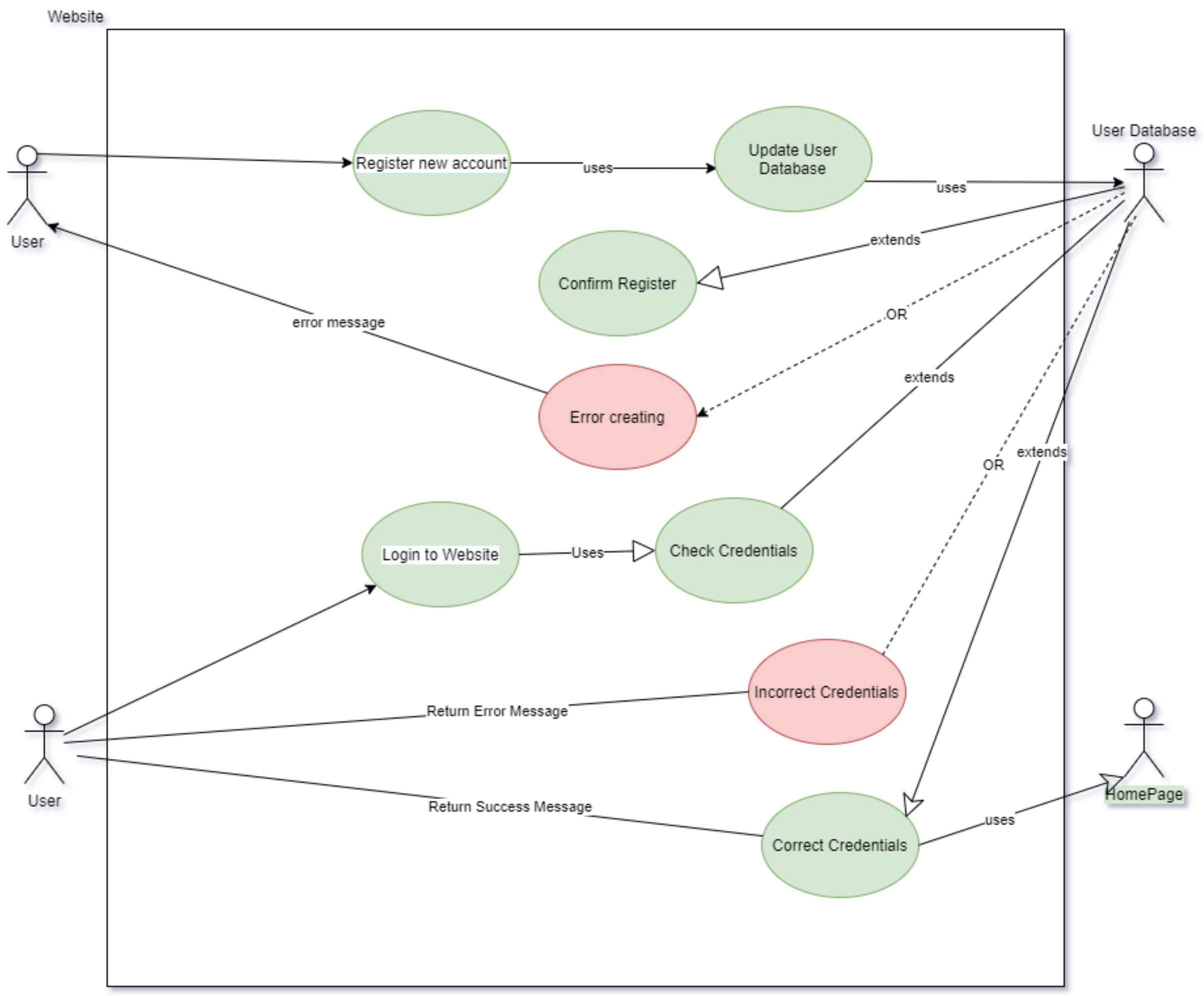
4.1 System Architecture

Below is a high-level system architecture diagram that explains how each piece of the core functionality works and interacts with other components. We will focus on each component within the next section, 4.2 and onwards.

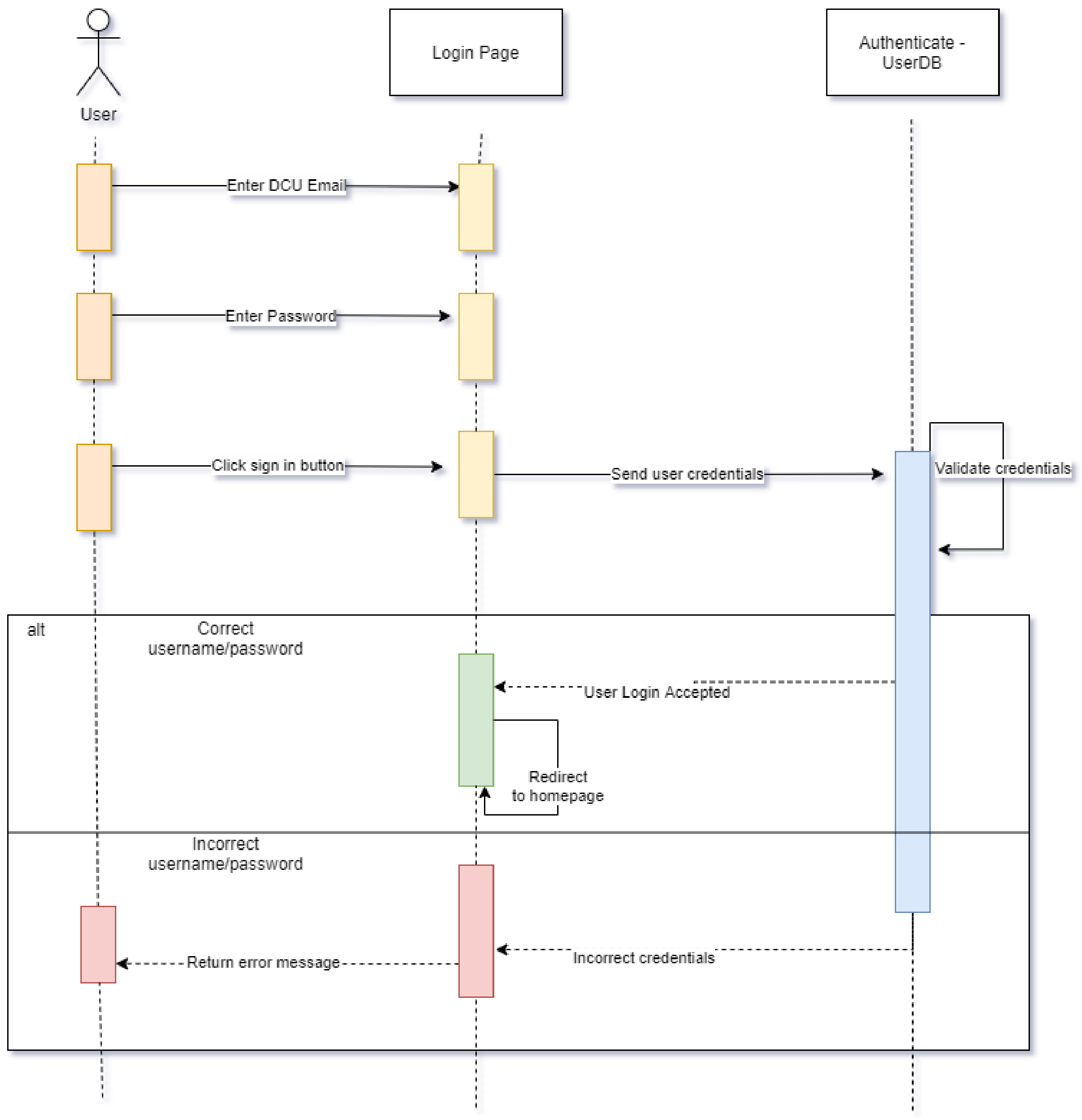


4.2 Register & Login Component

The register and login component is the first point of contact with users. Here is where we wish to be able to ensure users are securely and safely authenticated. This will require them to create an account with a DCU email address and password for our system to securely save in a database and use it as a future reference to users when logging into the system.



Above is a use case diagram for the Register & Login component; as you can see, a user can create an account that will interact with a user database that stores all users details securely. If applicable, the details will be stored and saved in the database. This then leads into the Sign-in component, which again will go and check this database and ensure if the entered credentials are valid and will then allow access to the application’s homepage.

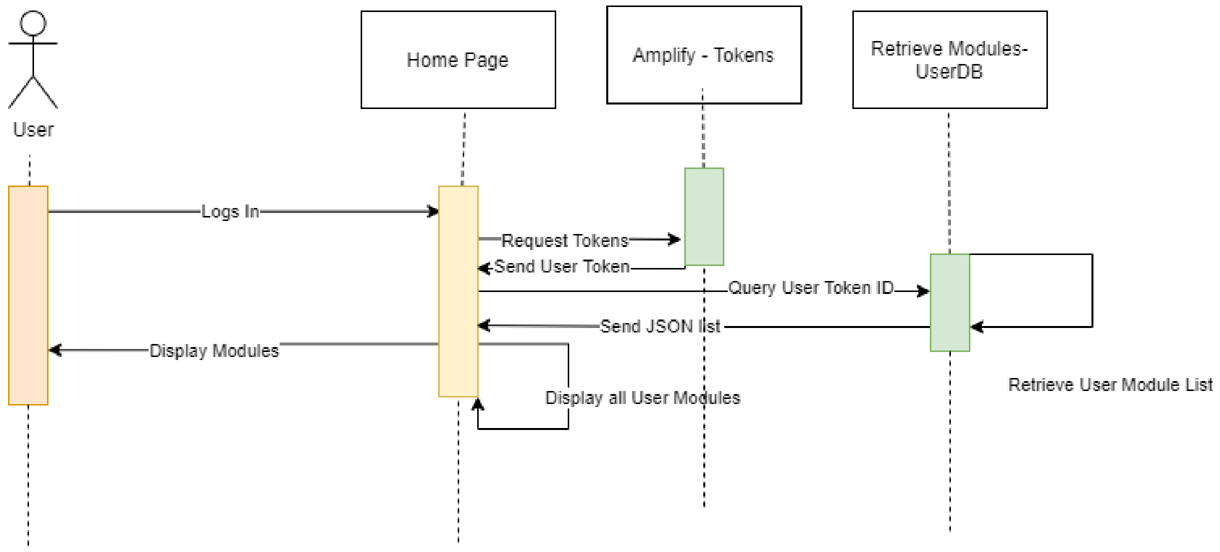


Here we have a sequence diagram showcasing the sequence of a user logging into our web application. We can see they first enter both their email address and password and then once the sign-in button is pressed, this will trigger a lookup in the user database to confirm whether or not the credentials are correct.

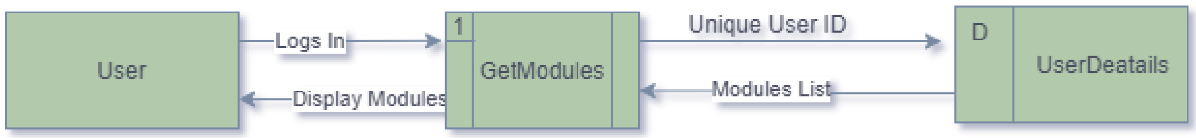
4.3 Home Page Component

The home page is a significant part of the web application as it is customised to each user and allows them to fully utilise the web application as it lets the user interact with their modules. The idea was similar to DCU's own student website, Loop. We imagined that once a user logs in, it would display all their enrolled modules for students and taught modules for lecturers. This would then allow for quick and easy access to their modules, videos and data.

Here we have a sequence diagram showing how once a user logs in, a request to the UserDB is instantly made to fetch all modules that the user undertakes. Once this is completed, the homepage will display all the user's modules on the page to them, allowing them to select one.



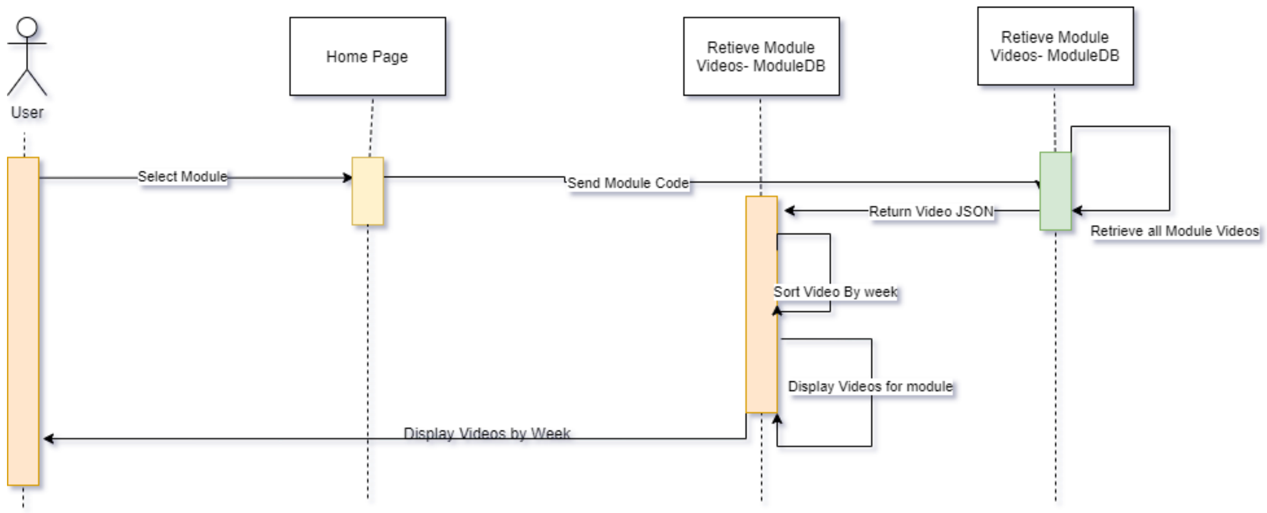
Here we have a data flow diagram showing how once a user logs in, a request to the process "GetModules" is called. This process sends the user's unique ID, taken once logged in, and is sent to the datastore, UserDetails. This will perform a lookup and, once found, return the needed modules list. "GetModules" will then return the list back to the user in the form of the home page.



4.4 Module Page Component

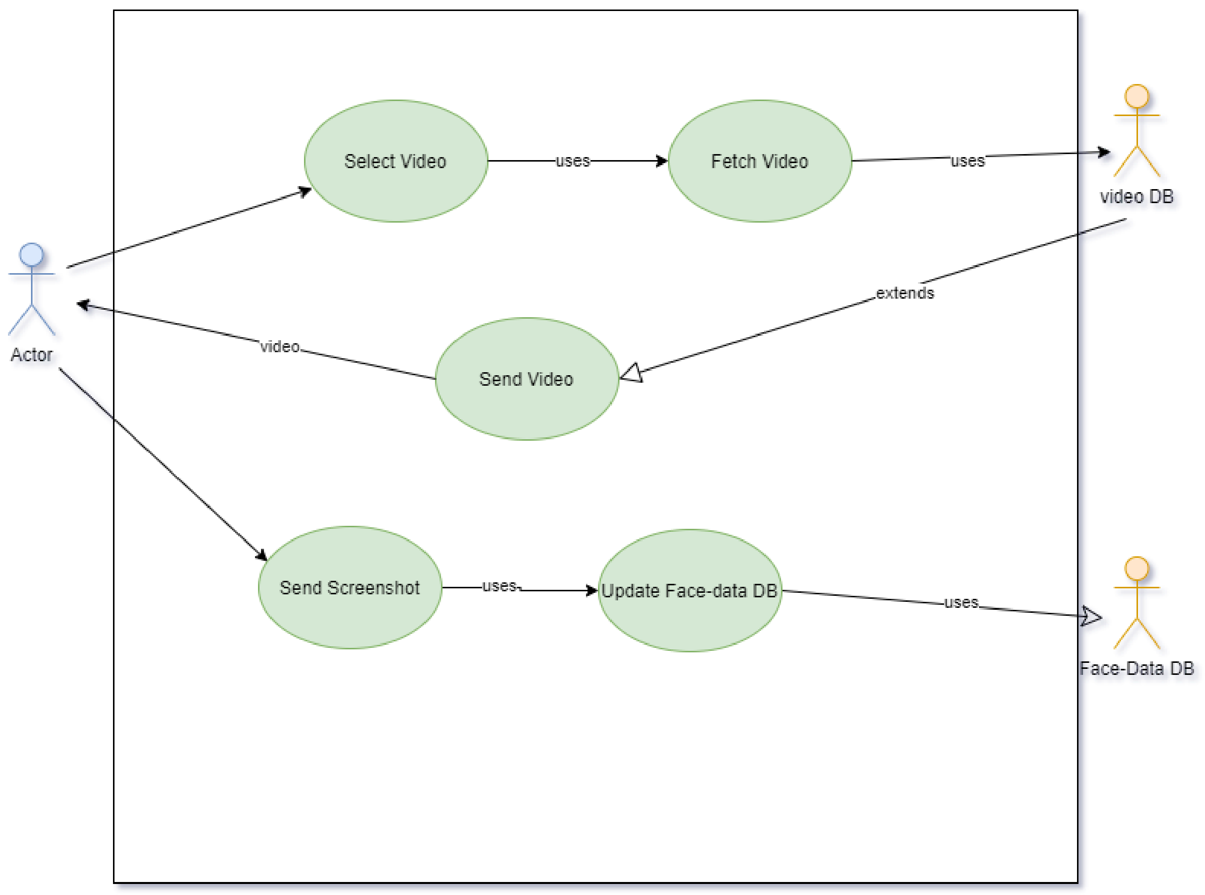
The module page is another vital component as it is the component that returns and shows what videos are available in the given module. This, in turn, either lets the student choose which one to watch or the lecturer choose which one to see a data report on. It was also decided between both of us that it would be best to split all videos by week to make it easier for both kinds of users to distinguish and easily find videos based on what week they are on or wish to go back and view.

