CS261: Software Engineering Project Final Report - Group 23

Harry Fallows, Simon Heap, Bhavik Makwana, Pavnish Mohan, Antonia Newey

Contents

1 Introduction

The purpose of this document is to provide a full description of the design and creation of the deliverable. It intends to outline the features of the deliverable and the design choices taken in its production. It should evaluate how well the system meets the client's needs and the improvements that could be made to the system were it to continue development. It will outline the steps taken by the team to work effectively together to create the product and come to a conclusion regarding the success of the project as a whole.

2 Research

2.1 Frontend

For the task of building a UI, there are several types of tools that can be used, including different CSS frameworks and JavaScript libraries.

For CSS frameworks, the top contenders were Bulma, Bootstrap, Foundation, and Pure. A number of criteria were taken into account when making the choice:

- Popularity: a popular framework with many users means access to more real-world examples and solutions to common issues. It also invites the development of third-party extensions and is likely to have better upkeep.
- Documentation: good documentation makes a framework far easier to learn and use, allowing pages to be built quickly and easily.
- Development activity: as web technology changes, a good framework must evolve to stay functional on as many devices and browsers as possible.
- Complexity: if a framework has too steep of a learning curve, time will be wasted trying to understand it.
- Responsiveness: this determines whether the framework is suitable for building a web-app on different size screens, which is an increasingly important factor of website design.
- Appearance: the frontend UI is all most users will actually see; it should, therefore, be as attractive as possible.

In the end, the Bulma CSS framework [1] was chosen. It meets the criteria as follows:

- Bulma is the newest framework amongst those mentioned, but its popularity is rising and it has been starred on GitHub more than all of the others except Bootstrap.
- Bulma has reasonably comprehensive, and very readable, documentation. It details overviews of general concepts and ideas as well as individual components, e.g. a 'Layout' section as well as 'Column' etc.
- The Bulma GitHub repository is updated regularly, so constant improvements are being made.
- Bulma is very simple and easy to learn, it has well-defined elements that can be combined easily. There are a number of global variables that can be customised to tailor it to a user's requirements; these global variables and class names also have natural language names making it even easier to decide when to use what (e.g. has-text-centered is a class that will center the text in an element).
- Bulma is based on the flex-box model and as such scales and aligns content automatically. There are a number of different layout methods offered, such as levels and columns as well as the standard grid, and all can change with viewport size.
- Bulma CSS is clean and simple. It is customisable but even the default stylesheet is visually appealing. The overall look is modern and professional.

jQuery was the only JavaScript library seriously considered for a number of reasons; it is lightweight but still very powerful, has a wide variety of plugins available, has good documentation, and because of its chaining capabilities.

2.2 Backend

API

When researching the API options two major technologies were considered, Flask and Django. These are both Representational State Transfer (REST) [2] frameworks, which take advantage of standard HTTP requests in a web API. In the end, Flask was chosen for its lightweight, simplistic, and minimalist design [3]. For the development of the Flask app, the Flask docs were regularly referenced [4]. The docs contain a tutorial for developing a simple blog website, these provided a very useful example for implementing the key features of a Flask app.

Database

In the research done regarding the database system the main decision came in using an SQL system such as SQLite or MySQL or a NoSQL system of which one of the most popular and widely documented was MongoDB. MongoDB was chosen as the ideal system due to the data set provided being given as a JSON file, a file type easily processed and stored in MongoDB's document-oriented system. MongoDB is also widely used and well supported by the other systems that the system would be using (e.g. Flask has its own pymongo module). Upon deciding to use the MongoDB system, research continued in all database based design choices by referencing the thorough MongoDB documentation [5].

Machine Learning

The majority of research for Machine Learning was carried out during the design and planning stage. The general approach remains the same in comparing each CV to a model CV, using cosine similarity to calculate how similar they are to label them for supervised learning. The method taken to do this is similar to an unsupervised learning approach called clustering[6], but is not a standard clustering technique.
Other Machine Learning libraries such as sklearn were explored. This is a higher level API than TensorFlow, abstracting away some more of the complexities. However, sklearn is CPU bound whereas TensorFlow can be written to make use of GPUs which allows for faster training.
Further research was done into preprocessing the data to get it ready for machine learning. Sklearn provides an inbuilt function to calculate the cosine similarity of two vectors. Converting the data into pandas as is done with standard machine learning programs proved difficult using JSON; there was no functionality in any of the APIs researched which dealt with this, meaning either bespoke functions to parse JSON would have to be written or the JSON would require some preprocessing for nested values such as languages. This would then get converted into a csv to be loaded into a pandas frame. Online learning also appeared to be more of a challenge than anticipated. Sklearn provides a partial_fit for some models to learn on top of their previous state, this is widely used in e-commerce and social networking sites where models must be kept up-to-date. This was similar to what is required of the system being developed [7]. The alternative to online learning would be retraining periodically to compensate for this instead, which would make updating the system much slower.

3 Implementation

3.1 Frontend

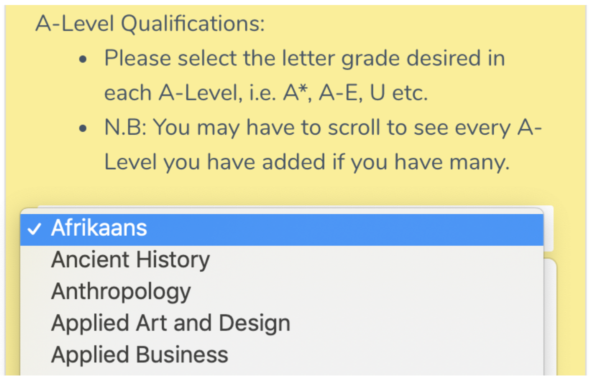
User Interface

Installing Bulma with nodeJS allowed the use of the SASS pre-processor [8] to build the stylesheet dynamically, Bulma's recommended method for customisation. Modules required could be imported in the .scss file from the Bulma node module and the relevant classes would be added into the CSS.

The web app was built using Flask[4], a python micro-framework, which requires a certain structure that doesn't lend itself to node and its modules. Enough work had been done that it was unlikely any modules for new elements/components would need to be imported and so any further styling could be done by adding more, or more specific, classes or selecting individual elements by id to change. Initially, a small separate build that still used node was kept just in case. Overall, it was clear that much more would be gained by eliminating node than would be lost.

Flask builds the web app and ties together the UI, the API, and the database, with the help of the Jinja2 templating engine. One of the most useful things about this method is that Jinja2's templates are extensible. For example, a 'base.html' can be made which contains elements that should appear on every page (like the navbar, footer, and the inclusion of the CSS file in the header) along with some blank content blocks. It is then possible to build upon this template to create new pages by specifying what content should go in these blocks, this minimises code repetition. Jinja2 also allows parameters to be easily passed from the backend to the frontend, which is useful for building pages dynamically. An example of this is: when the job listing page is generated, the database is queried for active jobs. The job listing template can then iterate over this list and add content to the page in the same structure for each job.

Bulma is a pure CSS framework, and therefore more complicated functionality could only be achieved by adding JavaScript content. The web app is built partly with pure JavaScript and partly with the jQuery JavaScript framework [9]. The pages with the most JavaScript are the main forms (filling in your details and creating a job) and the initial test page. Some inputs were simple text boxes such as names and descriptions, some were basic selects, but other fields were much more complex. For skills, hobbies, A-Levels, and programming languages, each candidate or client can choose a variable from a number of options (e.g. Figure 1) with an associated grade or expertise variable (e.g. Figure 2).


A-Level Qualifications:

- Please select the letter grade desired in each A-Level, i.e. A*, A-E, U etc.
- N.B: You may have to scroll to see every A-Level you have added if you have many.

Figure 1: Selecting an A-Level to Favour

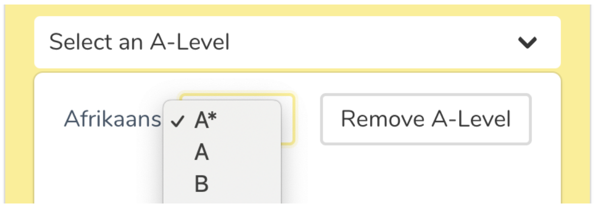
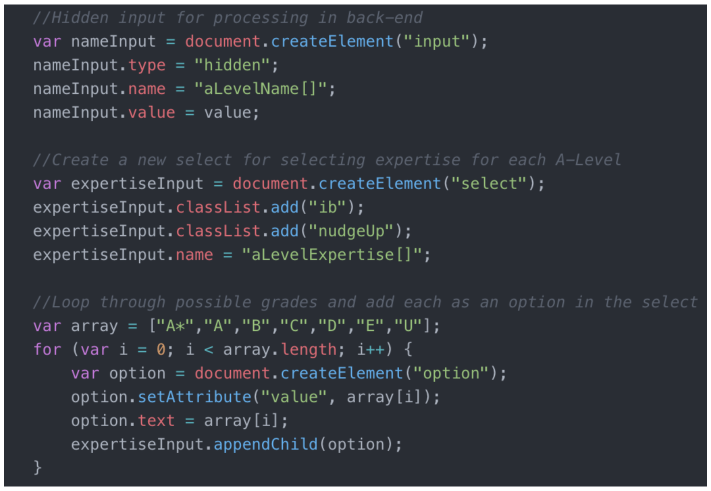


Figure 2: Selecting a Grade to Favour in an A-Level

There were so many options in all of these different categories that it became necessary to add them dynamically (as seen in Figure 3).


```
//Hidden input for processing in back-end
var nameInput = document.createElement("input");
nameInput.type = "hidden";
nameInput.name = "aLevelName[]";
nameInput.value = value;

//Create a new select for selecting expertise for each A-Level
var expertiseInput = document.createElement("select");
expertiseInput.classList.add("ib");
expertiseInput.classList.add("nudgeUp");
expertiseInput.name = "aLevelExpertise[]";

//Loop through possible grades and add each as an option in the select
var array = ["A*","A","B","C","D","E","U"];
for (var i = 0; i < array.length; i++) {
    var option = document.createElement("option");
    option.setAttribute("value", array[i]);
    option.text = array[i];
    expertiseInput.appendChild(option);
}
```

Figure 3: Dynamically Adding an A-Level

It can be seen from Figure 3 that there is a hidden field called aLevelName[] alongside the select. This allows the form to POST in the correct format; multiple fields with the same name will be collected together and can then be processed into Python dictionaries. Python dictionaries match the format of JSON objects which allows the new data to be stored in the same fashion as the data received from the client.