Below we have a sequence diagram displaying the sequence of events of the module page. Once a user selects a module, that module code is sent from the home page.


User
Home Page
Retieve Module Videos- ModuleDB
Retieve Module Videos- ModuleDB
Select Module
Send Module Code
Return Video JSON
Retrieve all Module Videos
Sort Video By week
Display Videos for module
Display Videos by Week

4.5 Video Player Page Component

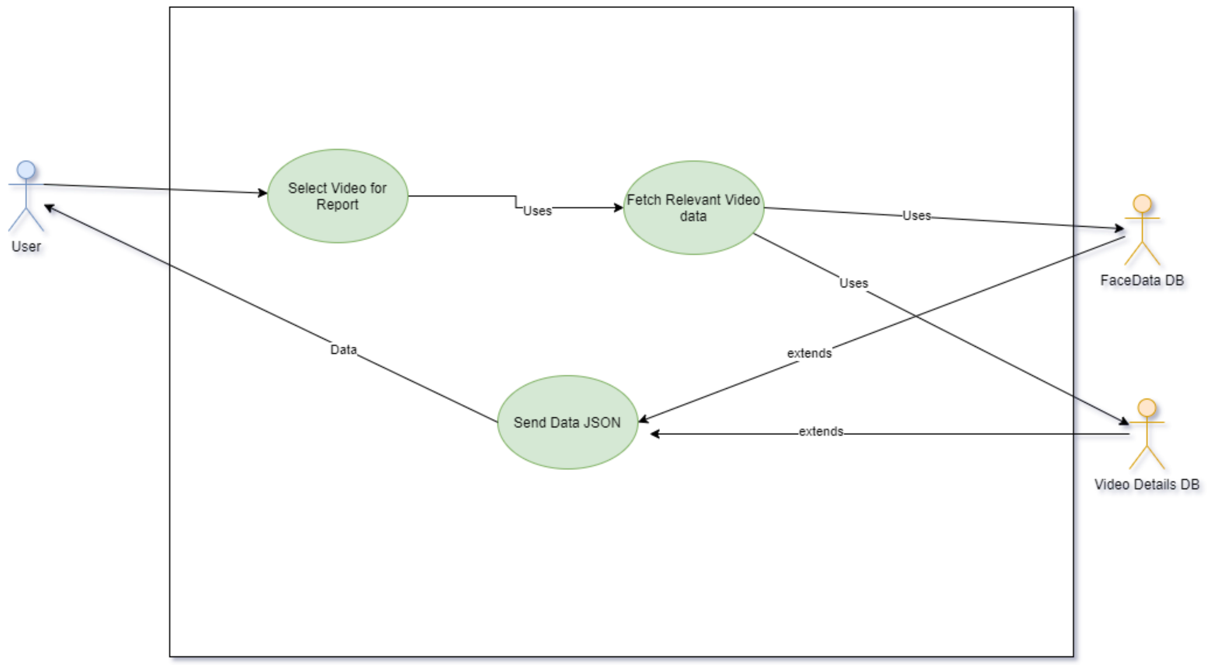
The video player page is the primary source of data for the lecturer as it is where we plan to have the students watching the video and have their face analysed while they watch it. Here we attempt to keep the whole process as streamlined as possible for the student. The idea of the component is that whilst they watch the lecture video, a timer is used to constantly take photos of the student whilst they watch the video. This photo is then sent to be processed and analysed along with multiple other pieces of information such as video timestamp, time of day, player volume and much more. The video will be fetched and loaded as soon as they enter the video player page using the unique video ID passed through from the module page. The process of taking photos is planned to begin as soon as the video starts.



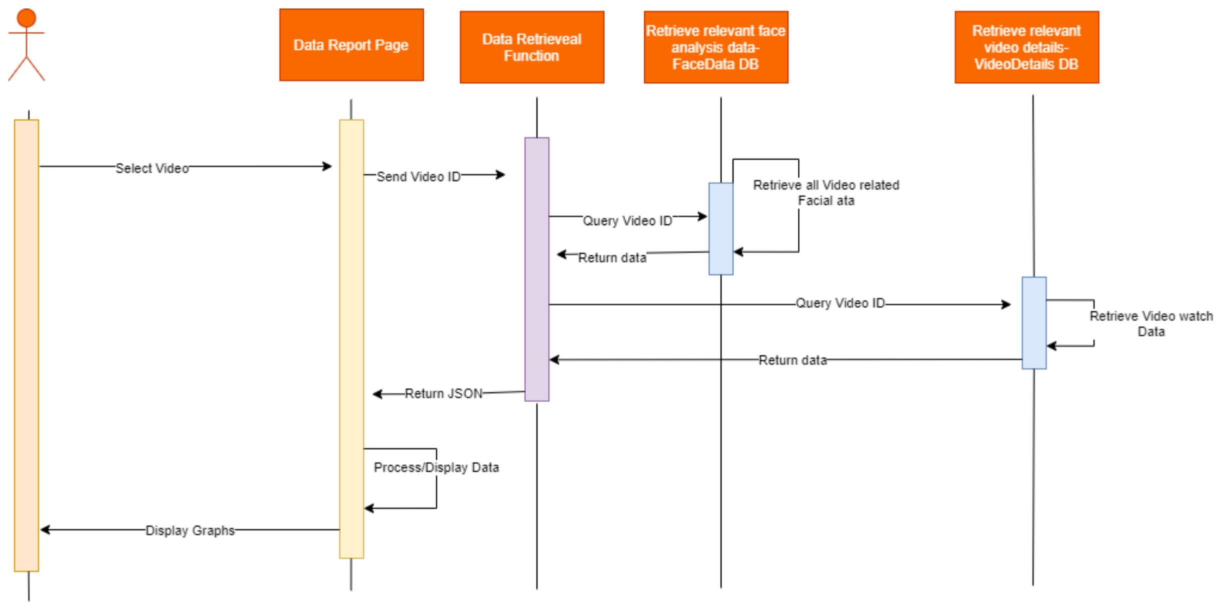
Above, we have a use case diagram that illustrates the basic concept of the video player. Here the user selects a video which then fetches the desired video from the video database. This continues the process where the video is sent back and returned to the user to watch. Once produced, screenshots are sent from the user side to be sent and updated in the face-data database.

4.6 Data Report Page Component

In the data report page, we planned on utilising the whole process of retrieving video relevant data from a database and, once returned, process and display the data in easy to understand graphics. The idea behind this being that once they enter the page, a request is sent with the unique video ID and a behind the scene request is made to query the multiple necessary databases and then return the data in a massive chunk to be processed.



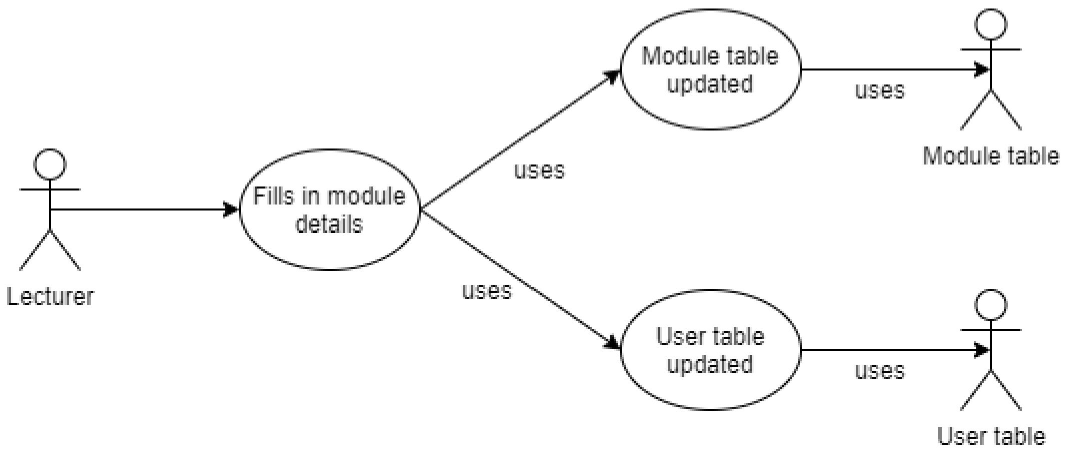
Above we have a use case diagram illustrated, which shows the general purpose of the data report page. We see the user selects a video for a report which uses the fetch relevant video data process that uses the two different databases, FaceData and VideoDetails. These proceed to then extend to deliver and return the data to the user.

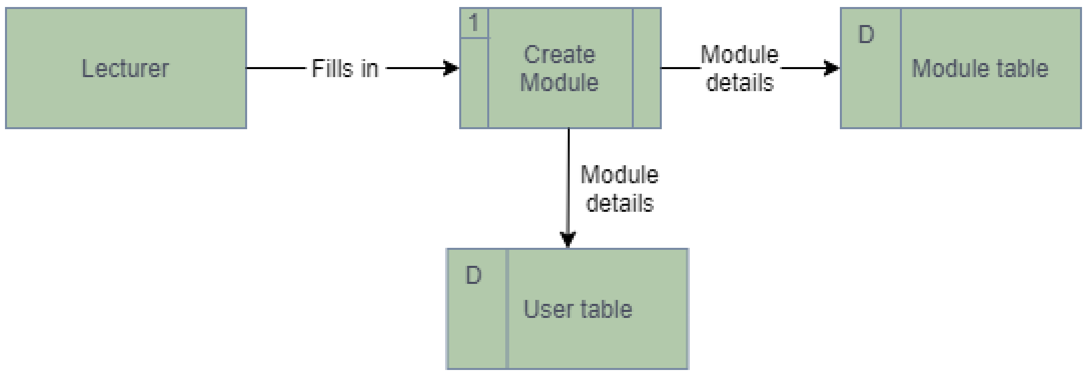


For a more in-depth look at the process and how it is intended to work, we use a sequence diagram to visualise it above. Here we see the user beginning the events by selecting a video going into the data report page. This triggers the next function Data Retrieval by sending the selected video ID. This function, in chained events, begins by a query to the FaceData database. Once returned, that process is ended with the FaceData database and the next database, VideoDetails, is then queried. Once this is returned to the function, this returns all the data as a whole back to the data report page. The data report page finishes its process by processing the received data and inputting it into graphs to display back to the user.

4.7 Create Module Component

The create module page is the page used by lecturers to create a module. The lecturer enters a module code, a module password and chooses a background for the module. This information is then stored in the module details table.

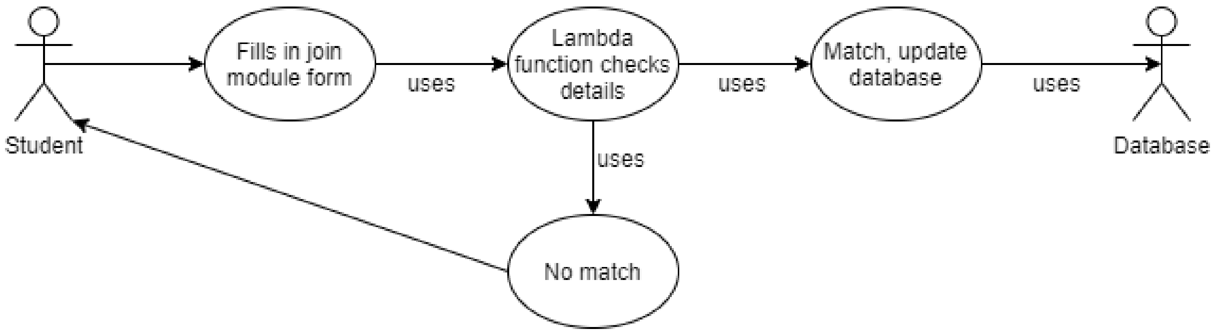


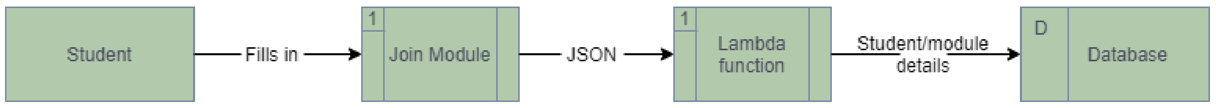


As you can see from the above use case diagram and data flow diagram, the lecturer fills in the module details. This then triggers a Lambda function that updates both the module table and the user table with the relevant module details.

4.8 Join Module Component

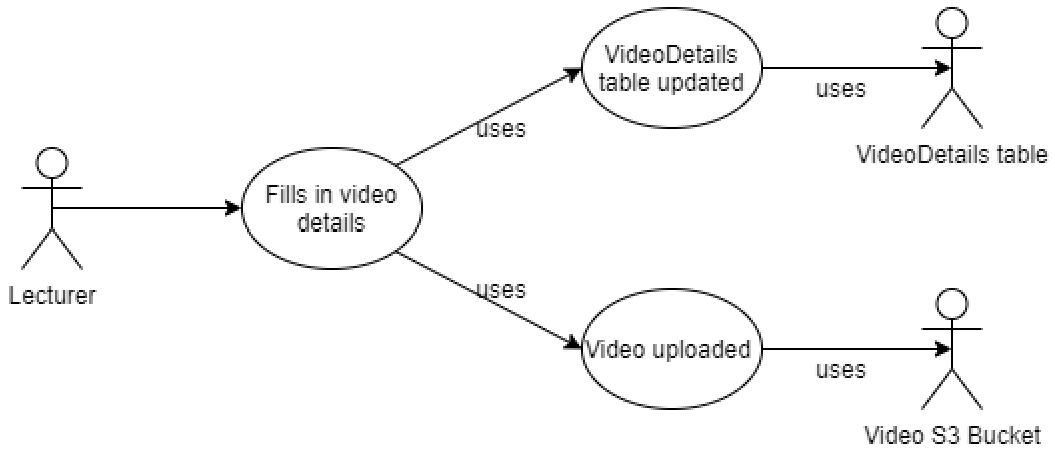
The join module page allows students to join a module provided they have the module code and password. When the form is submitted, a Lambda function is triggered. This Lambda function checks the entered module code and password; if a module is found with the entered password, the student is added to that module. If a module cannot be found with that password, an error message is displayed.