Accessibility

It was stated in the requirements analysis document for this project that the web app would aim to meet WCAG2.0 standards to Level A. This was achieved quite simply, as many of the guidelines discussed content that the web app does not include, e.g. there is no audio or video or flashing content to consider. The remaining guidelines were met in the following ways. They appear in the format **Guideline** - How it Was Met:

- **All non-text content that is presented to the user has a text alternative that serves the equivalent purpose.** - Every image and icon used in the web app has alternative text.
- **Information, structure, and relationships conveyed through presentation can be programmatically determined or are available in text**. - Information is almost always available in text form and related items will have related names, e.g the select for choosing a language has an id 'language' and it adds content to a container with the id 'languageContainer'.
- **When the sequence in which content is presented affects its meaning, a correct reading sequence can be programmatically determined.** - Sequence never affects content's meaning.
- **Instructions provided for understanding and operating content do not rely solely on sensory characteristics of components such as shape, size, visual location, orientation, or sound.** - All instructions are given in text.
- **Colour is not used as the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element.** - Almost all information is also conveyed textually. The only

exception is the results of the tests appearing in a radar chart, but they are given in text as well.

- **Web pages have titles that describe topic or purpose.** - Page titles are descriptive, e.g. on the page displaying your details the title is 'Your Details'.
- **If a web page can be navigated sequentially and the navigation sequences affect meaning or operation, focusable components receive focus in an order that preserves meaning and operability.** - Only one page can be navigated sequentially: the tabs on the application testing page. When you move to the next tab, the content of the previous tab is hidden so you can focus on the relevant tab.
- **The purpose of each link can be determined from the link text alone or from the link text together with its programmatically determined link context, except where the purpose of the link would be ambiguous to users in general.** - Links are descriptive, e.g. 'Apply For Job' sends you to the page where you apply for a job.
- **For user interface components with labels that include text or images of text, the name contains the text that is presented visually.** - None of the user interface components have labels.
- **The default human language of each web page can be programmatically determined.** - The templating model allows all content to be put in a 'section' to which the attribute 'lang="en"' was added. This value can be read programmatically.
- **When any component receives focus, it does not initiate a change of context.** - No context changes, i.e. new tabs or windows opening, occur within the web app.
- **Changing the setting of any user interface component does not automatically cause a change of context unless the user has been advised of the behaviour before using the component.** - No context changes occur, same as above.
- **If an input error is automatically detected, the item that is in error is identified and the error is described to the user in text.** - Required fields will produce a hovering notification when not filled in before submission.
- **Labels or instructions are provided when content requires user input.** - Clear instructions are given where user input is required, sometimes with example placeholders.
- **In content implemented using markup languages, elements have complete start and end tags, elements are nested according to their specifications, elements do not contain duplicate attributes, and any IDs are unique, except where the specifications allow these features.** - No floating tags occur in the web app and ids, when present, are unique. No attribute was added more than once.

UI Design Choices

When the project had just begun, and before any development had started, the easiest design choice to get started with was picking a colour palette. Figure 4 shows the logo created from the colour palette. As suggested in the Design and Planning Document the team previously produced [10], 5 colours were picked; these can be seen in Figure 5:

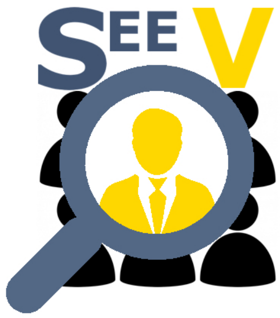


Figure 4: See V Logo

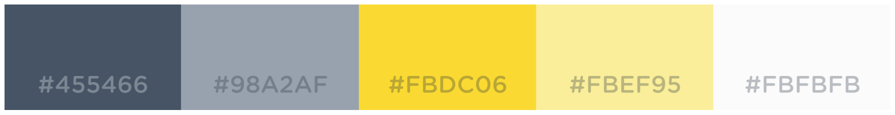
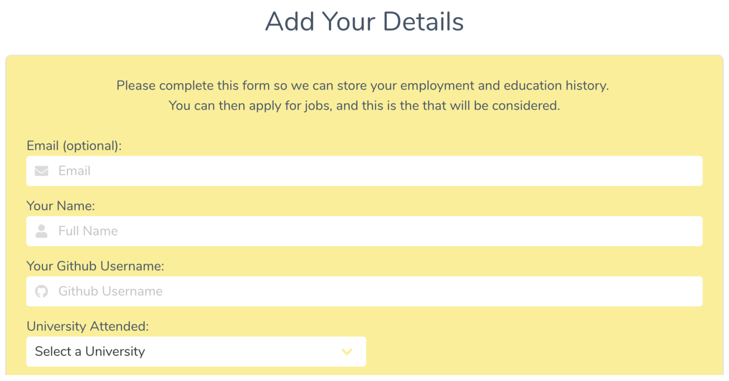


Figure 5: See V Colour Palette

It was decided that white would be the background on all pages, as it improves readability by limiting distractions from the main content. The light yellow was intended to be a light accent colour, used for contrast and for catching attention, and the bright yellow the main brand colour, however, the brighter yellow proved to be too harsh. It was jarring when it appeared in a large area, and it was difficult to read white text on it (as would be the standard in a Bulma button), while text in the darker colours also looked off. In the end, only a few buttons in the brighter colour remain. The paler yellow was cheery and attractive without being distracting or offensive to the eye, and as such became the main page colour. The light grey was used as another accent colour, to contrast directly with the light yellow, while the darker blue-grey was used as 'inverted' colour - the text colour for dark-on-light elements and the background for light-on-dark.

The favoured content container for the project was the Bulma class "box". It is based on the flex-box and so realigns content well, and also has rounded corners to soften the appearance of a page and make it look less formal. Many of the pages of the web app have a box in the light yellow as the main focal point, for example, those seen in Figures 6 & 7.


Add Your Details

Please complete this form so we can store your employment and education history.
You can then apply for jobs, and this is the that will be considered.

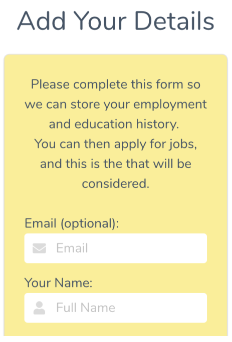
Email (optional):
Email

Your Name:
Full Name

Your Github Username:
Github Username

University Attended:
Select a University

Figure 6: A Box as Seen on a Desktop Monitor


Add Your Details

Please complete this form so we can store your employment and education history.
You can then apply for jobs, and this is the that will be considered.

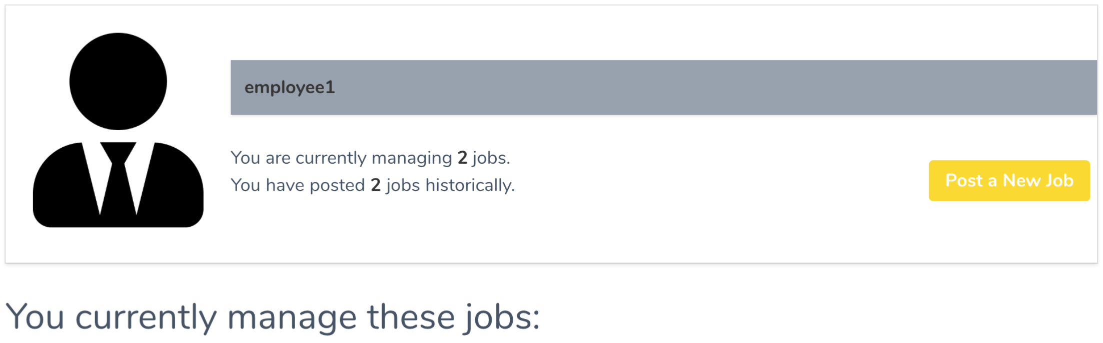
Email (optional):
Email

Your Name:
Full Name

Figure 7: A Box Re-aligning Content for a Smaller Viewport

One of the more unique pages of the web app is the 'apply.html' page where you perform multiple choice questions. This page contains a box as normal, but in white with a yellow border instead of all yellow. It also has tabs. This is largely due to the length of the literacy questions provided in the system. The reading comprehension paragraphs are quite long and this can look odd on the yellow background and also may cause the page to become too long if there are many questions of this type.