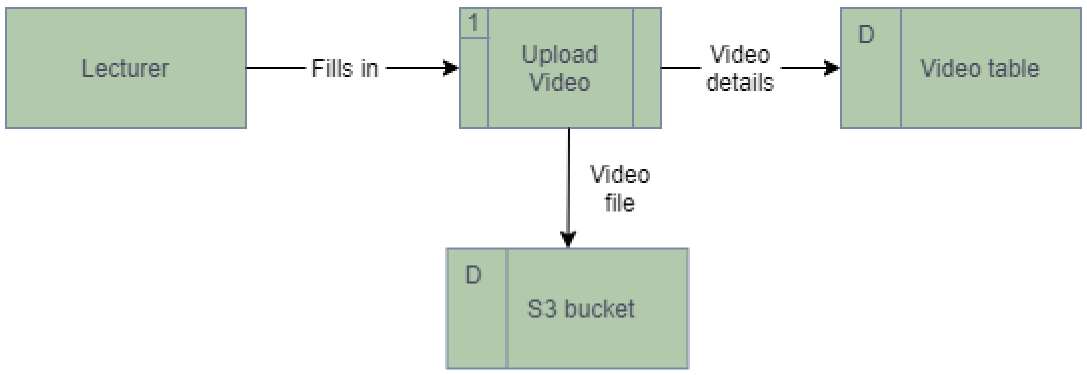




4.9 Upload Lecture Component

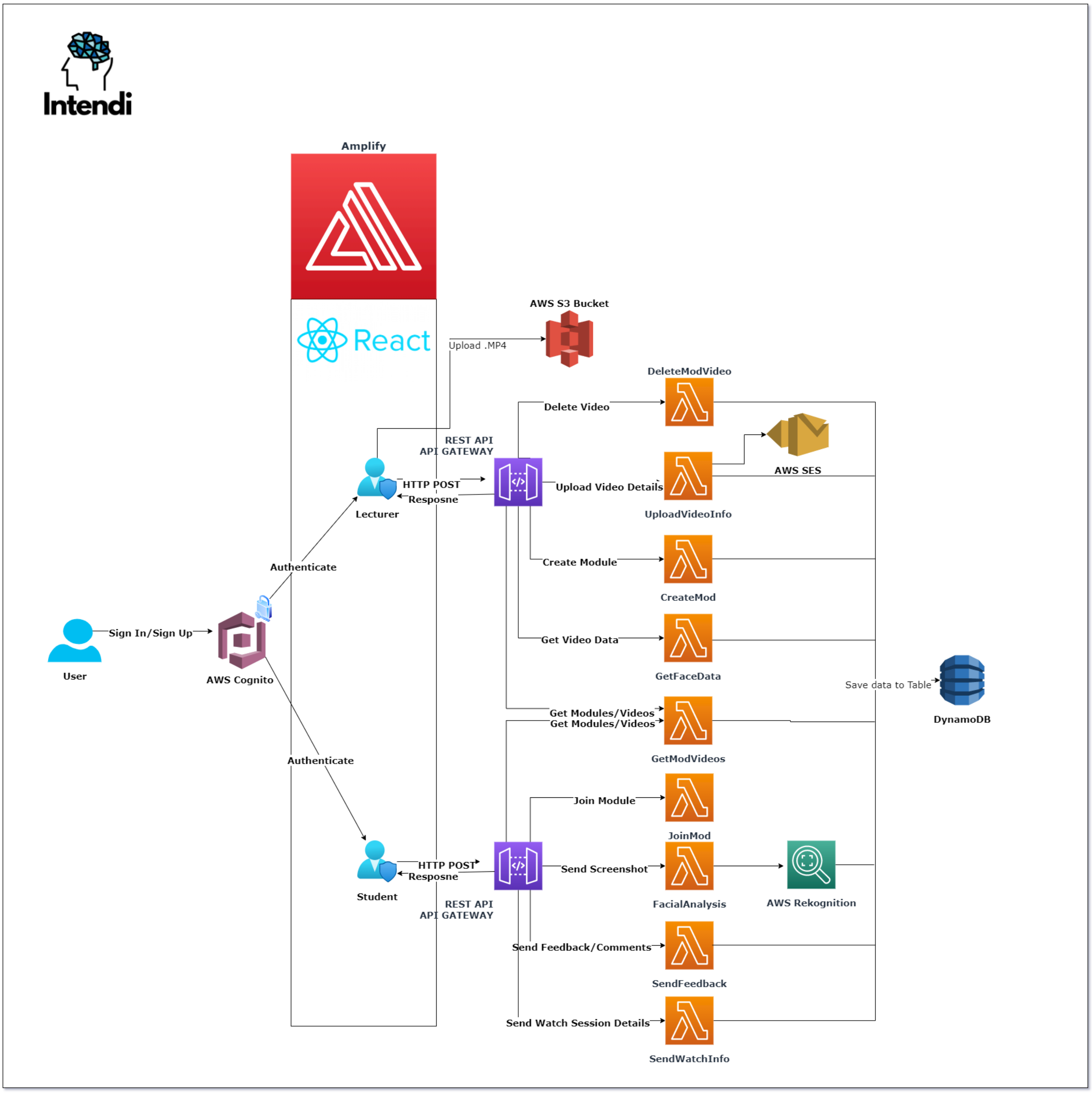
The upload lecture page is used by lecturers to upload lectures. The lecturer chooses a video file from their PC, selects the relevant module from a dropdown list of all their modules, selects a week from a dropdown list of all semester weeks and then types in a video title and video description. The lecturer then clicks the “Upload File” button and the video is then uploaded to an S3 Bucket and the video details are stored in the VideoDetails table.





5. Implementation

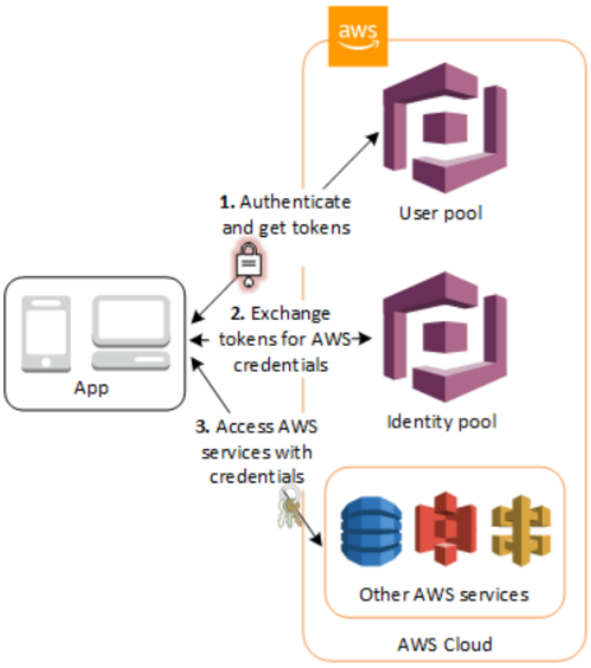
To begin, the project is a React-based web application using AWS Amplify. Amplify allows the project to utilise various AWS tools and services throughout the web application. It will enable the project to become a scalable, secure, fast and efficient website whilst also being completely serverless architecture.



5.1 Login/Register

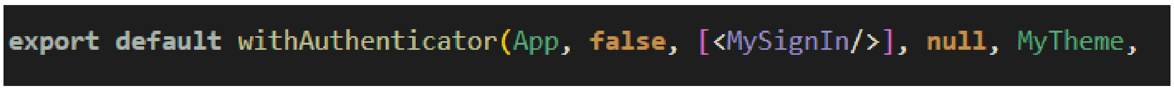
Our whole authentication, authorisation and user management process of the website is powered by AWS Cognito. With Cognito, there is a user pool where any new and existing users are stored securely and their respective credentials. We use Cognito to handle other parts of authentication such as email verification. Once a user has successfully signed up and verified their DCU email, they can log in and, using their authenticated tokens received by Cognito, begin to use all of what Intendi has to offer.

For this project, we decided to focus solely on the use case of DCU. Therefore the website only allows DCU domain email addresses. The website then distinguishes between student and lecturer by reviewing the email address being used for logging in. If the email ends in “@mail.dcu.ie”, we assume it is a student, whereas if the email ends in “@dcu.ie”, we make the assumption that this is a DCU lecturer. This then leads to 2 different sides of Intendi, a student-based side and a lecturer-based side, providing different functionality for both.


aws
1. Authenticate and get tokens
User pool
2. Exchange tokens for AWS credentials
App
Identity pool
3. Access AWS services with credentials
Other AWS services
AWS Cloud

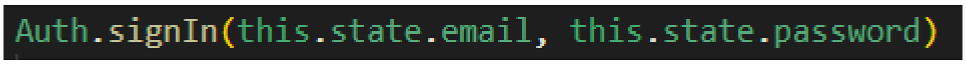
https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html

We implemented Cognito within our application using the AWS Amplify CLI. With this, we were able to create the user pool and decide what permissions users will be granted. The higher-order component “withAuthenticator” was then used within each webpage. This authenticates the user throughout our website once signed in successfully.



```
export default withAuthenticator(App, false, [<MySignIn/>], null, MyTheme,
```

This also provided a default sign-in/sign up form but we decided to create our own component to provide a better user experience. The custom sign-in/sign-up form still uses Amplify to send the standard authentication HTTP requests to Cognito and perform all necessary user pool changes/updates.

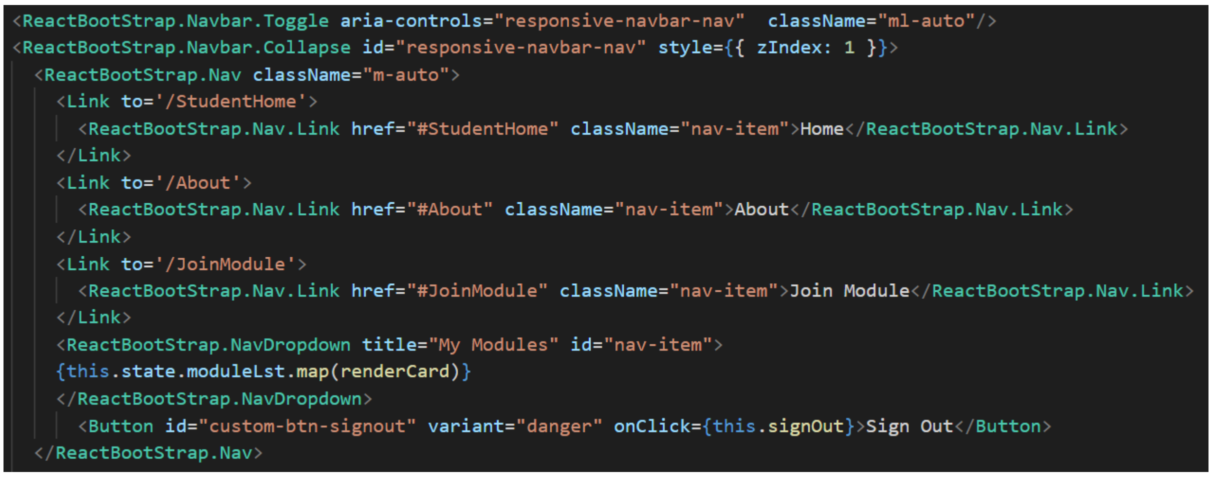


```
Auth.signIn(this.state.email, this.state.password)
```

Similar HTTP requests are made for the forgot password and sign up process. The only difference is that there is an extra layer of security for both, where a six-digit number is sent to the user's email, and they must then verify the six-digit number in the sign-up/forgot password form.

5.2 Navigation Bar

Our navigation bar is situated at the top of our application. Both the student accounts and lecturer accounts have a navigation bar that differs slightly in design. Both the student and the lecturer have Home, About, My Modules and Sign Out on their navigation bar. However, they do differ slightly. The lecturer has Create Module and Upload while the student has Join Module. We decided to use React-Bootstrap in order to have a robust, responsive navigation header. In order to compartmentalise our system further, we decided to create the navigation bars as their own separate components. We created the student navigation bar and lecturer navigation bar as individual components. However, React-Bootstrap merely provides the appearance of the navigation bar, not the functionality. The routing functionality is provided by React Router. Lastly, in order for the My Modules dropdown to work, an API POST request is sent along with the users unique Cognito sub ID. This POST request uses an AWS API Gateway to trigger a lambda function. The Lambda function then queries the UserDetails DynamoDB Table and retrieves the user's list of modules.



```
<ReactBootStrap.Navbar.Toggle aria-controls="responsive-navbar-nav"  className="ml-auto"/>
<ReactBootStrap.Navbar.Collapse id="responsive-navbar-nav" style={{ zIndex: 1 }}>
  <ReactBootStrap.Nav className="m-auto">
    <Link to='/StudentHome'>
      <ReactBootStrap.Nav.Link href="#StudentHome" className="nav-item">Home</ReactBootStrap.Nav.Link>
    </Link>
    <Link to='/About'>
      <ReactBootStrap.Nav.Link href="#About" className="nav-item">About</ReactBootStrap.Nav.Link>
    </Link>
    <Link to='/JoinModule'>
      <ReactBootStrap.Nav.Link href="#JoinModule" className="nav-item">Join Module</ReactBootStrap.Nav.Link>
    </Link>
    <ReactBootStrap.NavDropdown title="My Modules" id="nav-item">
    {this.state.moduleLst.map(renderCard)}
    </ReactBootStrap.NavDropdown>
      <Button id="custom-btn-signout" variant="danger" onClick={this.signOut}>Sign Out</Button>
  </ReactBootStrap.Nav>
```

In the above code snippet, you can see some of our student navigation bar code. There are a number of different tags related to the React Bootstrap navigation bar. A Link tag is a

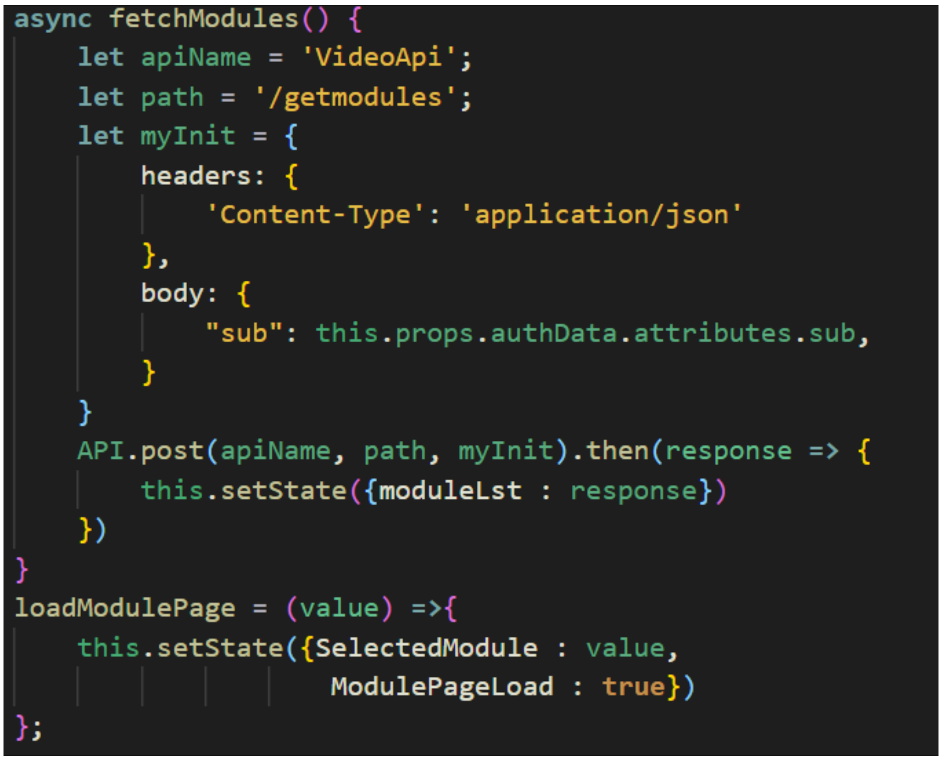
React Router tag that links a navigation link to a particular route. For example, we can see that the Join Module link routes to the /JoinModule route.

After the user signs in, we check whether the user is a lecturer or student and based on that we display the relevant navigation bar.

5.3 Home Page Component

The home page component is the first page that the user sees after signing in. Once signed in and when the component/page is mounting, an API POST request is sent along with the user's unique Cognito sub ID.