Another repeated design element is a small "banner" created for the applicationList, 'clientJob' and 'jobDetails' page. It shows some basic information about the user or job, along with an icon, but is eye-catching at the top of the page thanks to its stripe of grey and bright button.


employee1

You are currently managing **2** jobs.
You have posted **2** jobs historically.

You currently manage these jobs:

Figure 8: Banner from clientJobs

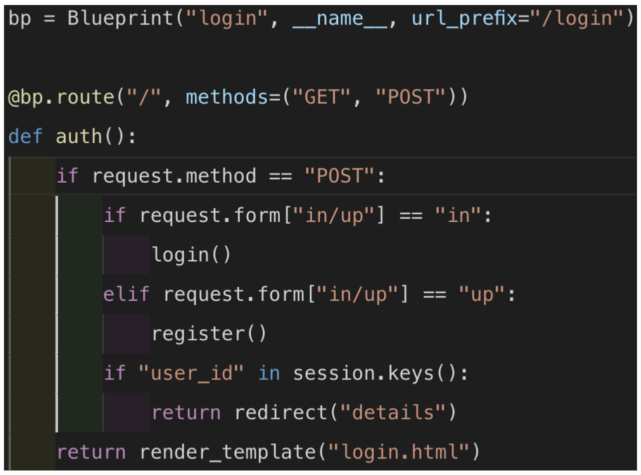
3.2 Backend

API

The backend API system was built using Flask, a Python microframework. Flask dictated the general structure of the project, the components of the frontend are kept in the templates/static subfolders and the APIs in the general project directory [4].

Blueprints: The functionality of pages are defined within the blueprints. For example, the candidate login page blueprint contains all of the code required to login and register for an account; this includes checking the username and password fields, connecting with the database to check if the user exists and setting up the user session. The blueprints are imported into the ___init___.py file, which initialises the app and establishes the appropriate URL path for each function.

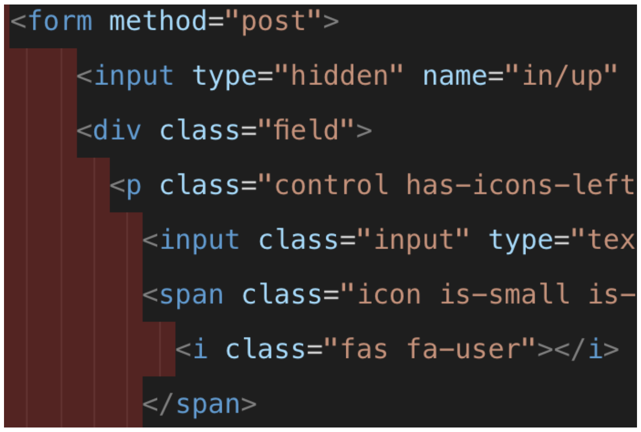
As shown in Figure 9, the auth() function is called when there is action on the '/login' path; this is because the 'login' blueprint has the url_prefix '/login' and the auth() function has the URL '/' which just means the current URL. When the '/login' page is called in a 'GET' request, the 'login.html' template is rendered. When a 'POST' request is made, by clicking the form's submit button (Figure 10), the attribute ["in/up"] is checked. The ["in/up"] variable defines whether a candidate is signing in or signing up. Once this is checked, either the login() or register() function is called accordingly.



```python
bp = Blueprint("login", __name__, url_prefix="/login")

@bp.route("/", methods=("GET", "POST"))
def auth():
    if request.method == "POST":
        if request.form["in/up"] == "in":
            login()
        elif request.form["in/up"] == "up":
            register()
        if "user_id" in session.keys():
            return redirect("details")
    return render_template("login.html")
```

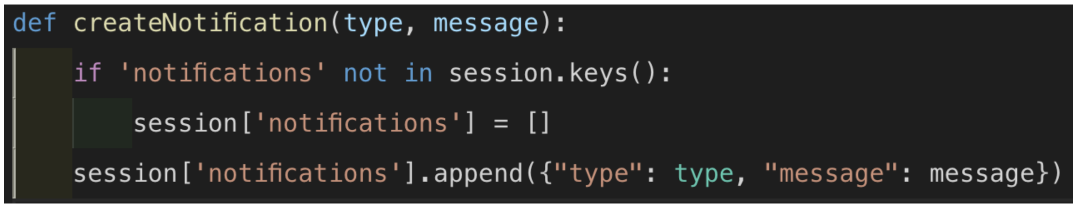
Figure 9: Login Blueprint



```html
<form method="post">
    <input type="hidden" name="in/up"
    <div class="field">
        <p class="control has-icons-left
            <input class="input" type="tex
            <span class="icon is-small is-
                <i class="fas fa-user"></i>
            </span>
```

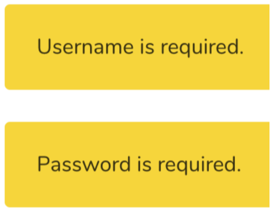
Figure 10: Login Form

Login Feature: The login feature of the app requires the use of a session. The session data is handled in the majority by flask, upon user login the session['user_id'] is set along with the session['client_flag']. The client flag determines whether the user is a client or candidate, True if client or False if candidate. Once the user has logged in, there are given access to more pages and features, for example, a client can view their job adverts. This is implemented using a @client_login_required tag, which is placed before the 'Managed Jobs' function. The client_login_required() function simply checks the session to see if a 'user_id' exists and the 'client_flag' is set to True, if not the user is redirected to the login page.



```python
def createNotification(type, message):
    if 'notifications' not in session.keys():
        session['notifications'] = []
    session['notifications'].append({"type": type, "message": message})
```

Figure 11: Notification System Code Snippet



Username is required.

Password is required.

Figure 12: Notification System Example Output

Notification System: Whilst navigating through the system the API occasionally needs to inform the user of updates and issues. In this system, the Flask session has a 'notifications' attribute; this attribute stores a list of dictionaries, each of these dictionaries contain 'type' and 'message' values (as seen in Figure 11). The 'type' of a notification is either 'error' or 'success' and dictates the colour of the notification on the frontend; the 'message' of the notification dictates what is says, e.g. 'Username is required' (Figure 12).

Notable Backend Functions/Features:

- GitHub contributions - connects to the GitHub API to check for a user's community contributions
- Candidate coding test - takes a python input from the candidate, tests it, and scores them out of 100
- Candidate details auto-fill - if a candidate has already input their details, they will be automatically filled in when they revisit the details page

3.3 Database

Collections

A database system with 6 collections was created. These served to easily store all data needed for each facet of the system (Figure 13).

- **Candidate**: a collection to store the account information and CV of a candidate on the system. This collection stores all the data in one location and prevents user information from being accessed accidentally by another system.

- **Employee**: a collection to store the account information of an employee. This collection was made to be very simple so it could be easily integrated with any existing systems the client may have.
- **Job**: a collection storing all information relevant to a specific job. It stores the ideal qualification information for each job (for training the machine learning model), the information regarding the candidate shortlist that should be generated, and the employee the job was created by.
- **Application**: a collection to connect a candidate to a job they have applied for. It stores the results of their online testing as well as the IDs of both the job and candidate.
- **TestQuestions**: a collection containing all the questions for the online test by storing their type (Literacy, Numeracy, Technical) and difficulty.
- **InputFields**: a collection storing all of the stored skills, universities, degree types, etc. This collection is used for the dropdown lists in the job creation and candidate details pages.

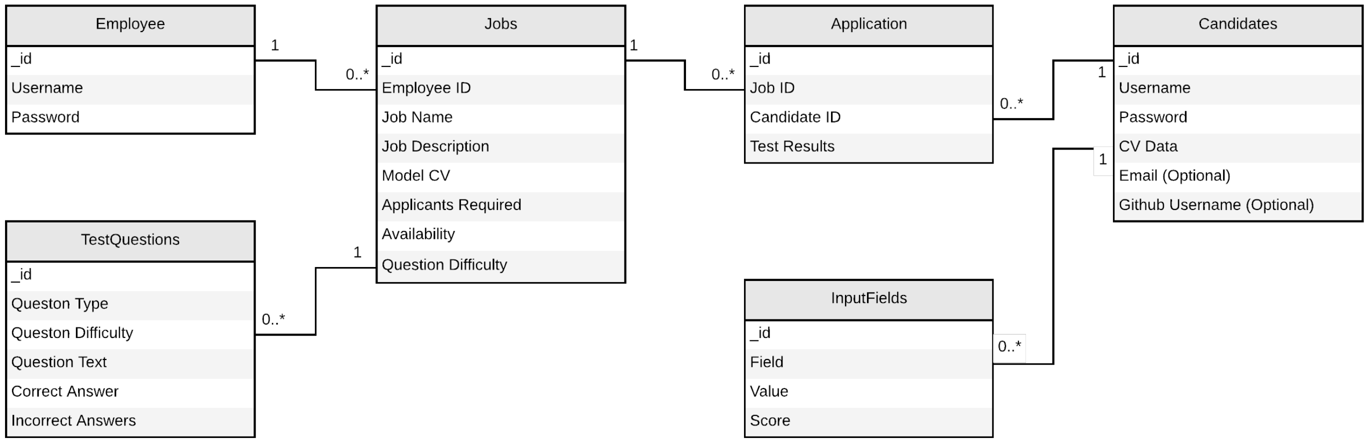


Figure 13: Database Entity Relationship Diagram

Database Functions

To ensure security and code maintainability, all methods regarding interacting with the database were created and held within one file, 'db.py'. This was to allow access and manipulation of the database to be strictly controlled, making it easy to locate database bugs within the system. Additionally, all of the methods could be sanitised locally ensuring safe interaction with the database at all times.

3.4 Machine Learning

Design Choices Justification

The machine learning model was built using the sklearn library in python, which is a high-level API for machine learning. This was chosen over TensorFlow as it allowed for faster integration of the machine learning system.

The majority of preprocessing could not be done easily using standard libraries, as data was given in a JSON format. For data to be read in using pandas and be processed using regular machine learning libraries, nested categories such as skills, languages, and previous employment would need to be scored using a heuristic so it could be converted into a csv and loaded into a panda dataframe. The approach taken instead was to build a bespoke preprocessing suite to extract JSON data directly, as this is what was supplied and the application was built around receiving and parsing JSON data. Each CV is processed by comparing it to a model CV created by the client when it is supplied to an instance of the MLModelTrainer.

A model is created for each job role; this allows for specialised models to be created for each job so that the best candidate is always chosen. The model and any relevant data are saved once it is created. This results in a large amount of data being stored, especially as each model has its own dataset so it can perform reinforced learning for their respective job role. This results in a significant performance delay, running locally on the same server due to the heavy I/O and storage costs. As the client has stated that computational resources and storage are not an issue, the system would perform without throttling.

ML Implementation

Language: python
Machine learning library: sklearn
Supplementary libraries: numpy, math, JSON, pickle, os

The machine learning code is split into four main classes. 'MLUtility', 'MLModelTrainer', 'MLModel', and 'ReinforcedModel' as illustrated in the UML Class in Figure 14. The reason this was done is because it allowed the Machine Learning model to be abstracted into separate classes based on functionality, and be called within the API for what is needed at that point from the model.

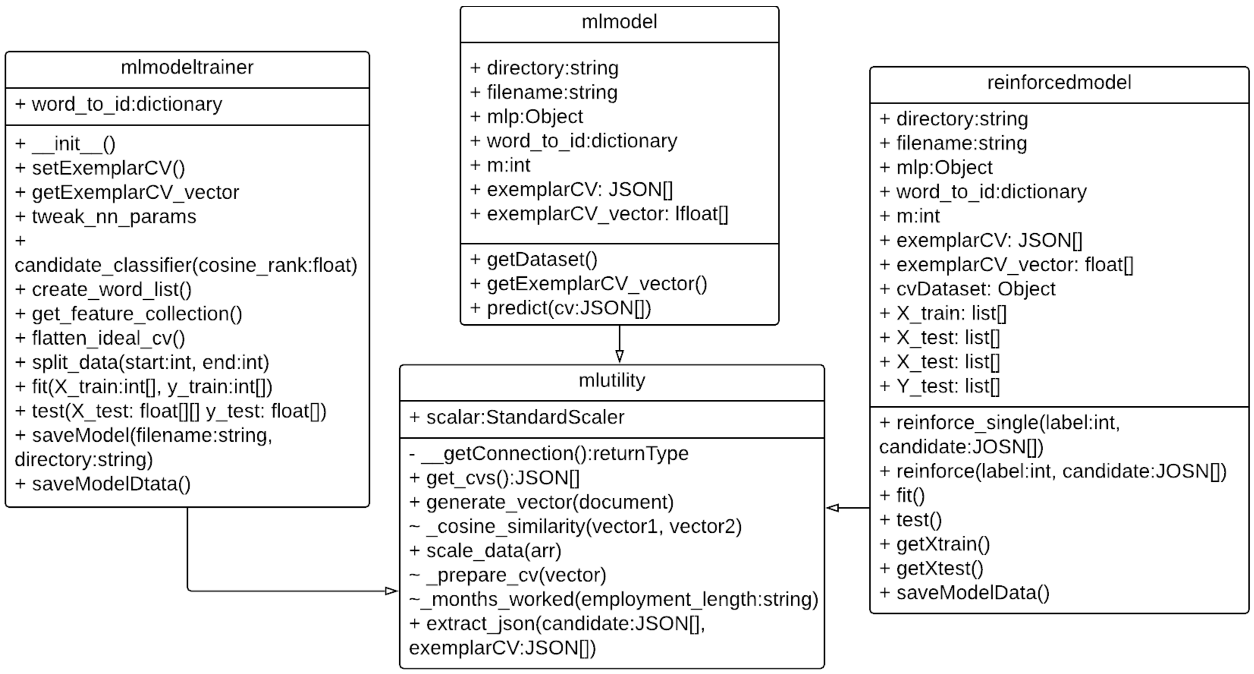


Figure 14: Machine Learning Class Diagram

Creating the Model: This process takes place every time a new job is created. To compare the CVs to a model CV, a word list is created from the given dataset. The respective CVs are then converted into n-dimensional vectors where n is the number of unique words found. A vector is made from the exemplar CV by giving a 1 if a value occurs, and a 0 if not. For fields which have an expertise rating or length of employment, a higher weighting is given by giving a higher number at that index, for that value. To deal with overqualified candidates the system caps their score at the expertise level required. This means overqualified people are given no advantage in the machine learning models process even though they more than meet the criteria, they may be fitted better for another role.

Once vectors have been created, they can be compared. This is done using cosine similarity, which checks the angle between two given vectors. If the vectors are pointing in a similar direction, a higher score will be given. If they are perpendicular, a score of 0 will be given, this means the candidate did not suit the role at all. This value is then used to give a label of 1-10 to each cv based on set weightings, the higher the score the better. This is essentially clustering which will allow for supervised techniques to be used.

Since a label has now been generated for each CV, the data is now scaled, normalised and passed into a multilayer perceptron neural-net. This neural net contains three layers with thirteen nodes, using the default settings given by sklearn; this was found to give the highest accuracy for the model.

Using the Model: The model is used when the client requests to see the list of candidates selected by it. An API call is made from the 'jobDetails' page, which creates an instance of MLModel, passing in its name and directory. This model predict function takes in a candidate's CV then returns a ranking from 1-10 based its contents.

The names of the selected candidates are listed on the client's job page. Once the application window closes the client can then rank the quality of the candidates to give feedback to the system, this will provide reinforcement to the model. This will then update that model's dataset, so when it is re-trained it will use the updated data. This will occur if the client decides to post the same job position again, when another (identical) position opens.

Saving the Model: Each model is saved locally in its own folder named after the jobs name. This is done using pickle in python, to serialize the objects, so they are able to be stored as .sav files. This is what allows the model to be trained and loaded separately throughout the system, it also means if the server ever shuts off the model's state will be saved.

4 Program Flow (Figure 15)

:

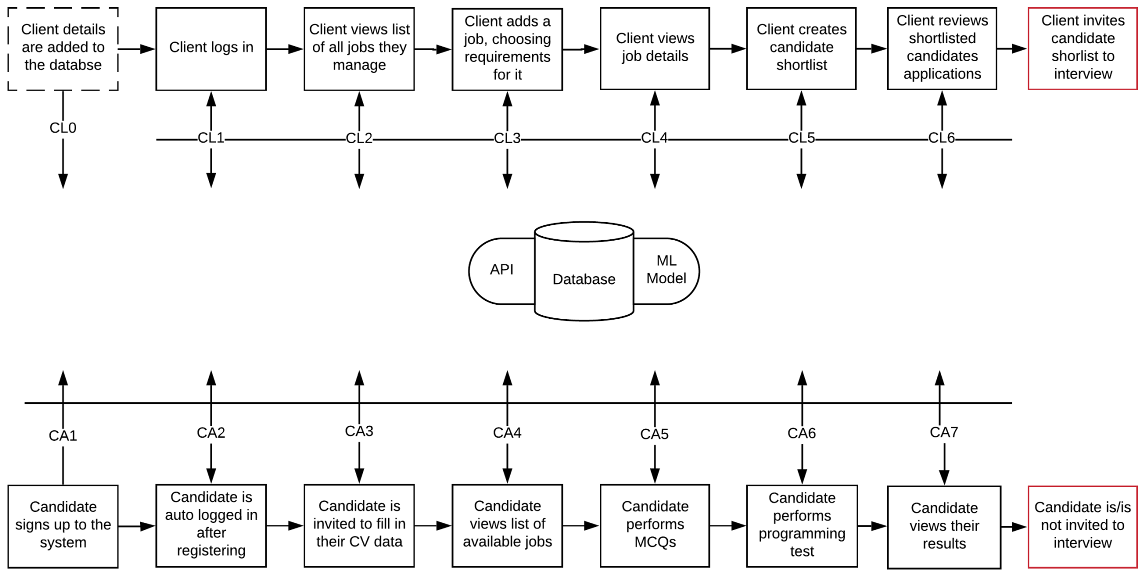

Client details are added to the databse
Client logs in
Client views list of all jobs they manage
Client adds a job, choosing requirements for it
Client views job details
Client creates candidate shortlist
Client reviews shortlisted candidates applications
Client invites candidate shorlist to interview
CL0
CL1
CL2
CL3
CL4
CL5
CL6
API
Database
ML Model
CA1
CA2
CA3
CA4
CA5
CA6
CA7
Candidate signs up to the system
Candidate is auto logged in after registering
Candidate is invited to fill in their CV data
Candidate views list of available jobs
Candidate performs MCQs
Candidate performs programming test
Candidate views their results
Candidate is/is not invited to interview