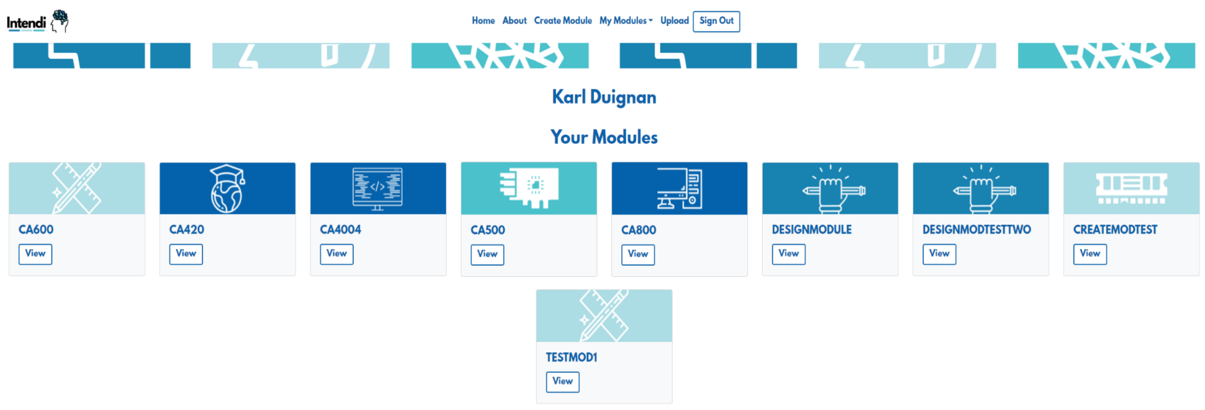
This POST request uses an AWS API Gateway to trigger a lambda function.



```
async fetchModules() {
    let apiName = 'VideoApi';
    let path = '/getmodules';
    let myInit = {
        headers: {
            'Content-Type': 'application/json'
        },
        body: {
            "sub": this.props.authData.attributes.sub,
        }
    }
    API.post(apiName, path, myInit).then(response => {
        this.setState({moduleLst : response})
    })
}
loadModulePage = (value) =>{
    this.setState({SelectedModule : value,
                    ModulePageLoad : true})
};
```

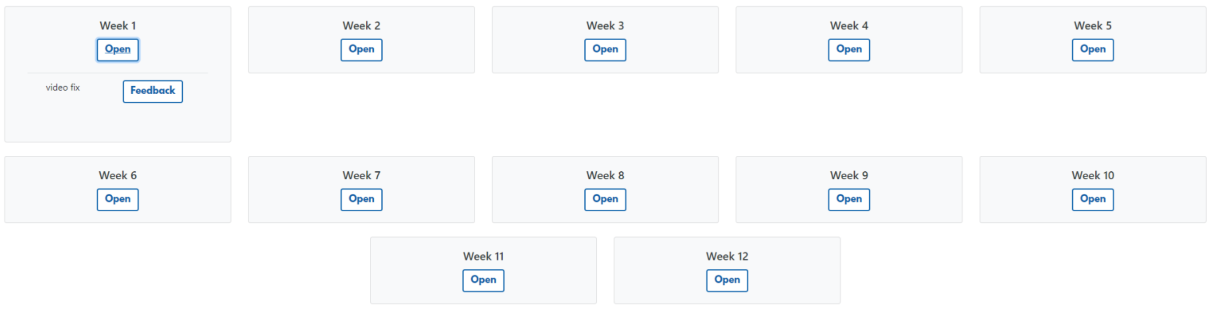
The Lambda then queries the UserDetails DynamoDB Table and retrieves the user’s list of modules. Within the list, along with each module, is the background image URL chosen by the lecturer when creating it. This allows us to know which background to load from the S3 bucket on the front-end. This is then sent back as a JSON response to the front-end.

From here, the JSON list is set to a state and mapped through and then each module is rendered on a react-bootstrap card in order to display each enrolled module neatly on the user’s screen.



5.4 Module Page Component

The module page is similar to the home page except it is module-specific and instead returns all the uploaded videos for a particular module. Once a user clicks on the View button of a module on the home page, they will be brought to the Module Page. Similarly to the Home Page, an API POST request with the selected module code is sent to the API Gateway, invoking the GetVideos Lambda. This queries the VideoDetails DynamoDB Table using the module code and finds the unique video ID code and other details such as video title and week number. Once these are all found, they are then sent back as a JSON response to the client to then parse through and display in cards ranging from week 1 to 12, with any corresponding videos within each expandable card. Beside each video is also a button to go in that specific video.



Depending on whether a lecturer or student logged in, they will either see a “Feedback” button or a “Watch” button, respectively. Depending on the user type (student or lecturer), the “VideoShow” state will change true when either button is pressed and, in turn, will render either the video player page (for student) or data report page (for lecturer) within the module page of the video the user clicked on.

5.5 Data Report Page Component

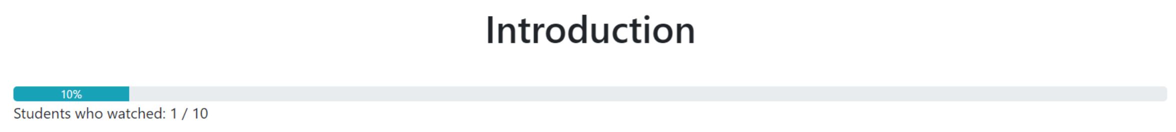
The data report page is where the core functionality of Intendi is seen. This is only accessible through a DCU lecturer account, and each lecturer can only see data for their taught modules. As the idea behind Intendi is to keep everything anonymous, no student-specific data is displayed to the lecturer, so everything is kept to complete anonymity.

The data report page can be viewed once a lecturer clicks on the feedback button alongside any of their uploaded videos to any of their taught modules. From here, the data report page is rendered. This instantly sends an API call to retrieve all available data that is related to the selected video they wish to have data on. All that needs to be sent in this API is the video's unique ID to allow the Lambda to find all relevant data within multiple DynamoDB tables, particularly the VideoAnalysisData Table. This is where all the AWS Rekognition face analysis data is stored.

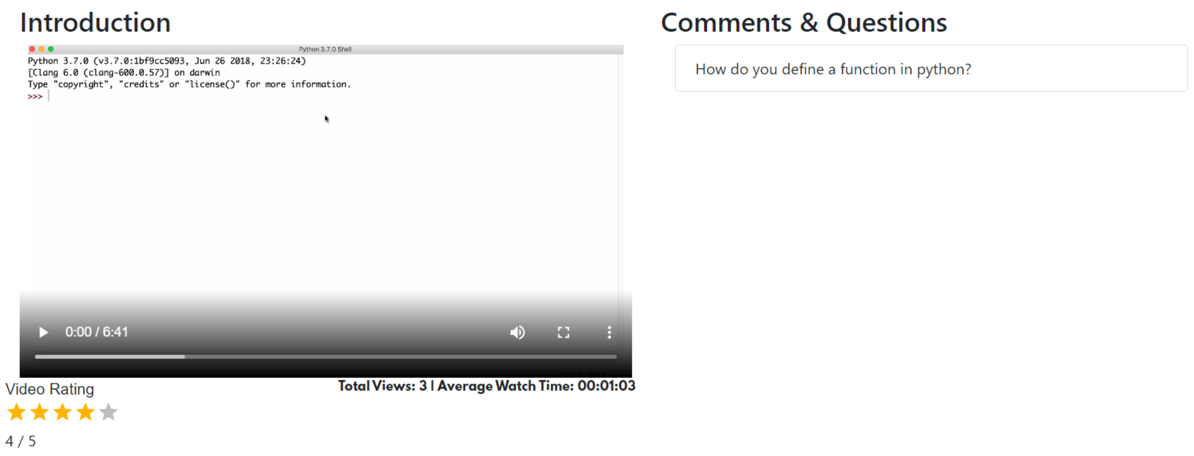
The Lambda retrieves a magnitude of data such as the facial analysis data, which in turn calculates the concentration level, the timestamp of the lecture video of when the data was taken, the video's average rating, average watch time, any comments, how many students watched (unique and collectively), date of watches, day of the week that the video was watched, the time it was viewed and much more.

This is all then returned to the client-side to be parsed, calculated and inputted into clean, easy to understand charts.

On the front-end, we first have the video title, followed by a percentage bar which gives a quick idea to the lecturer how many students out of the total enrolled amount have actually watched the video.



Underneath this, we can see the video itself if the lecturer wishes to watch it. The underneath the video is the video star rating which is voted on by the students who have watched the video as well as how many total views the video has and the average length students tend to watch the video for. To the right side of all this, we have the comments and questions section where if any student submitted a comment or question at the end of the video feedback form, it would appear here for the lecturer to see.


Introduction

```
Python 3.7.0 Shell
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

Comments & Questions

How do you define a function in python?

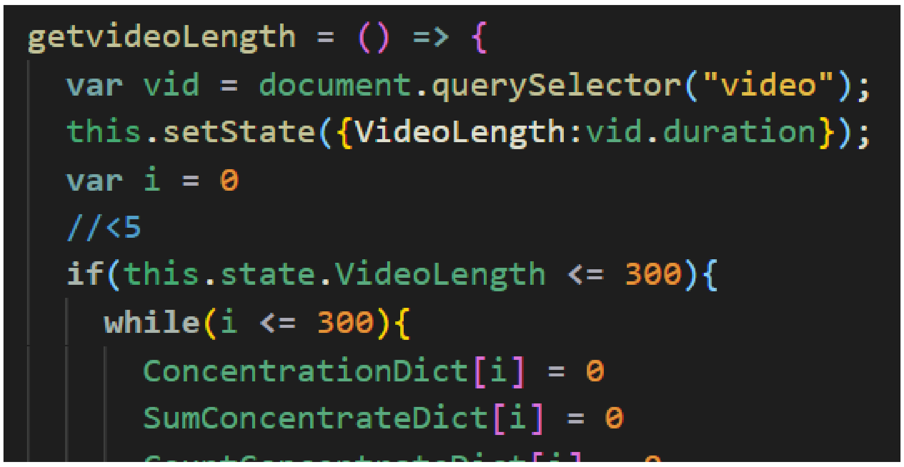
Video Rating

4 / 5

Total Views: 3 | Average Watch Time: 00:01:03

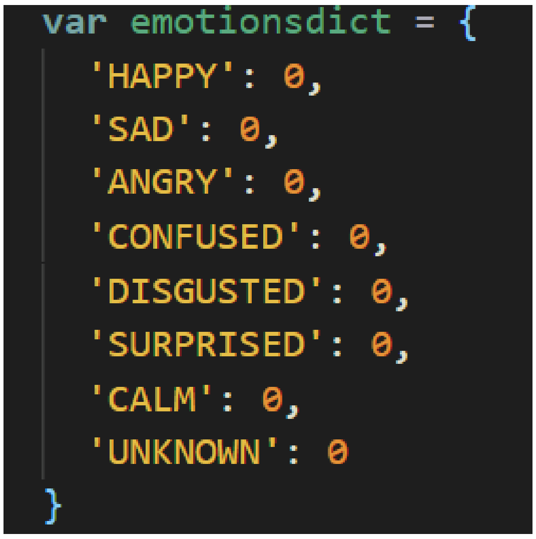
For the charts, we used the Recharts library in order to create the interactive and clean look of the data report page.

Before all was put into the charts, many types of data processing had to be done to create the charts accurately. This included first returning the lecture video length to dynamically add intervals of 30 seconds to the necessary charts to account for different length videos.



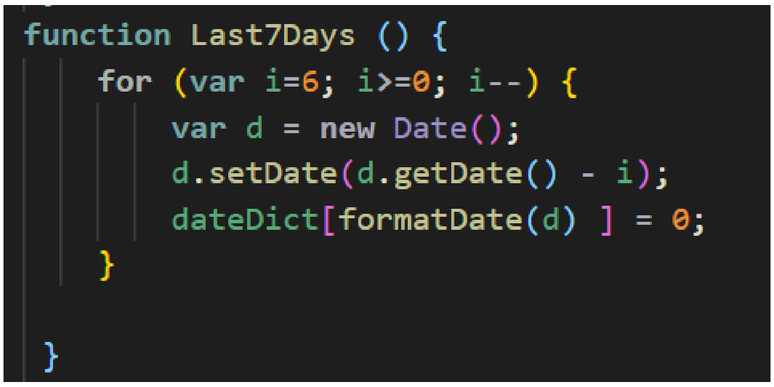
```
getvideoLength = () => {
    var vid = document.querySelector("video");
    this.setState({VideoLength:vid.duration});
    var i = 0
    //<5
    if(this.state.VideoLength <= 300){
        while(i <= 300){
            ConcentrationDict[i] = 0
            SumConcentrateDict[i] = 0
```

Multiple dictionaries were used to total up certain pieces of data such as the total amount of each kind of emotion, totals for different watch lengths, time of day watches, average concentration throughout and much more.



```
var emotionsdict = {
    'HAPPY': 0,
    'SAD': 0,
    'ANGRY': 0,
    'CONFUSED': 0,
    'DISGUSTED': 0,
    'SURPRISED': 0,
    'CALM': 0,
    'UNKNOWN': 0
}
```

A function to also calculate the last seven days dates starting from the current date was also used in order to look out for any views within this timeframe.

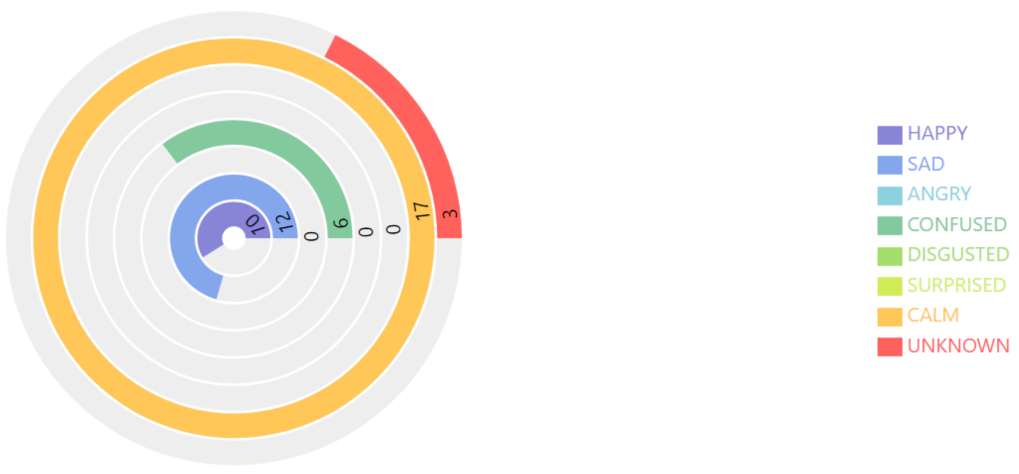


```
function Last7Days () {
    for (var i=6; i>=0; i--) {
        var d = new Date();
        d.setDate(d.getDate() - i);
        dateDict[formatDate(d) ] = 0;
    }