Figure 15: Program Flow

Client Side:

- CL0: The employee's login details are added to the database's Employee collection. This should be handled by the client company to prevent non-permitted personnel from creating jobs, perhaps using the individual's company login details.
- CL1: The client's login details are POSTed to the API and checked against the values stored in the database, if the input values match the stored values then the API will authenticate the client and create a session.
- CL2: When the template for this page is built it calls upon the database to return 3 lists: a list of active jobs managed by the client and two lists of inactive jobs managed by them (one of completed jobs and one of non-completed jobs, i.e. those that received no applications). This is done by comparing the client's ID to all those stored as 'Employee ID' in the Job collection.
- CL3: When the template for this page is built it calls upon the database to return all previously known universities, degrees, job positions, skills, programming languages and A-Levels. Once the form is submitted, the selected/inputted values are stored in the Job collection. The ML model will create a model CV for the job using this data, which will also be stored in the Job collection.
- CL4: When the template for this page is built, it calls upon the database to retrieve information about a specific job. It retrieves the start and end date of the job (from the Job collection) along with information about candidate applications (from the Application collection).
- CL5: The client can generate the current shortlist of candidates to send to interview. The candidates are ranked using a score generated by comparing their CV and their test data to the model CV created by the client. The client can view the applications of these candidates. If a shortlist has already been generated, then the client can update it. The ML model will re-consider all applicants, including any new ones, to produce the shortlist.
- CL6: If there is a shortlist the client can view the applications of the candidates and then after the job has ended, score their applications out of 10. These values are then POSTed to the API where they can be sent to the ML model to help it retrain.

Candidate Side:

- CA1: On the register form the candidate fills in a username and password. These are POSTed to the API, which calls to check if the username doesn't already exist. If it does, then the candidate is authenticated and moves on to

CA2, otherwise, the page will display a notification and invite the candidate to choose a new username.

- CA2: The candidate is either automatically logged in after registering, or they have gone to the login page where their login details were POSTed to the API and checked against the values stored in the database. If the input values matched some stored values then the API would authenticate the candidate and create a session.
- CA3: When the template for this page is built, it calls upon the database to return all previously known universities, degrees, job positions, skills, programming languages, hobbies and A-Levels. Once the form is submitted, the selected/inputted values are stored in the Candidate collection.
- CA4: When the template for this page is built, it calls upon the database to return a list of all jobs from the Job collection. It will either inform the candidate that they have already applied to the job (if there exists an application in the Application collection with their Candidate ID and the given Job ID) or it will offer them the chance to apply. If the candidate chooses to apply to the job, a GET request is sent to the API to pull up the 'apply' page for the given job, by passing its ID.
- CA5: When the template for this page is built, it calls upon the database to return a list of all the multiple choice questions and answers associated with this job. When the candidate submits their answers they are compared against the database answers and given a score, which is added to their application in the Application collection.
- CA6: When the template for this page is built, it calls upon the database to return a random programming task stored in the TestQuestion collection. The candidate can run their code to receive feedback on it before submitting it. Their score is generated and the value is updated in their Application element.
- CA7: When the template for this page is built, it calls upon the database to return the information for this candidate's application. Their results will be displayed to them both textually, and graphically in a radar chart.

4.1 Installation Guide (For Development)

Requirements

The system can be tested on any computer which supports the technologies mentioned in this document. The ones which require installation are python2, python3, MongoDB and Flask. Included in the code is a file called 'setup', this will install MongoDB, Flask and all of the python libraries required.

Setup File

The 'setup' file in the app directory is for installing the python requirements, MongoDB, python virtual environment and Flask on OSX. This file assumes that the system is already installed with python2, python3 and the installers pip/pip3 and brew. If these have not been installed, tutorials for installing them can be found in the references of this document [11] [12]. The setup file can be run by typing './setup' into the terminal.

Python Requirements

Lists all of the python libraries required for the backend are stored within the python requirements files; these have been generated using 'pip(3) freeze $pythonrequirements.txt'andcanberunusingthecommand'pip(3)install-rpythonrequirements.txt'$.

Running The System:

The submitted code contains a file called 'run', this file contains all of the commands required to run the system locally. These commands include those required to run Flask and the Mongo Database.

4.2 Deployment

Issues with running the system locally

During development, the entire system has been running locally on a commercial laptop. This is beneficial for the software engineers, as it gives them better control when testing and developing the application. Once deployed updates can still be made but they will have to be tested locally in a virtual environment and ensure they don't break any of the production servers on integration. However, running the system locally has shown the amount of load running Flask, MongoDB and the Machine Learning code places on the server.

To successfully deploy this application it should be hosted using a cloud service such as Microsoft Azure to allow for ease of horizontal scaling to deal with heavy traffic on the server.

Deploying the database

The database can be deployed using MongoDB Atlas which is a cloud service hosted by MongoDB which is available on Azure. A migration guide is provided by mongoDB[13] to switch over to MongoDB Atlas. As MongoDB Atlas is a fully automated cloud service built by MongoDB it receives proven and operational security practices helping keep within GDPR compliance and deals with database availability, data redundancy and more[14]. This abstracts some of the complexity of managing the database away and spreads the load of the application across multiple servers.

Deploying the API Server

Azure allows for servers to be hosted on their cloud platforms at scale[15]. This will allow the app to be deployed on a large scale with a platform to update the OS it's hosted on, capacity limiting and auto load balancing. It also allows for source code integration from GitHub allowing for patches to be made and pushed seamlessly during production without downtime.

It also allows for global scale on demand which is important for the client which is a global company. This also reduces needed to worry about any hardware constraints or management as it is all taken care of by the cloud platform provider.

However if the client did decide to host this on their own machines so security is taken care of internally, this would be possible. Backups would need to be running at different data centres owned by the client for redundancy in case one of the servers goes down there is no downtime.

Deploying the Machine Learning

This can also be deployed using Microsoft Azure. Launching the Machine Learning on its own web server would allow for it to store and process data locally without worrying about slowing down the main server.

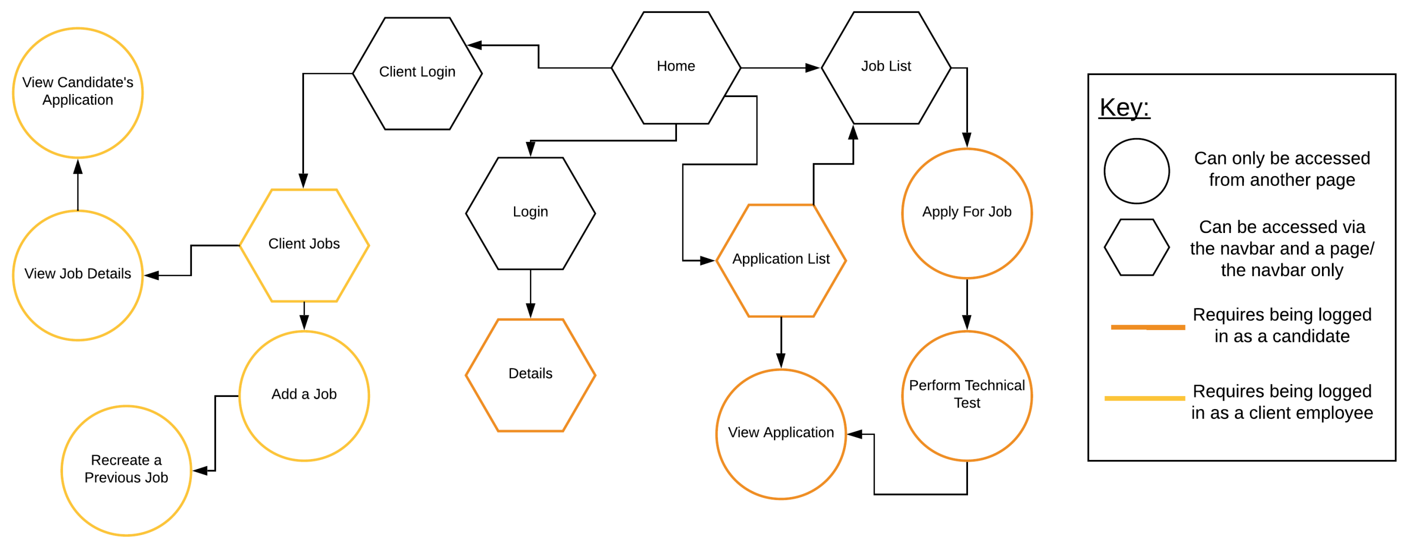
Distributed Deployment

By deploying the system on multiple servers provided by the same cloud platform, it allows for the server load to be spread out while maintaining simplicity by providing the same provider and interface for each server.

4.3 Interface Guide

The web app consists of 14 pages, which can be divided into three categories: non-specific, client, and candidate. The following diagram represents the forward flow of pages in the site:

(To improve clarity no back edges are shown)



View Candidate's Application
Client Login
Home
Job List
Key:
Can only be accessed from another page
Can be accessed via the navbar and a page/ the navbar only
Requires being logged in as a candidate
Requires being logged in as a client employee
View Job Details
Client Jobs
Login
Application List
Apply For Job
Add a Job
Details
Perform Technical Test
View Application
Recreate a Previous Job

Figure 16: Relationship Between Pages

Non-Specific Pages

These pages can be accessed by any type or multiple types of user.

- **Home**: the main home-screen of the web app. It is the first thing any kind of user should see.
  - Can be accessed at all times from the navbar via the system logo

    - Can be accessed from any page via the logout button when logged in
- **Login**: a page with a form to register or login as a candidate. There is also a link to Client Login.
    - Can be accessed whenever not already logged in via the login button in the navbar
    - Can be accessed at all times from a button on Home
- **Client Login**: a page with a form to log in as a client. There is no option to register as a client. This should be handled by the client company themselves to prevent abuse of the system.
    - Can be accessed at all times from a button in Home
    - Can be accessed from a link in Login
- **Job List**: a page showing a list of live job postings. Depending on login status, the output will be slightly different. If the user is logged in as a candidate, jobs that they haven't applied to will offer them the chance to apply, but if they have already applied, then the page will remind them of this and allow them to view their application.
    - Can be accessed from the navbar when **not** logged in as a client
    - Details also redirects here after you submit/save your details

Client-Specific Pages

These pages can only be accessed when logged in as a client.

- **Client Jobs**: a page showing all the jobs that the client is currently managing. It also keeps track of how many jobs the client has ever managed. Each job has a button allowing the client to go to Job Details and view information about the job. The page also has a button that directs the client to Add a Job where they can post a new job.
    - Can be accessed from the navbar when logged in as a client
    - Can be accessed by clicking your username when logged in as a client
    - Both Add a Job and Recreate a Previous Job redirect here on submission
- **Add a Job**: a page with a long form where the client can detail a new job and its requirements. It produces a new job in the system, which can then be applied to etc.
    - Can be accessed via a button in Client Jobs
- **Recreate a Previous Job**: a page that allows a slight shortcut in creating a job. If the client needs to re-list a job, for example, they can load a previous template and redeploy it.
    - Can be accessed via a button in Add a Job
- **View Job Details**: a page showing information about a specific job. It keeps track of when the job was listed, when the listing will close, and how many applications have been made to it so far. This page also allows the client to generate (or update) a shortlist showing the current favoured candidates that should be invited to interview, with an option to view each suggested candidate's application. If the end date of the job has passed, the client is able to grade the suggested candidates' applications.
    - Can be accessed via a job-specific button in Client Jobs
- **View Candidate's Application**: a page showing information about a candidate's application to a job. It shows all of a candidate's CV and test data.
    - Can be accessed by a candidate specific button in Job Details

Candidate-Specific Pages

These pages can only be accessed when logged in as a candidate.

- **Details**: a page where a candidate can store or update information about themselves. It contains their CV data, including their employment and education history, their skills, hobbies, and the programming languages they use. This data is used to help decide their suitability if they choose to apply for a job.
    - Login redirects here after login, allowing you to fill in initial details
    - Can also be accessed by a clicking your username, when logged in as a candidate, allowing you to update your details in the database
- **Application List**: a page showing all the jobs that a candidate has applied to. Each application has a button that allows the candidate to go to View Application and see information about their application to the job.
    - Can be accessed from the navbar when logged in as a client
- **View Application**: a page showing the results of a candidate's application to a job. It displays a radar chart comparing the candidate's skills to the skill requirements dictated by the client posting the job.
    - Previously made applications can be accessed at any time via an application-specific button in Application List
    - Perform Technical Testing redirects here on submission, returning your test results immediately
- **Apply for Job**: a page where a candidate can perform some short multiple choice tests to determine their suitability to the role. On submission, the page redirects to the Perform Technical Testing page.
    - Can be accessed from a job-specific button on the Job List page
- **Perform Technical Testing**: a page where a candidate performs a short programming test. On submission, the page redirects to the Application List page.

– Apply For Job redirects here after submitting its input

4.4 Example Use Paths

A candidate may use the site in the following manner:

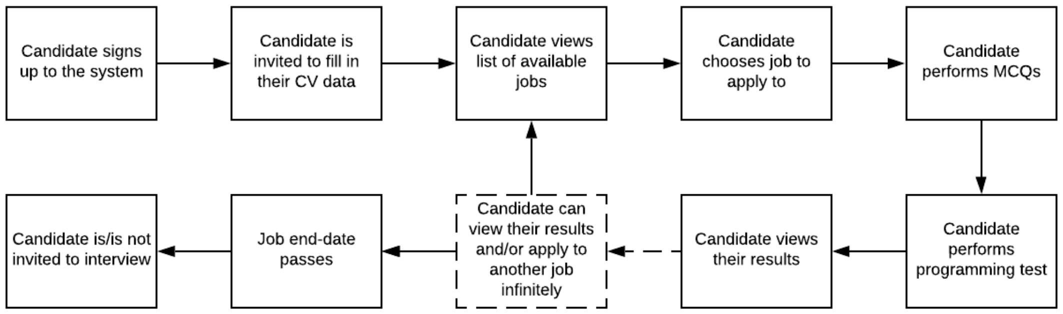

Candidate signs up to the system
Candidate is invited to fill in their CV data
Candidate views list of available jobs
Candidate chooses job to apply to
Candidate performs MCQs
Candidate performs programming test
Candidate views their results
Candidate can view their results and/or apply to another job infinitely
Job end-date passes
Candidate is/is not invited to interview

Figure 17: Example Candidate Use Path

A client may use the site in the following manner:

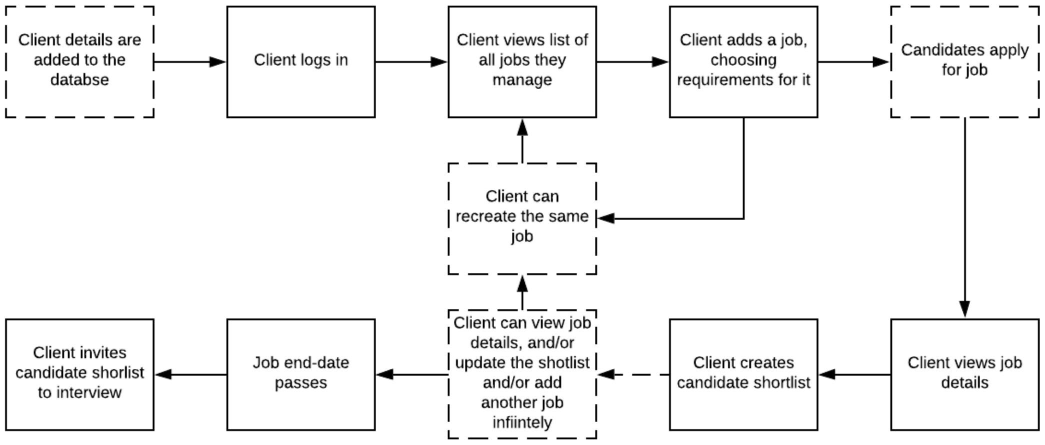

Client details are added to the databse
Client logs in
Client views list of all jobs they manage
Client adds a job, choosing requirements for it
Candidates apply for job
Client can recreate the same job
Client views job details
Client creates candidate shortlist
Client can view job details, and/or update the shotlist and/or add another job infiintely
Job end-date passes
Client invites candidate shorlist to interview

Figure 18: Example Client Use Path

5 Testing

5.1 Unit Testing

Specific, white-box unit tests were carried out on all components of the system to validate key functionality within the system to help in maintaining and expanding the system safely (Table 1).

Test	Input	Expected Output	Actual output	Spec comparison
User can input CV details	Valid set of data	CV stored in the database, system notifies user	Database successfully updates,notification displayed	R.1
Github API successfully called when user details submitted	"Bhavik-Makwana"	Contributions to public repos scored, giving a binary score of 100 or 0	100 stored in database as user has made contributions to public repos	R.4
Candidate feedback displayed after job application process finished	Non-technical test results, technical test results and community contributions score Radar chart graphically displays user test scores	Radar chart graphically displays user test scores	Radar chart produced displaying users' test scores	R.5
Candidate can log in and store their application details in the system	Username: "JaunitaAdell", Password: "password"	Successful authentication passed and details stored in the database	Candidate is authenticated successfully	R.6
Applicants are evaluated and selected by the machine learning model, dependent on the jobs requirements	JSON object of cv matching the job requirements exactly	Applicant is shortlisted as one of the top candidates for the job	Candidate is shortlisted as one of the top candidates to be recommended for interview	R.7, R.8

Table 2: System Function Test Extract

Test	Type	Expectation	Result
Establish a connection to the correct MongoDB database	Normal	Connection is successfully established, no exceptions are thrown.	Connection successfully made
Establish a connection to an incorrect MongoDB database	Invalid	Connection incorrect database selection is detected and exception is thrown	Database is connected to and no exception is thrown
Acquire unique CV data fields from the database	Normal	Dictionary of stored fields returned	Dictionary returned, all unique fields acquired
Create notification to display error message to the user	Normal	Orange text box is displayed upon next page load	Text box is displayed and does not linger when page is changed
Parse CV data into the machine learning object	Normal	The CV data used to train the machine learning model is loaded into the class and no exception is thrown	The CV data is loaded by the system and no exception is thrown
Convert JSON file to a one dimensional array	Normal	A one dimensional array containing the flattened JSON data is returned, ready to be converted into a vector by the machine learning model	A one dimensional array containing all the correctly formatted data is returned
Convert a one dimensional array into a vector	Normal	One dimensional array is processed by the machine learning model and a vector is returned	A valid vector is returned by the system
Machine learning model accuracy	Normal	80% accuracy should be returned by the machine learning model for all classifications	The model averaged at least a 80% accuracy for all classifications given the full dataset

Table 1: System Unit Test Extract

5.2 Functional Testing

Black-box functional tests were carried out on the system to validate the systems functionality when compared to the system's requirements (Table 2).