}
```

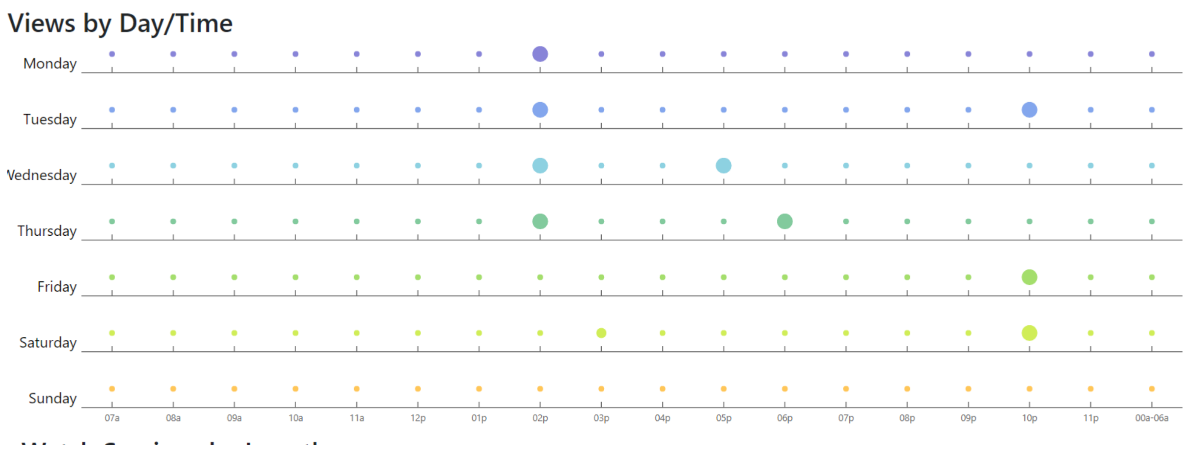
Once all these lists/dictionaries were set and ready to go, it was a matter of looping through the JSON response of all the data and creating the necessary data input for each different recharts chart.

These graphs include:

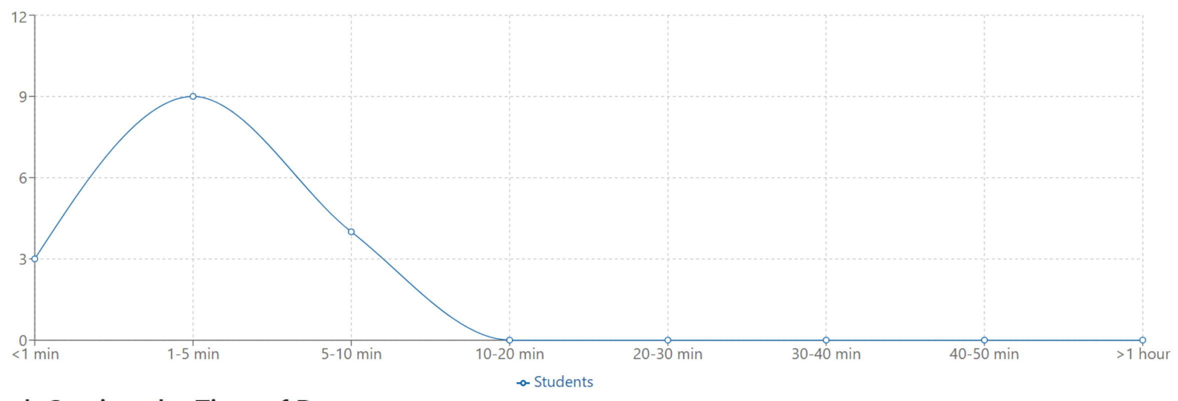
A breakdown of all emotions recognised throughout the video



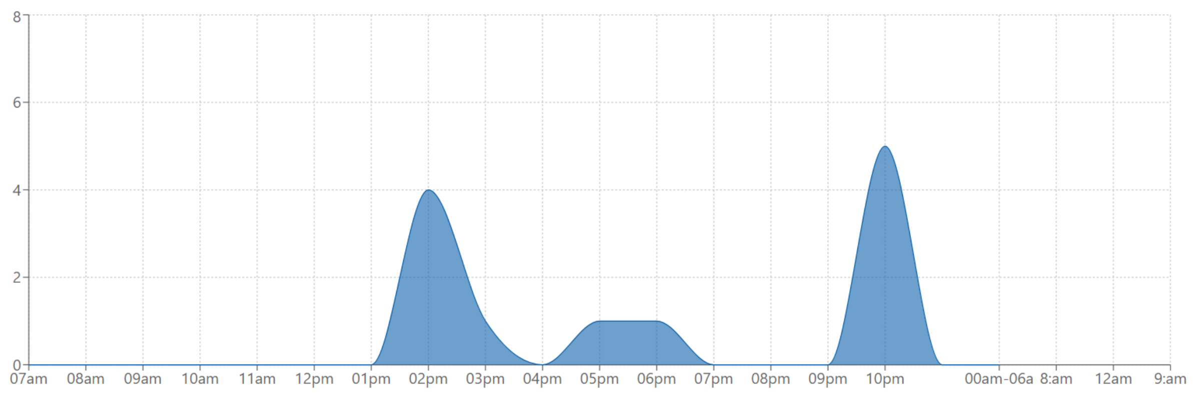
A breakdown of days and time of day of when students have watched the video



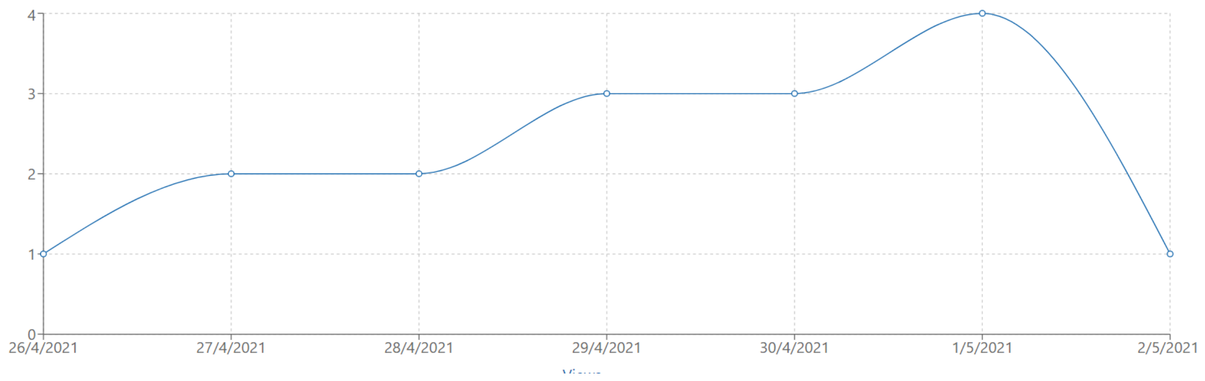
A line chart showcasing all the different length of watch sessions from when the user begins playing the video up until they leave that page.



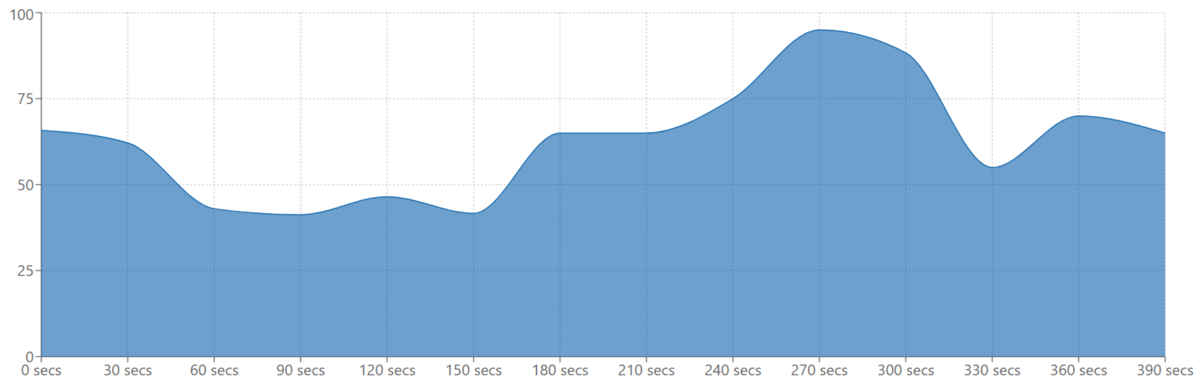
All watch sessions average time of day the video was watched



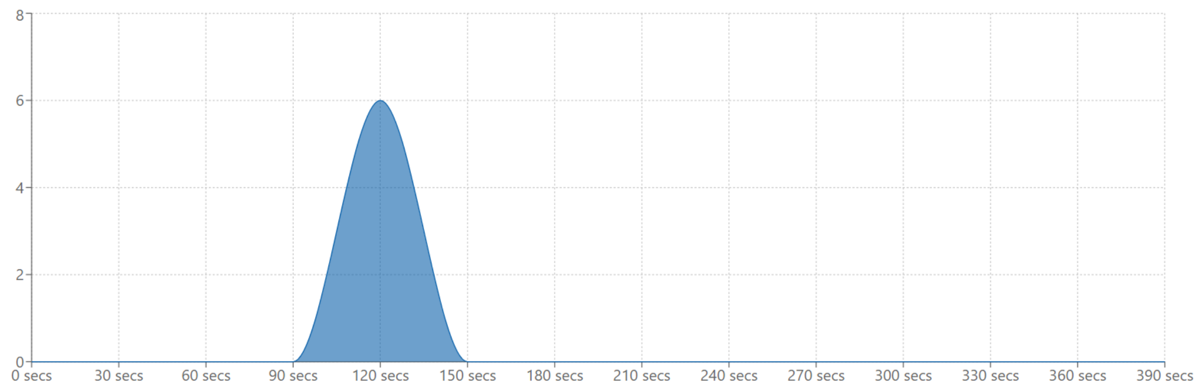
All views encountered within the last seven days of the current date.



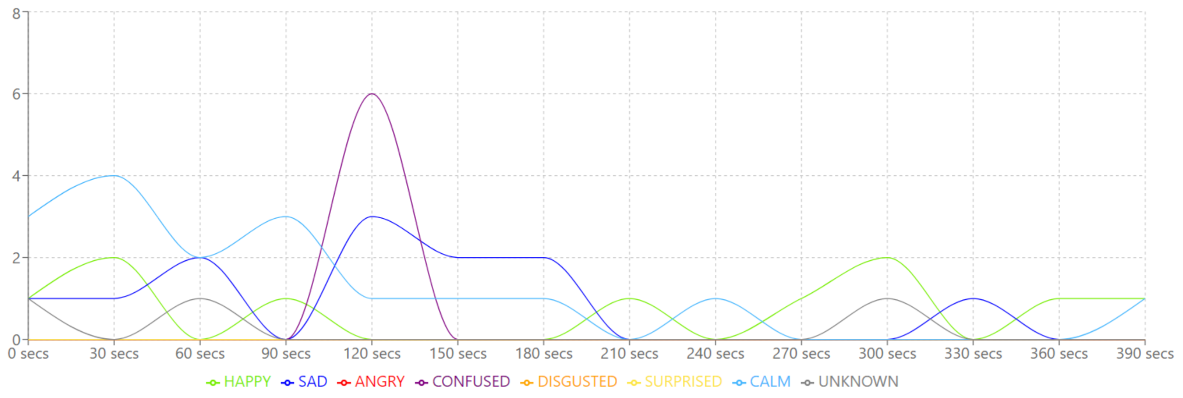
The average concentration at each point of the video.



A graph to show if, at any point, a part of the video caused any suspected confusion among students.

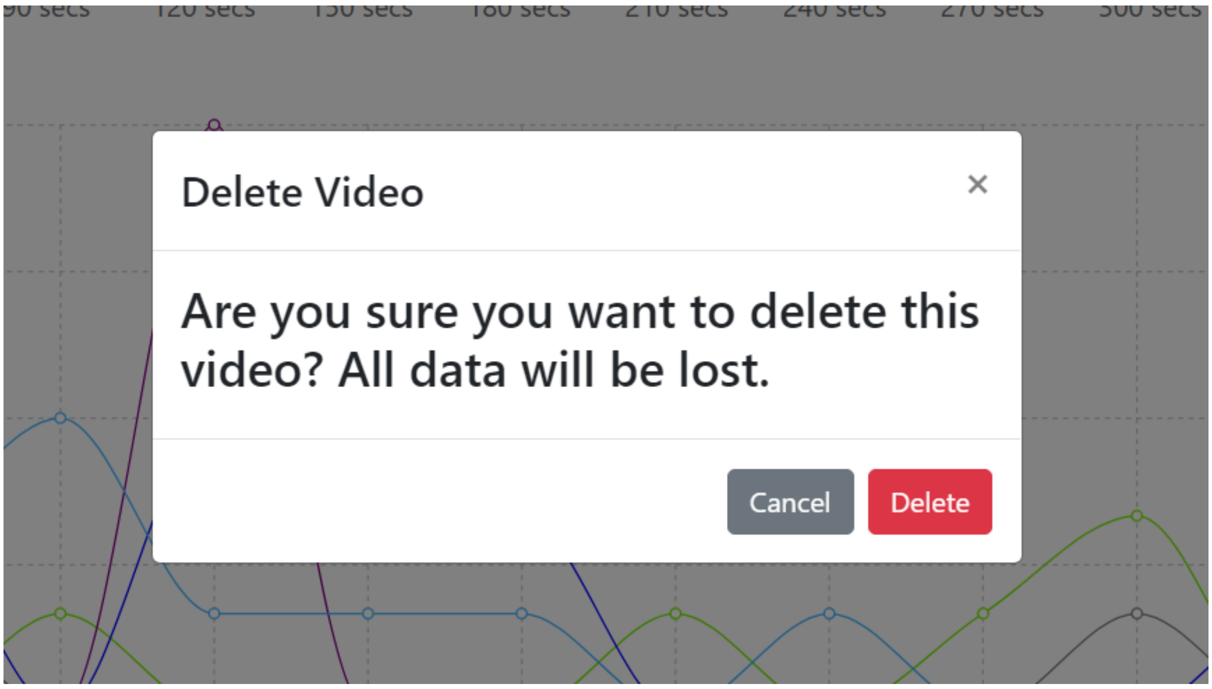


Finally, all emotions as seen at each point throughout the video.



While this data is being retrieved and put into graphs, a loading spinner is shown to inform users that the application is working, although this process completes in less than one second most of the time.

Also, at the very bottom of this page, to ensure no accidental presses, a delete video button allows the lecturer to delete the currently selected video and all of its data. To further ensure no unintentional deletes, we provide a pop up asking for confirmation before the delete is processed.

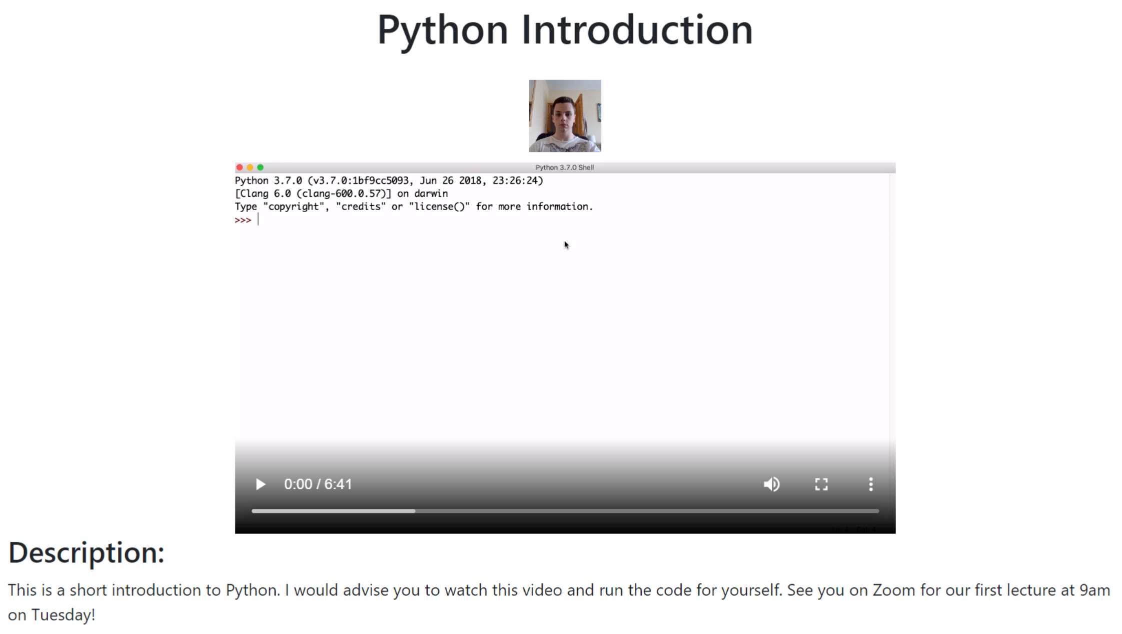


Delete Video ×

Are you sure you want to delete this video? All data will be lost.

5.6 Video Player Page Component

The video player page is one of the main parts of our application from the student's perspective. The video player page looks as follows:



Python Introduction

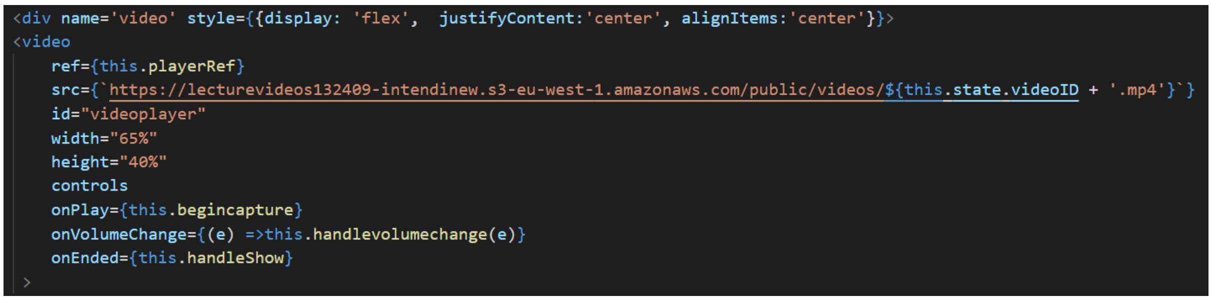
```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> |
```

Description:

This is a short introduction to Python. I would advise you to watch this video and run the code for yourself. See you on Zoom for our first lecture at 9am on Tuesday!

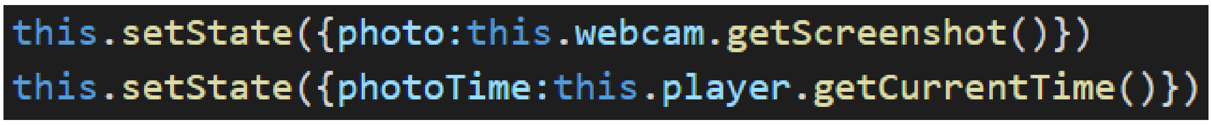
The student can see the video title, webcam display, the video itself and a video description.

The video player code can be seen below:



```
<div name='video' style={{display: 'flex',  justifyContent:'center', alignItems:'center'}}>
<video
    ref={this.playerRef}
    src={`https://lecturevideos132409-intendinew.s3-eu-west-1.amazonaws.com/public/videos/${this.state.videoID + '.mp4'}`}
    id="videoplayer"
    width="65%"
    height="40%"
    controls
    onPlay={this.begincapture}
    onVolumeChange={(e) =>this.handlevolumechange(e)}
    onEnded={this.handleShow}
  >
```

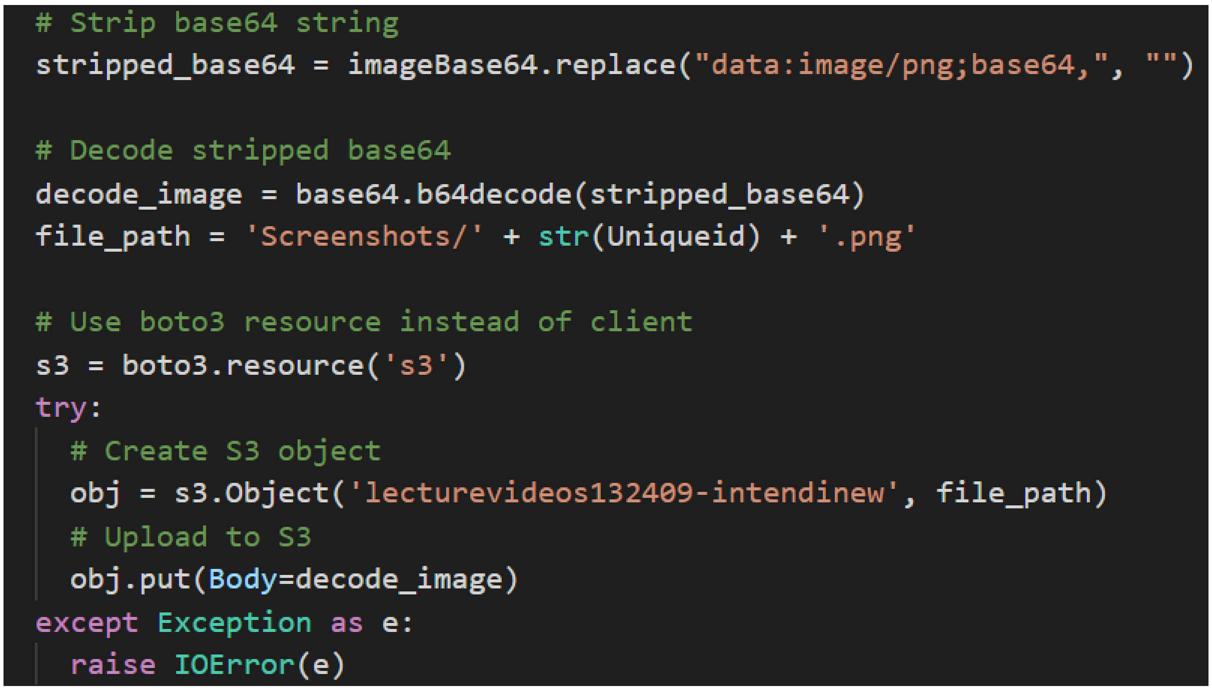
You can see that we are accessing the relevant video using the video's unique ID. We are using HTML5's <video> tag and its onPlay attribute to trigger the capture method when the video begins. The capture method does a number of things. The capture method takes a screenshot when the video starts playing and continues to take a screenshot every ten seconds. When a screenshot is taken, the timestamp is noted. This allows group screenshots and their further analysis by timestamp.



```
this.setState({photo:this.webcam.getScreenshot()})
this.setState({photoTime:this.player.getCurrentTime()})
```

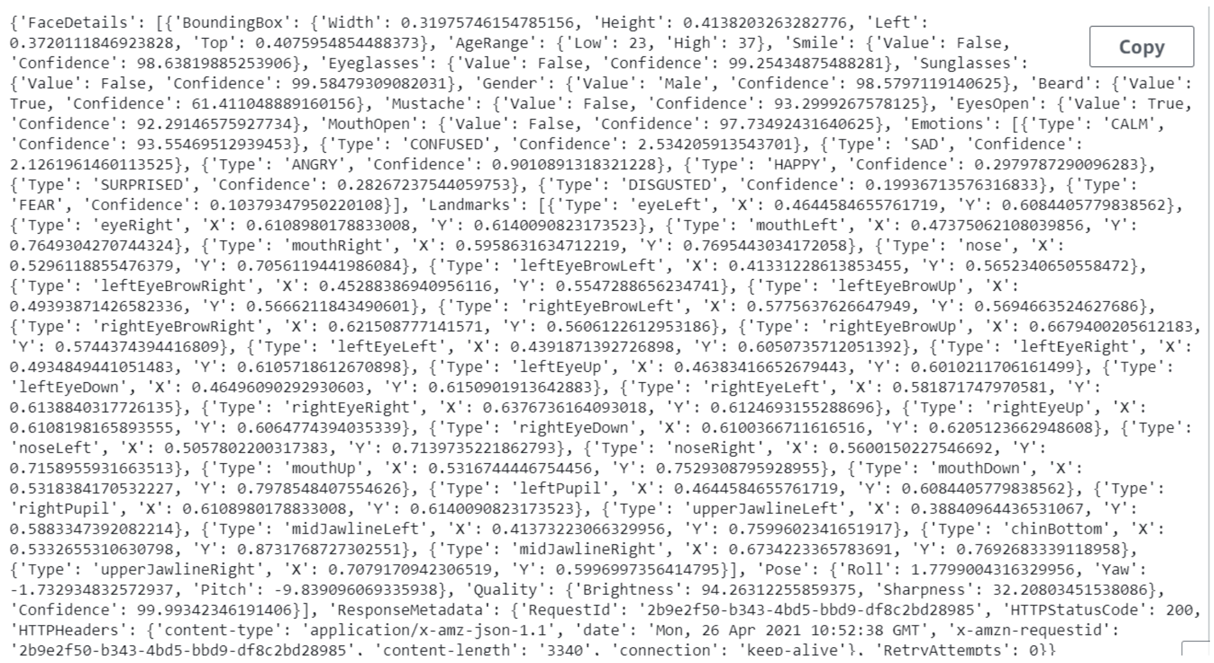
This method then sends that screenshot to our Lambda function as a base64 string along with information such as whether or not the user was tabbed in and whether or not the user had the video on mute.

A JSON object is passed into the Lambda function. Inside the Lambda function, the image base64 string is decoded and uploaded to an S3 bucket as a png file.



```
# Strip base64 string
stripped_base64 = imageBase64.replace("data:image/png;base64,", "")

# Decode stripped base64
decode_image = base64.b64decode(stripped_base64)
file_path = 'Screenshots/' + str(Uniqueid) + '.png'

# Use boto3 resource instead of client
s3 = boto3.resource('s3')
try:
    # Create S3 object
    obj = s3.Object('lecturevideos132409-intendinew', file_path)
    # Upload to S3
    obj.put(Body=decode_image)
except Exception as e:
    raise IOError(e)
```

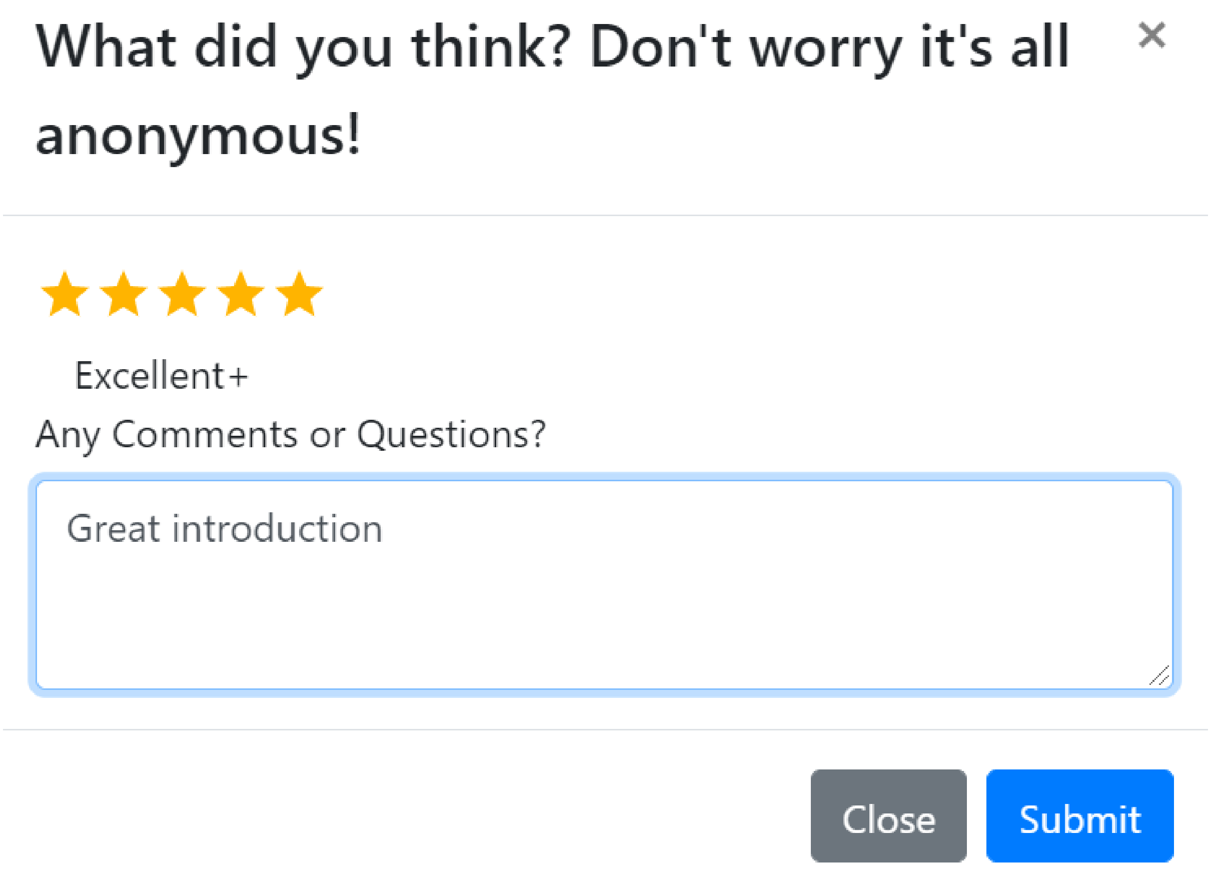
After this, the image is then analysed by AWS Rekognition. AWS Rekognition returns a JSON of results. The JSON looks like this:



```
{'FaceDetails': [{'BoundingBox': {'Width': 0.31975746154785156, 'Height': 0.4138203263282776, 'Left':
0.3720111846923828, 'Top': 0.4075954854488373}, 'AgeRange': {'Low': 23, 'High': 37}, 'Smile': {'Value': False,
'Confidence': 98.63819885253906}, 'Eyeglasses': {'Value': False, 'Confidence': 99.25434875488281}, 'Sunglasses':
{'Value': False, 'Confidence': 99.58479309082031}, 'Gender': {'Value': 'Male', 'Confidence': 98.5797119140625}, 'Beard': {'Value':
True, 'Confidence': 61.411048889160156}, 'Mustache': {'Value': False, 'Confidence': 93.2999267578125}, 'EyesOpen': {'Value': True,
'Confidence': 92.29146575927734}, 'MouthOpen': {'Value': False, 'Confidence': 97.73492431640625}, 'Emotions': [{'Type': 'CALM',
'Confidence': 93.55469512939453}, {'Type': 'CONFUSED', 'Confidence': 2.534205913543701}, {'Type': 'SAD', 'Confidence':
2.1261961460113525}, {'Type': 'ANGRY', 'Confidence': 0.9010891318321228}, {'Type': 'HAPPY', 'Confidence': 0.2979787290096283},
{'Type': 'SURPRISED', 'Confidence': 0.28267237544059753}, {'Type': 'DISGUSTED', 'Confidence': 0.19936713576316833}, {'Type':
'FEAR', 'Confidence': 0.10379347950220108}], 'Landmarks': [{'Type': 'eyeLeft', 'X': 0.4644584655761719, 'Y': 0.6084405779838562},
{'Type': 'eyeRight', 'X': 0.6108980178833008, 'Y': 0.6140090823173523}, {'Type': 'mouthLeft', 'X': 0.47375062108039856, 'Y':
0.7649304270744324}, {'Type': 'mouthRight', 'X': 0.5958631634712219, 'Y': 0.7695443034172058}, {'Type': 'nose', 'X':
0.5296118855476379, 'Y': 0.7056119441986084}, {'Type': 'leftEyeBrowLeft', 'X': 0.41331228613853455, 'Y': 0.5652340650558472},
{'Type': 'leftEyeBrowRight', 'X': 0.45288386940956116, 'Y': 0.5547288656234741}, {'Type': 'leftEyeBrowUp', 'X':
0.49393871426582336, 'Y': 0.5666211843490601}, {'Type': 'rightEyeBrowLeft', 'X': 0.5775637626647949, 'Y': 0.5694663524627686},
{'Type': 'rightEyeBrowRight', 'X': 0.621508777141571, 'Y': 0.5606122612953186}, {'Type': 'rightEyeBrowUp', 'X': 0.6679400205612183,
'Y': 0.5744374394416809}, {'Type': 'leftEyeLeft', 'X': 0.4391871392726898, 'Y': 0.6050735712051392}, {'Type': 'leftEyeRight', 'X':
0.4934849441051483, 'Y': 0.6105718612670898}, {'Type': 'leftEyeUp', 'X': 0.46383416652679443, 'Y': 0.6010211706161499}, {'Type':
'leftEyeDown', 'X': 0.46496090292930603, 'Y': 0.6150901913642883}, {'Type': 'rightEyeLeft', 'X': 0.581871747970581, 'Y':
0.6138840317726135}, {'Type': 'rightEyeRight', 'X': 0.6376736164093018, 'Y': 0.6124693155288696}, {'Type': 'rightEyeUp', 'X':
0.6108198165893555, 'Y': 0.6064774394035339}, {'Type': 'rightEyeDown', 'X': 0.6100366711616516, 'Y': 0.6205123662948608}, {'Type':
'noseLeft', 'X': 0.5057802200317383, 'Y': 0.7139735221862793}, {'Type': 'noseRight', 'X': 0.5600150227546692, 'Y':
0.7158955931663513}, {'Type': 'mouthUp', 'X': 0.5316744446754456, 'Y': 0.7529308795928955}, {'Type': 'mouthDown', 'X':
0.5318384170532227, 'Y': 0.7978548407554626}, {'Type': 'leftPupil', 'X': 0.4644584655761719, 'Y': 0.6084405779838562}, {'Type':
'rightPupil', 'X': 0.6108980178833008, 'Y': 0.6140090823173523}, {'Type': 'upperJawlineLeft', 'X': 0.38840964436531067, 'Y':
0.5883347392082214}, {'Type': 'midJawlineLeft', 'X': 0.41373223066329956, 'Y': 0.7599602341651917}, {'Type': 'chinBottom', 'X':
0.5332655310630798, 'Y': 0.8731768727302551}, {'Type': 'midJawlineRight', 'X': 0.6734223365783691, 'Y': 0.7692683339118958},
{'Type': 'upperJawlineRight', 'X': 0.7079170942306519, 'Y': 0.5996997356414795}], 'Pose': {'Roll': 1.7799004316329956, 'Yaw':
-1.732934832572937, 'Pitch': -9.839096069335938}, 'Quality': {'Brightness': 94.26312255859375, 'Sharpness': 32.2080345153808б},
'Confidence': 99.99342346191406}], 'ResponseMetadata': {'RequestId': '2b9e2f50-b343-4bd5-bbd9-df8c2bd28985', 'HTTPStatusCode': 200,
'HTTPHeaders': {'content-type': 'application/x-amz-json-1.1', 'date': 'Mon, 26 Apr 2021 10:52:38 GMT', 'x-amzn-requestid':
'2b9e2f50-b343-4bd5-bbd9-df8c2bd28985', 'content-length': '3340', 'connection': 'keep-alive'}, 'RetryAttempts': 0}}
```

From this JSON, we can extract data such as emotions and facial landmarks. The data above provides many useful pieces of data. We can see data such as age range and gender but felt this was generally not very useful. Useful data was mainly from the actual face position and angles of the face such as pitch and yaw. These are measured on both the X and Y axis and through testing we were able to determine quite specific numbers that we decided would indicate a loss of concentration. We used this information in the implementation of our concentration algorithm which we will talk about further on.

After we are done analysing the image, it is immediately deleted. This process continues every ten seconds until the video has ended. When the video ends, a feedback form will pop up.


What did you think? Don't worry it's all anonymous!

★★★★★

Excellent+

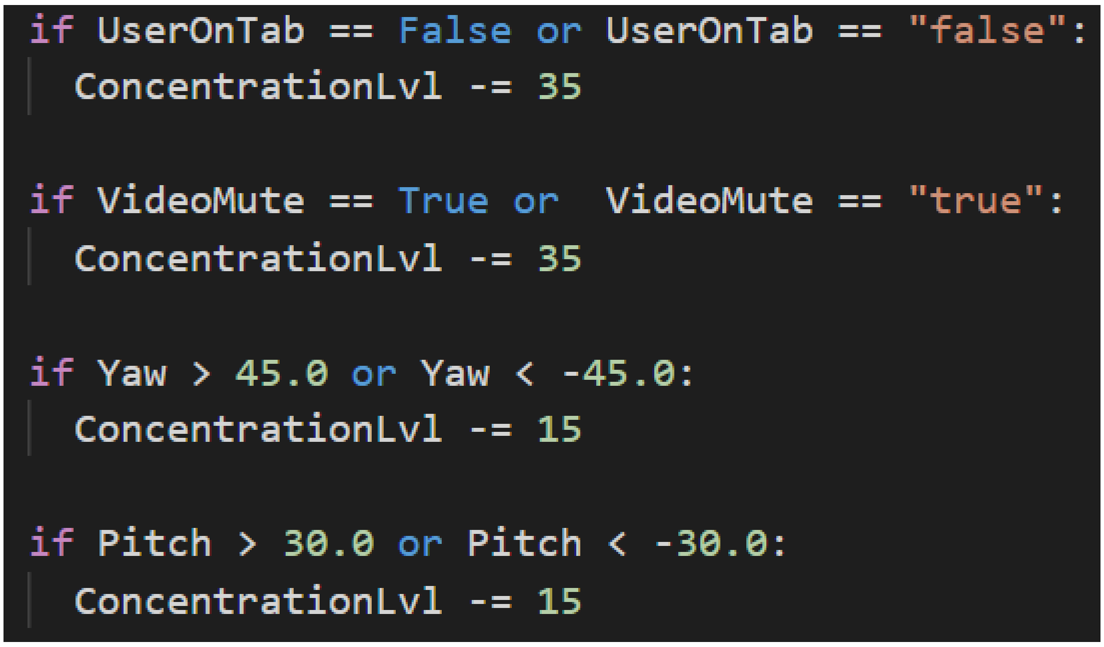
Any Comments or Questions?

Great introduction

The students have the option to rate the video out of five stars and to submit some feedback in text form. This allows lecturers to get instant feedback on their videos. Our feedback form uses another Lambda function that is triggered when the “Submit” button is pressed. This Lambda function adds the rating and comment to our VideoDetails database table.

5.6.1 Concentration Algorithm

We also have our concentration algorithm in the facial analysis Lambda. For this we decided on multiple key findings extracted from the facial analysis which we felt would indicate a certain level of concentration. This includes if the user is on the tab, if the video is muted and then facial directions such as if the user is looking upwards or downwards and if they are strongly looking to the left or right. With these we felt we could gauge a certain level of concentration.

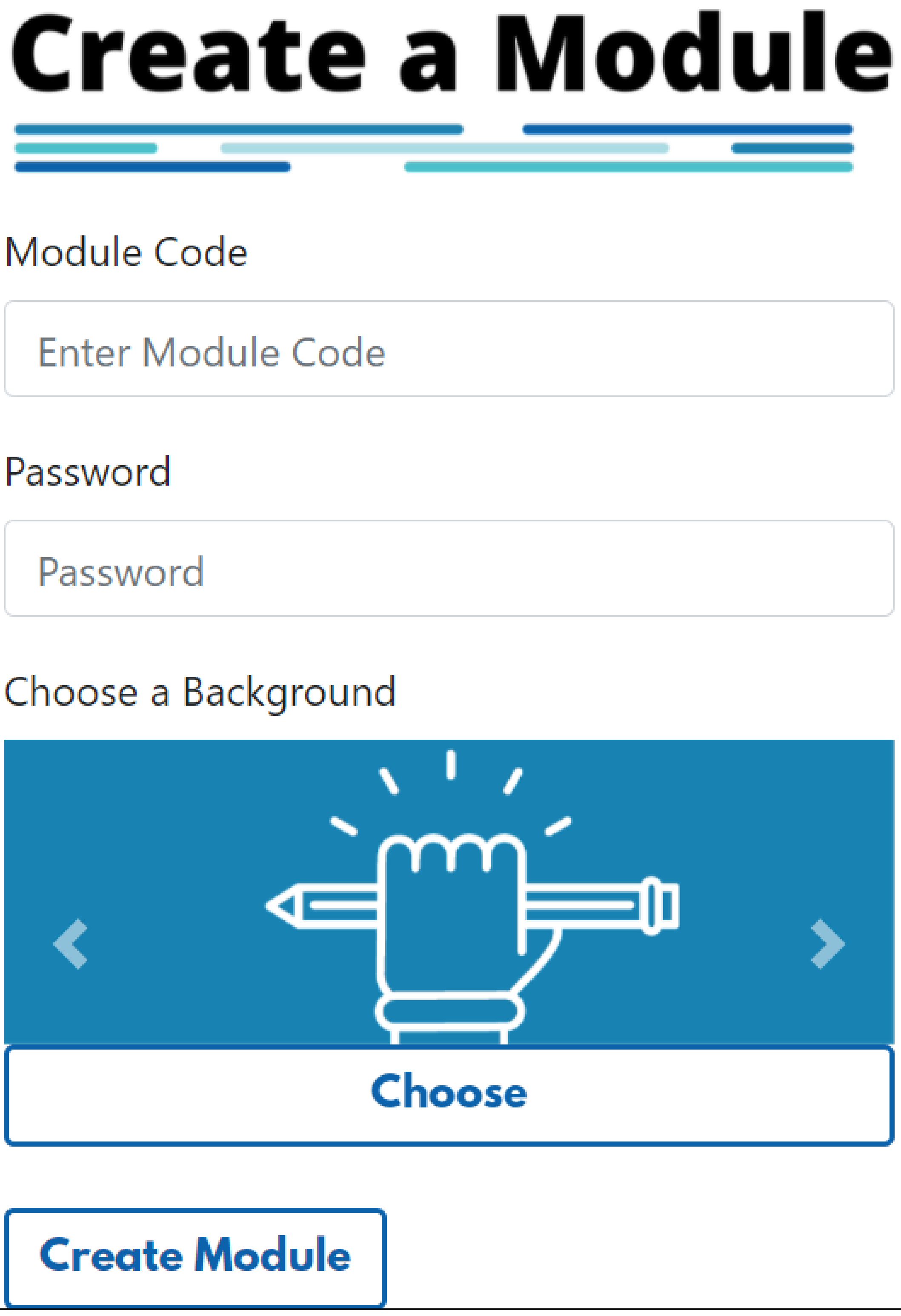


```
if UserOnTab == False or UserOnTab == "false":
    ConcentrationLvl -= 35

if VideoMute == True or  VideoMute == "true":
    ConcentrationLvl -= 35

if Yaw > 45.0 or Yaw < -45.0:
    ConcentrationLvl -= 15

if Pitch > 30.0 or Pitch < -30.0:
    ConcentrationLvl -= 15
```

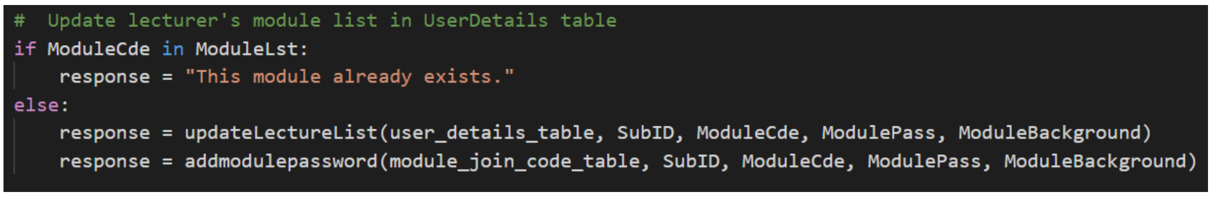
Concentration starts at 100, but we subtract points from the student's concentration level if specific criteria are met. For example, if the user is not currently in the video tab, their concentration level is subtracted by 35. These levels of decreasing the concentration level resulted from both of us discussing what we felt contributes to realising if a student is not concentrating. For example, the user on tab we felt would highly indicate the user is not paying attention and instead looking at the internet while the video plays in the background. Whilst if the student is looking very far left or right we decrease the amount but we felt as it is images being analysed and not video we also take into account they could have been only looking for a brief minute or two. This algorithm is based on our own experience and assumptions and is not therefore entirely accurate to a professional level. For developing this project further we would aim to garner professional feedback from people within relevant fields.

5.7 Create Module Component

The create module page is used by lecturers to create modules. The lecturer enters a module code, a module password and chooses a background for the module.



This information is then stored in the ModuleDetails table. When the lecturer clicks the "Create Module" button, a Lambda function is triggered. Information such as the module code, module password and the selected background image is passed into the Lambda module in JSON format.

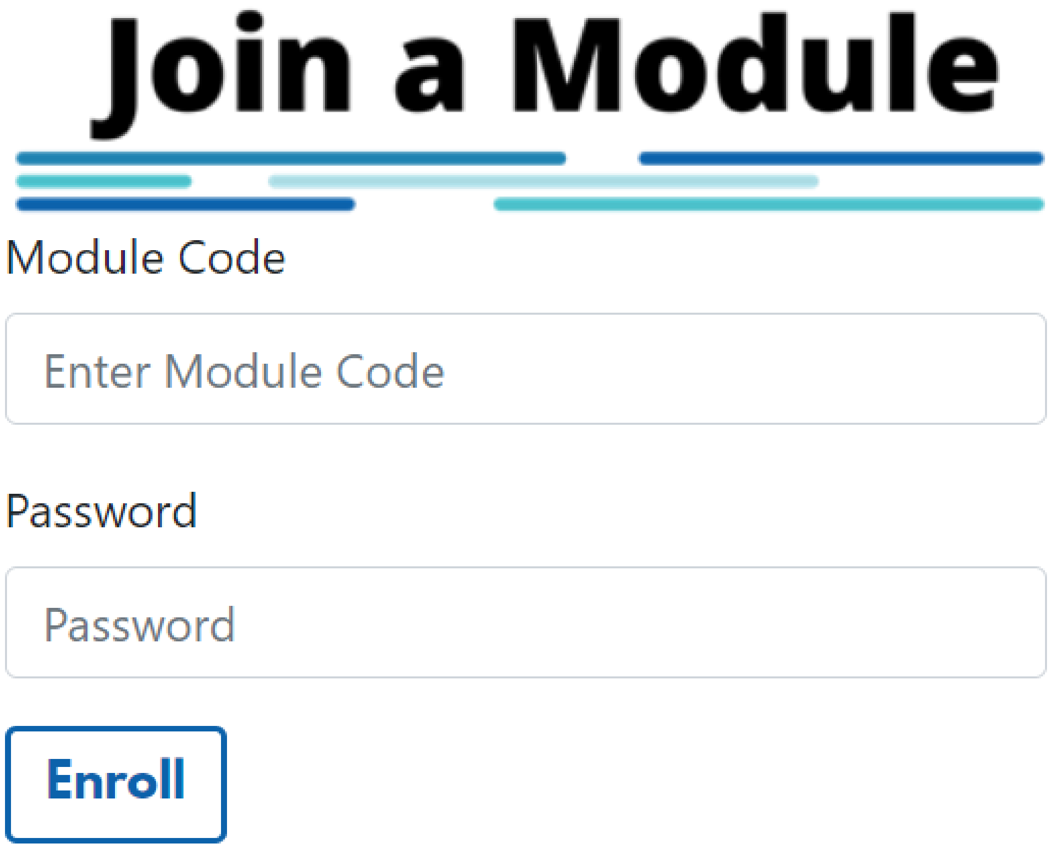