6 Project Management

This project utilised the Scrum development methodology; scrum is an agile framework geared towards rapid development of software. The initial idea of the project was to rotate the project leader weekly to ensure maximum effort and input from every member of the group. This was done for the first 4 weeks of the project during the design, planning and research stage however it was decided that a fixed project manager would work better during the development stage. The project managers for each week were as follows: Bhavik Makwana (weeks 2 & 3), Antonia Newey (week 4), Simon Heap (week 6), Harry Fallows (weeks 5, 7, 8 & 9).

6.1 Meetings

For the duration of the project there were three official group meetings per week:

- **Monday 1pm-2pm:** Weekly sprint review. The backlog of tasks was monitored and each member's tasks for the week were identified.
- **Thursday 11am-11:30am:** Short meeting with the project supervisor to ensure the development approach was oriented correctly for the client.
- **Friday 1pm-4pm:** Communal work session. Work topic for each member was selected from those allocated in the sprint meeting earlier in the week. This session focused on ensuring that all parts of the system could run together.

6.2 Technical Roles

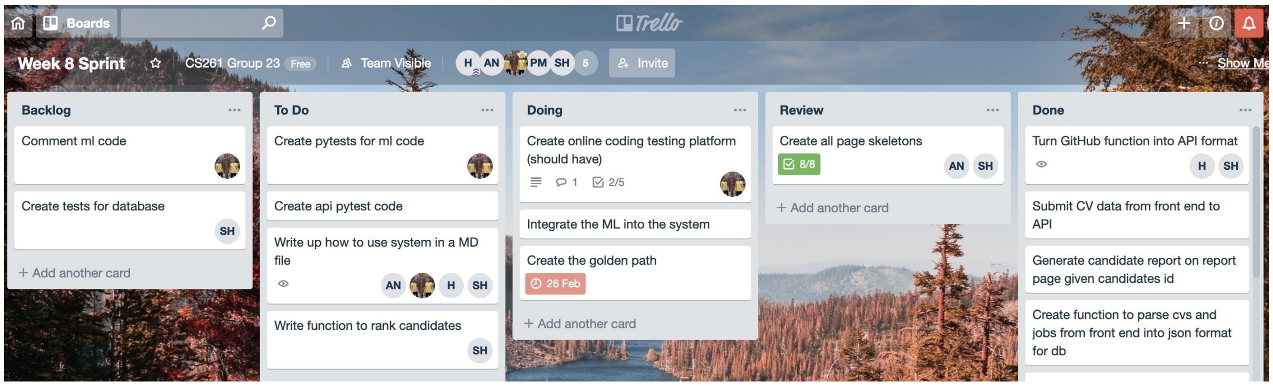
Technical roles were assigned to each member of the group to signify which areas of the system they needed to work on. Sometimes technical roles overlapped, for example, backend with frontend and database. The integration of various parts of the system required input from multiple roles, this involved group work and pair programming.

- **Front End (Antonia):** This role involved creating all of the templates and stylings; the use of Javascript and Jinja in the templates required collaboration with the backend team.
- **Back End:**
  - **API (Harry):** This role involved setting up the whole Flask system and dealing with 'POST' requests from the frontend.
  - **Database (Simon):** This role involved setting up the MongoDB 'schema' and all of the functions for accessing the database. These can all be found within the 'db.py' file.
- **Machine Learning (Bhavik):** This role involved creating the whole CV selection system. This involved training a model based on an exemplar CV given by the client and classifying candidate CVs based on the model.

6.3 Keeping Track of The Project

Trello Board

During the weekly sprint meetings the weekly sprint board was created (Example: Figure 19). This involved going through the previous week's board, making sure everything had been completed and if not transferring it into the backlog of the current board. The use of a Trello board helped everyone stay on track and ensured that there was no overlap between team members' work.



Backlog
- Comment ml code
- Create tests for database — SH
- + Add another card

To Do
- Create pytests for ml code
- Create api pytest code
- Write up how to use system in a MD file — AN H SH
- Write function to rank candidates — SH

Doing
- Create online coding testing platform (should have) — 1 — 2/5
- Integrate the ML into the system
- Create the golden path — 26 Feb
- + Add another card

Review
- Create all page skeletons — 8/8 — AN SH
- + Add another card

Done
- Turn GitHub function into API format — H SH
- Submit CV data from front end to API
- Generate candidate report on report page given candidates id
- Create function to parse cvs and jobs from front end into json format for db

Figure 19: Screenshot of the Trello Week 8 Sprint Board

Tracking Progress

Progress was tracked by looking at the 'Done' list in the board at the start of each week. If all required tasks were completed by a team member, then they were seen to be on track; if not all tasks were completed, the role was re-evaluated and tasks were distributed among other members if necessary.

6.4 Version Control

Bitbucket: For this project a Bitbucket team was created, within this team the Web-App repository was used to store the code produced by each team member. Branches were created for various tasks (e.g. ML), and once some significant progress was made, pull requests were created by the relevant team member. These pull requests were then reviewed by the team in meetings, coding sessions or online calls and any necessary conflicts were fixed. Upon evaluation, pull requests were accepted and merged into the master branch.

6.5 Communication

Slack: A slack workspace was created in order to improve communication within the group. Slack channels provided the ability to target information to the teams who needed it. For example, a message containing information about the API connecting with the database did not need to be seen by Antonia, so was put in the 'backend' channel. Slack also allows for file sharing, which turned out to be very useful when writing the reports.

WhatsApp: For initial communications, a WhatsApp group chat was used; this is how everyone first got in contact. As the project took off Slack was being used more for team communication, however, the WhatsApp was still used for informal conversation and meeting reminders.

7 Module Evaluation

7.1 Frontend

The frontend UI designed for the system is clean and tidy and can display the work that has gone into the functionality and other subsections in an attractive way. Overall there is a cohesive theme to the pages and there are some useful JavaScript features that make the use of the system easier. It could be improved, however, by the complete removal of in-line styling to allow more page layout information to be stored in the cache without being updated. Some efforts were made to do this but time was limited. It would also have been interesting to try out more complex JavaScript functionality, perhaps by making more use of jQuery and its extended packages.

7.2 API

Overall the implemented API system provides a simple and well functioning interface between the frontend and backend logic. Modules are well laid out and relatively uniform in terms of the way they process requests. Some improvements that could be made are:

</> More strict variable naming standards - some confusion was caused due to a lack of communication between team members on the naming of variables, in particular, CV key values
</> More multipurpose blueprints - currently the blueprints tend to only serve one or two functions, increasing the capabilities of some of them could end up reducing code redundancy

7.3 Machine Learning

The model correctly predicts candidates' classifications and takes in user feedback, however in regards to online learning it was only realised that partial_fit could be used with the chosen model[16] once it was fully integrated into the system. This would require changes to 'mlmodel' to be made to incorporate a partial_fit method to train existing models and the removal of reinforced learning. This would be possible, but due to time constraints has not been implemented as there is not enough time for testing. The current reinforced model causes excessive storage of data which is unnecessary, this can be improved through the use of partial_fit.

7.4 Database

The database system works suitably for the system created. However, the methods used to interface with it could be refined and made more secure to account for errors. Additionally, due to MongoDB being a schema-less database system, the Mongoose Object Modelling driver [17] for MongoDB systems could be integrated into the system. This would allow a rigid object model to be set for the database, helping protect the database from storing data that could cause the system to break. This would also aid in the maintainability of the code allowing future users to read the schema and understand the structure of each collection.

8 System Evaluation

8.1 Analysis

The initial task of this project was to create a job candidate selection system for the client.

1. The system should allow employees to post jobs specifying which qualities they require in a candidate.
2. Candidates should then be able to input their details and apply for jobs.
3. The system should select the best candidates for the job and present them to the client.
4. The client should be able to contact the selected candidates to take them through for interview.

These requirements were analysed and a system design was conceived; the following list of requirements was created in order to develop the planned system. Requirements were created using the MoSCoW [18] method, they are split up into 'Must Haves', 'Should Haves', 'Could Haves' and 'Won't Haves' (not included in this document).

8.2 Evaluation of User Requirements

This section references the 'Requirements Analysis Report' [19] previously submitted in the project. The format of this analysis is: Brief overview of the requirement - Necessity - Analysis. For full details of the requirements please consult the original document.