```
#  Update lecturer's module list in UserDetails table
if ModuleCde in ModuleLst:
    response = "This module already exists."
else:
    response = updateLectureList(user_details_table, SubID, ModuleCde, ModulePass, ModuleBackground)
    response = addmodulepassword(module_join_code_table, SubID, ModuleCde, ModulePass, ModuleBackground)
```

In the above screenshot from the respective Lambda function, you can see that we check to see if the module already exists; if it does, an error message is returned. Otherwise, the module is created by adding the module to the lecturer's list of modules and adding the module to our module database table.

5.8 Join Module Component

The join module page is used by students to join modules. A student enters the module code and module password and presses the "Enroll" button.

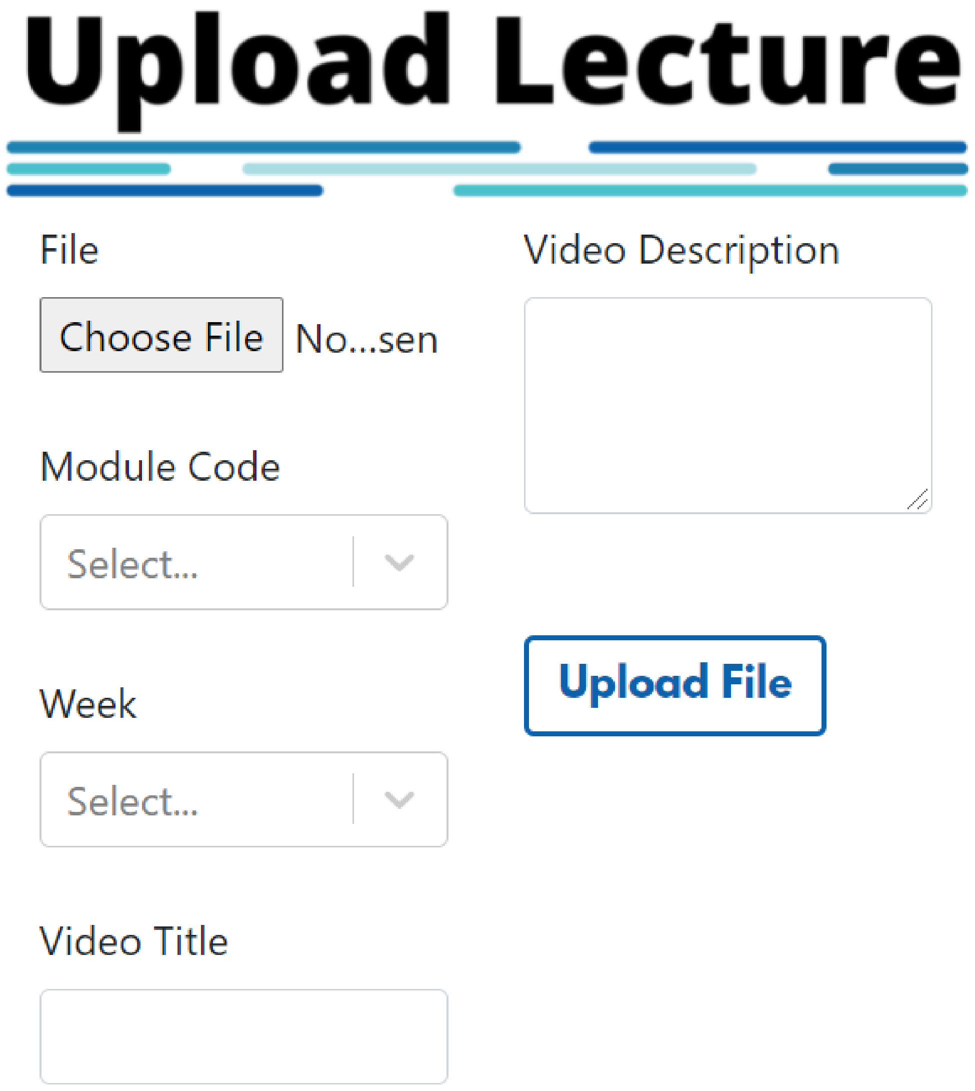


This triggers a Lambda function and the entered module code and password is passed in the Lambda function as a JSON object. The Lambda function checks if the student is already enrolled in the said module, if they are, then an error message is displayed. Otherwise, if the module exists and the module password is correct, the student joins the module. The module is added to the student's list of enrolled modules in the UserDetails database table and the student is added to the module's list of enrolled students in the ModuleDetails database table.

5.9 Upload Lecture Component

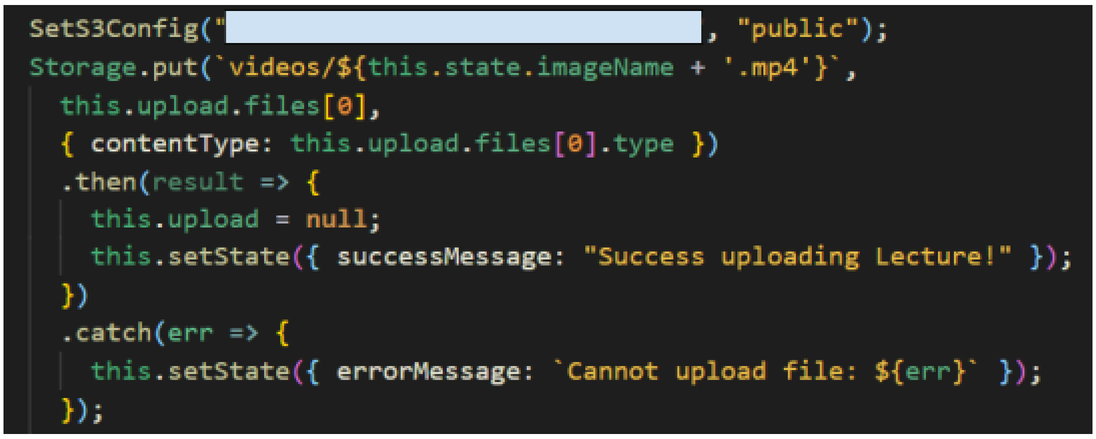
The upload page of Intendi is only available through a lecturer's login. Once the component has mounted, we first fetch the users modules through an API POST request to the getUserModules Lambda. This is used to populate the dropdown module selection within the upload form to avoid any errors from the lecturer accidentally typing in the wrong module code.

The form is then displayed, which involves a file picker that only accepts .mp4's at the moment. This will bring up the client file explorer to pick a video. Once selected, the data will be preserved on the client-side until uploaded. We also implemented a dropdown week selection and text boxes for both the video title and a multiline textbox for the description of the video.



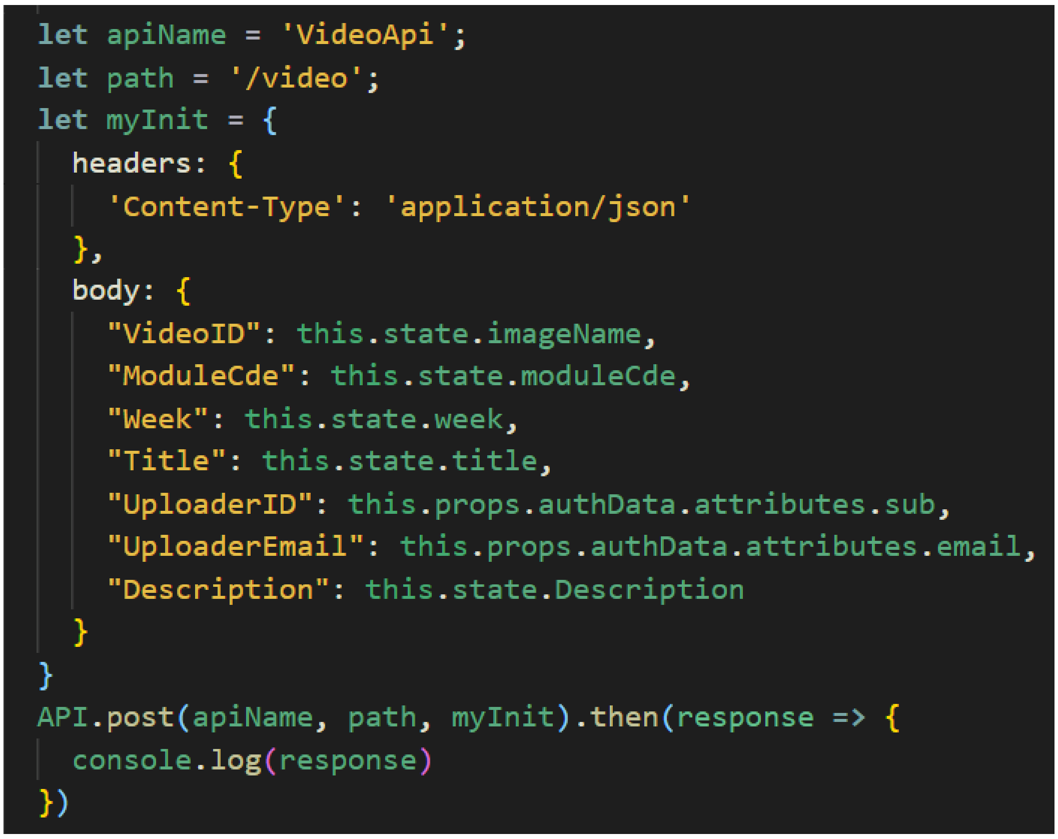
Once these are all filled in, the user may proceed to press the upload button. This calls the UploadVideo function, which begins the upload process.

Firstly we directly upload the .mp4 video straight to the S3 bucket rather than attempting to pass the video through to a Lambda as this can be quite difficult. Using the direct upload to S3 approach ensures efficient and quick results.



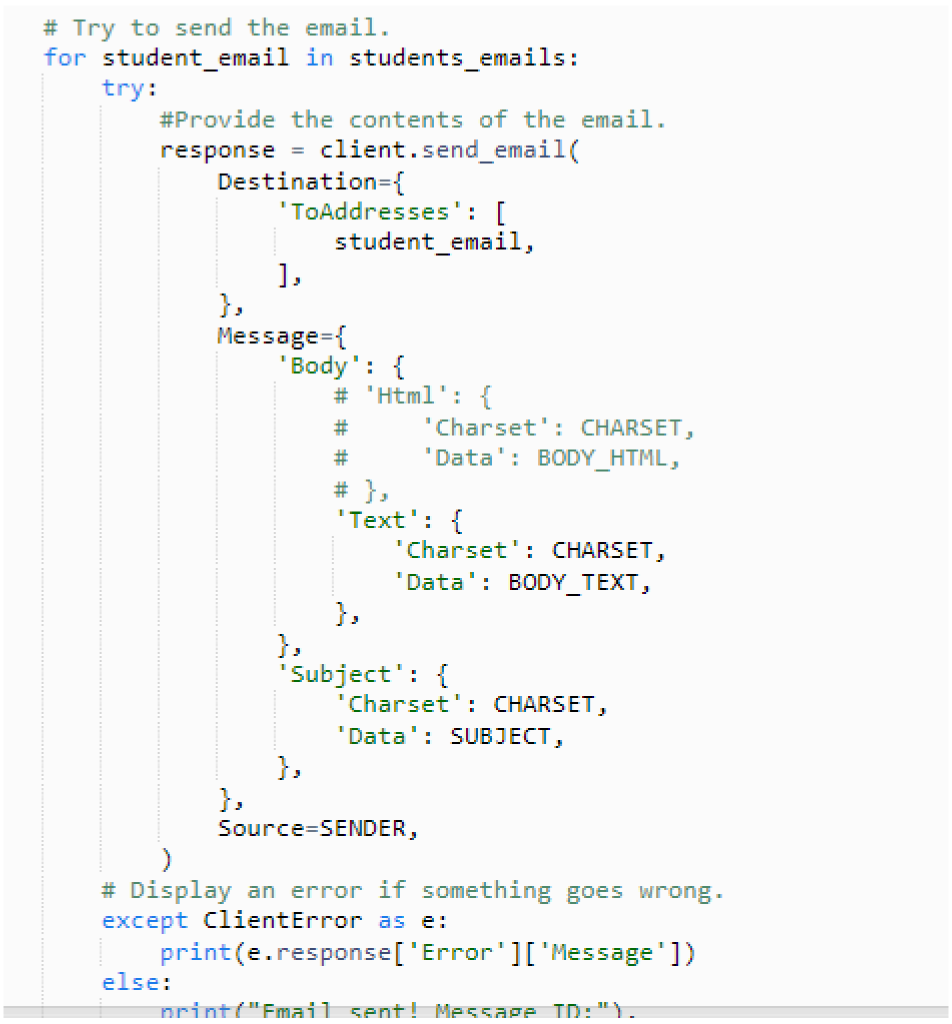
```
SetS3Config("[redacted], "public");
Storage.put(`videos/${this.state.imageName + '.mp4'}`,
  this.upload.files[0],
  { contentType: this.upload.files[0].type })
  .then(result => {
    this.upload = null;
    this.setState({ successMessage: "Success uploading Lecture!" });
  })
  .catch(err => {
    this.setState({ errorMessage: `Cannot upload file: ${err}` });
  });
```

Once that has completed an API with all the video information is called to invoke a Lambda function to upload to the VideoDetails DynamoDB Table. This involves the video ID, module code, the week it was uploaded to, video title, the unique ID of the user who uploaded it and their email (using Amplify/Cognito credential tokens). This table aids in the retrieval of the correct video from the S3 bucket as they are uploaded under unique IDs.



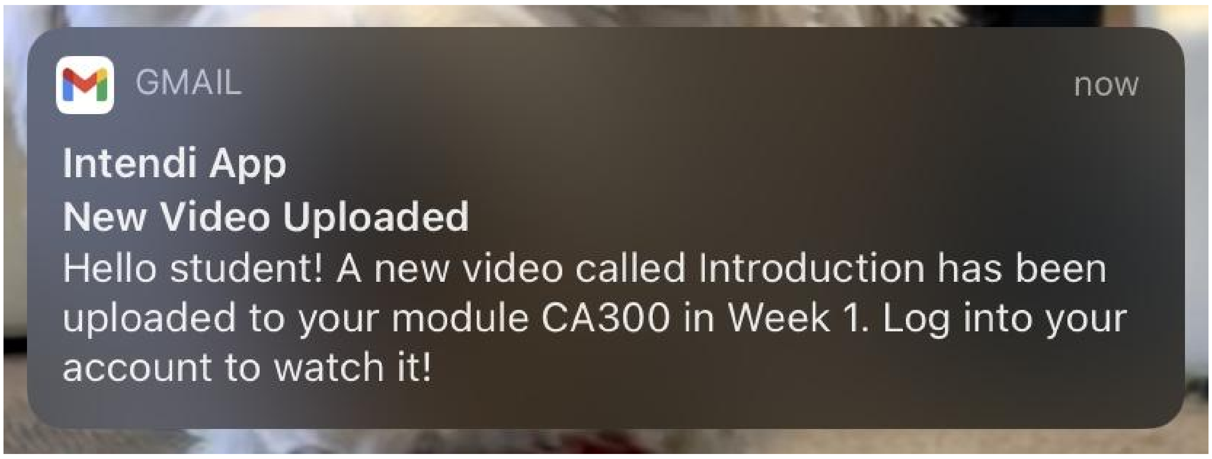
```
let apiName = 'VideoApi';
let path = '/video';
let myInit = {
  headers: {
    'Content-Type': 'application/json'
  },
  body: {
    "VideoID": this.state.imageName,
    "ModuleCde": this.state.moduleCde,
    "Week": this.state.week,
    "Title": this.state.title,
    "UploaderID": this.props.authData.attributes.sub,
    "UploaderEmail": this.props.authData.attributes.email,
    "Description": this.state.Description
  }
}
API.post(apiName, path, myInit).then(response => {
  console.log(response)
})
```

Finally, once this and the video has been uploaded, we loop through all the students who are enrolled in the module and using AWS SES (Simple Email Service), we send them an automatic email to let them know that a new video has been uploaded to one of their modules.



```
# Try to send the email.
for student_email in students_emails:
    try:
        #Provide the contents of the email.
        response = client.send_email(
            Destination={
                'ToAddresses': [
                    student_email,
                ],
            },
            Message={
                'Body': {
                    # 'Html': {
                    #     'Charset': CHARSET,
                    #     'Data': BODY_HTML,
                    # },
                    'Text': {
                        'Charset': CHARSET,
                        'Data': BODY_TEXT,
                    },
                },
                'Subject': {
                    'Charset': CHARSET,
                    'Data': SUBJECT,
                },
            },
            Source=SENDER,
        )
    # Display an error if something goes wrong.
    except ClientError as e:
        print(e.response['Error']['Message'])
    else:
        print("Email sent! Message ID:"),
```

As we can see here, we have already retrieved all enrolled students emails from the module table and now loop through them and perform the necessary AWS SES boto commands. After this code is executed, each student should receive an email similar to the one seen in the below screenshot:


GMAIL now

Intendi App
New Video Uploaded
Hello student! A new video called Introduction has been uploaded to your module CA300 in Week 1. Log into your account to watch it!

Once this is all complete, a success alert will appear on the front-end to inform the lecturer everything has completed successfully.

6. Problems and Solutions

Policies or character size limit for an IAM role or user Exceeded:

This problem arose towards the end of the project. After the 10th lambda was added to the system through the Amplify CLI and then once we attempted to push the changes to the cloud, there was an error returning stating how the authenticated user role policy character limit has exceeded 10,240 characters. This was a problem neither of us had seen before. Upon research of the error, we found the following article provided by AWS:

https://aws.amazon.com/premiumsupport/knowledge-center/iam-increase-policy-size/

Unfortunately, this was no help with solving the issue and we still needed to create another two API Paths with Lambdas, but we could not create any due to this error. Eventually, through some extensive online research, we found that the error was due to lazy policy creation performed by the Amplify CLI. We saw that for every API Path within the API Cloudformation template of our project, it was writing every available REST API command possible, e.g. POST, GET, FETCH and DELETE. This resulted in the exceeding of the character limit for the authenticated user policy. We, therefore, came up with the idea of simply merging all the requests for each path into one by simply declaring “ * ” for each one as this accounts for all requests. With this change, it cut the character limit by more than half and we were able to create and push the newly created API paths and their respective Lambdas.

AWS Rekognition unable to work with base64 images:

This problem was encountered whilst making the Lambda which sent the screenshots to be processed by the AWS Rekognition. Our original idea was to send the screenshots as a base64 encoded image to the Lambda and then perform the necessary Rekognition analysis on it. When we tried this, we could not seem to get either the encoded or decoded image to work correctly with the AWS Rekognition as it kept reporting an “incorrect image type”. We, therefore, decided to go down the route of passing in the image as base64, decoding it and then proceeding to upload the image to a secure S3 bucket. We then asked AWS Rekognition to retrieve the image and perform the facial analysis on it, which worked perfectly. We seemed to conclude that between how the react-webcam library converted screenshots to base64 and then how we decoded the image, AWS Rekognition was unable to verify what type of image was being passed through, e.g. .png or .jpeg. This is how we came to a conclusion to directly decode and attach the file extension to the end of the screenshot filename and upload it to the S3 bucket.

React Player unable to track volume change

We were initially using react-player for the student video player page, but unfortunately, after a few weeks went by, the event listener we had to check for any volume changes stopped working, resulting in inaccurate concentration level scores. We realised the problem was first related to how react-player does not have any built-in callback function for volume changes. We then attempted to look for the underlying HTML5 video tag that the react-player may have, but unfortunately, this kept resulting in “null” errors. We finally decided to revert to the basic original HTML5 video player and add some styling to it. This allowed us to use common HTML5 attributes such as onVolumeChange. As a result, we were able to check if the user has muted the video back working, which enabled us to perform our concentration algorithm to its original standard.

7. Testing

Testing can be seen within our testing documentation found within the same repo of this document.

8 Future Work

In terms of future work, if we were to move forward with the project working alongside a university, we would look into attempting to perform live raw video input facial analysis rather than image analysis. This, we feel, with the necessary funding could lead to even more in-depth student concentration and general feedback data.

We feel this idea/project would be very beneficial to the likes of MOOCS (Massive open online course) as they continue to grow in popularity. These are growing largely all over the world and more and more colleges are looking into providing completely online courses.

Another approach we felt we could begin to develop towards was more from a business side of things where a company may use our software in getting general emotional consensus and engagement of a new advert they were hoping to release. They could use our technology with random test audiences which would allow for the audience not being present in person and also become anonymous, which the test users may like.

Another aspect we would like to look into in the future would be to refine our concentration algorithm further and perhaps introduce some element of machine learning to create a much more accurate representation of the student concentration level. We would further add to this by getting in contact with various experts in the field of education, facial recognition and, in particular, psychologists.

9 References

1. Zaletelj, J. and Košir, A., 2017. Predicting students’ attention in the classroom from Kinect facial and body features. *EURASIP Journal on Image and Video Processing*, 2017(1).