Functional

- **R.1** Users can enter their details - Must Have - System Meets Requirement
- **R.2** Candidates complete non-technical testing - Must Have - System Mostly Meets Requirement, abstract reasoning and situational awareness example questions have not been implemented
- **R.3** Candidates complete programming test - Should Have - System Meets Requirement
  - **R.3.1** The test will be given in the relevant language - Could Have - System Does Not Meet Requirement, candidate testing is only available in python at the moment
- **R.4** Candidates can submit their community contributions - Should Have - System Meets Requirement, candidates can link their GitHub account so that the selection system can take them into account
- **R.5** Candidate feedback:
  - **R.5.1** Candidates receive a radar chart displaying their scores in the non-technical testing - Must Have - System Meets Requirement
  - **R.5.2** The radar chart also contains the candidate's score in the technical testing - Should Have - System Meets Requirement
  - **R.5.3** Candidates do not receive feedback on their CV - Could Have - System Does Not Meet Requirement
  - **R.5.4** Candidates will be sent a copy of this feedback - Could Have - System Mostly Meets Requirement, candidates can view their test feedback on their account
- **R.6 (inc. R.6.1 & R.6.2)** Candidate can log in to save their details and view application information - Should/Could Have - System Meets Requirement
- **R.7** Application selection
  - **R.7.1** The candidates' CVs are scored against requirements - Must Have - System Meets Requirement
  - **R.7.2 (inc. R.7.2.1 & R.7.2.2)** The client can select the volume, regularity and the cut-off date for applications - Should Have - System Mostly Meets Requirement, The client is able to select how many candidates they would like to receive in a given time period and the cut-off date for applications
  - **R.7.3 (inc. R.7.3.1)** Clients can select the number of candidates per position and re-open applications if no candidates pass the interview stage - Should Have - System Meets Requirement, Clients can select the number of candidates they want and re-make a job based on the job template if no candidates pass the interview stage
  - **R.7.4** Allow the client to stop an application - Should Have - System Mostly Meets Requirement, The list of potential candidates is continuously updated so the client can choose candidates from the list at any point
  - **R.7.5** Evaluate applications using ML - Must Have - System Meets Requirement
  - **R.7.6** Evaluate candidates based on tests - Must Have - System Mostly Meets Requirement, The candidates with the highest test scores are selected, the client can choose the level of difficulty of the question asked
  - **R.7.7** Evaluate candidates based on their community contributions - Should Have - System Meets Requirement

  - **R.7.8** Select candidates based on a combination of the above - Must Have - System Meets Requirement
- **R.8** Client interface:
  - **R.8.1 (inc. R.8.1.1 & R.8.1.2 & R.8.1.3)** Generate a report of candidates for a job, with their application details - Must Have - System Meets Requirement, the client is given a list containing the top potential candidates along with their application information
  - **R.8.2 (inc. R.8.2.1 & R.8.2.2)** The client can login to create jobs and view candidate listings - Must Have - System Meets Requirement
  - **R.8.3 & R.8.4** The client can create jobs with specific requirements and re-create old jobs - Should Have - System Meets Requirement
    * **R.8.4.1** The client will be able to select the amount of candidates to proceed to the interview stage - Should Have - System Meets Requirement
    * **R.8.4.2** The client will be able to select a deadline for the specified position - Should Have - System Meets Requirement
    * **R.8.4.4** The client will be able to view the constantly updating candidate list - Could Have - System Meets Requirement
  - **R.8.5 (inc. R.5.1 & R.5.2)** The client can provide feedback for reinforced learning - Must Have - System Meets Requirement, clients can rank the quality of candidates
  - **R.8.6** The client can contact candidates - Could Have - System Meets Requirement, the candidate must input an email address when creating a CV so that the client can contact them
  - **R.8.7 (inc. R.8.7.1 & R.8.7.2 & R.8.7.3)** Mobile application - Should Have - System Does Not Meet Requirement, however due to the adaptive design of the website it should be usable within a mobile browser

Non-Functional

- **R.9** Ease of Use:
  - **R.9.1 (inc. R.9.1.1)** The system will be suitable for non-technical users - Should Have - System Meets Requirement
  - **R.9.2 (inc. R.9.2.1)** The system requires no prior training to use - Should Have - System Meets Requirement, Website is simple and straight-forward to use
  - **R.9.3 (inc. R.9.3.1 & R.9.3.2 & R.9.3.3)** The system conforms to web accessibility guidelines - Level A[20] - Should Have - System Meets Requirement, see frontend section in this document
- **R.10** The system is responsive - Must Have
  - **R.10.1 (inc. R.10.1.2)** The system will be able to display correctly on most screen sizes - Must Have - System Meets Requirement, website layout is adaptive
  - **R.10.2** System performance is validated - Must Have - System Meets Requirement, upon deployment the system will perform well under heavy load
    * **R.10.2.1** The system should aim to keep processing times to a minimum and display a progress icon for longer wait times - Must Have - System Meets Requirement, load icons have been created for pages which require longer loading times
  - **R.10.3** The system remains up to date - Should Have - System Meets Requirement, software is easily reviewable at regular intervals
    * **R.10.3.1** The system should aim to meet the client's most recently stated requirements - Should Have - System Meets Requirement, the system is easily adaptable
    * **R.10.3.2** The client will be able to change their requirements and the machine learning model will adapt appropriately - Should Have - System Meets Requirement, the ML model is retrained for each job, so it can adapt to changing requirements

8.3 Potential Future Improvements

- **Mobile App** - The addition of a mobile app could be useful for clients who want to create jobs and review candidates whilst away from the office; this could be implemented with the same backend as the website.
- **Integration of Hackerrank** - The current candidate coding test is not fully complete and would not be scalable to handle thousands of applicants. The use of a premade system such as Hackerrank would allow for consistent and job-specific tests in a timed environment; this would produce a much more accurate idea of a candidate's coding ability.
- **Settings Page** - A settings screen would be a useful addition to the system for both the clients and the candidates. The settings page could handle notification settings, account deletion and the changing of username and password.
- **Machine Learning Dashboard** - A machine learning dashboard for advanced settings could be made for the client. This will allow them to tweak the parameters models and fine tune it to their needs for each job.

- 🔒 **Security** - Implementing security features in the system was out of scope for this project. Minimal features such as password hashing were the extent of security implementations. Hosting the system on a cloud platform abstracts away some security considerations, such as DOS attacks and database security.

GDPR Compliance: Before deployment of the system, the GDPR must be considered and the correct security measures must be put in place. As mentioned earlier, the system stores personal information about each candidate, some of which uniquely identifies them (such as full name and email). This means that the system would have to be GDPR compliant before deployment to prevent themselves from incurring large fines. It is strongly recommended that security contractors are hired to audit and fix any security issues found within the system. As a general rule, data should be transmitted over a secure network and encrypted or hashed when stored.

Meeting GDPR Obligations[21] Listed below are the requirements and solutions for GDPR conformity. Format: **Requirement**, Solution.

The data controller can only select data processors which provide proof they can perform their processing duties in compliance with GDPR. If deployed on Azure this would be met, as Microsoft provides proof of GDPR compliance[22], as does MongoDB Atlas[23].

Data controllers, as well as processors, must implement appropriate security measures in regards to GDPR, depending on the data. This would involve hashing and salting passwords, encrypting network traffic, logging network activity, sanitising user input to prevent attacks such as XSS or XSRF.

Data Controllers are obliged to inform data subjects if a breach occurs or if a breach is likely to affect them, and inform both data subjects and the data protection authority if the breached data contains monetisable data, no later than in 72 hours. This would involve the client having an in-house security team, or an appointed data controller, to have set up protocols to take place in the event of a breach.

Data processors are only able to process data according to the controller's requirements, specified in the controller/processor contract, meaning the data processor needs to comply with many of the data controllers obligations. Data must only be processed to inform the model on how to better evaluate candidates and must not be leveraged to be used elsewhere.

The processor is obliged to inform the controller about any new sub-contractors, and to reflect the obligations they have with the controller in their contract to the sub-contractor. The client is obliged to inform the data controller if they hire any security contractors to audit the system and report on their obligations to the sub-contractor.

The data processor is obliged to inform the controller if any of the instructions in the contract breach GDPR. A trustworthy controller who has undergone some form of security clearance should be appointed to this role to ensure they can be trusted to do so.

Processors must keep track of all the categories of processing activities. The client must ensure that the data is only being used for the purposes of training the machine learning models for each job. A logging system should be put in place on the system for digital forensic purposes so the client can track any suspicious behaviour.

Data processors must inform controllers in the event of a data breach as soon as possible. The client's employees in charge of managing the system must inform the data controller in the event of a breach.

Both data processors and controllers must appoint a data protection officer in situations such as when their activities require regular monitoring of data subjects on a large scale or involving sensitive data. As sensitive data is being processed on a large scale, it is important that the client hires a data protection officer to oversee these processes.

9 Conclusion

As stated in the evaluation section, most of the requirements have been met or exceeded. Those which have not, have been carefully considered and compensated in other areas. For example, the lack of a mobile app has been compensated with an adaptive frontend design for mobile browser use.

The system is capable of creating and hosting job listings, and candidate applications with CV data via HTML forms. Candidates can be ranked by the ML model in the order to be selected, and passed on to the client. The client can provide feedback on the success of the model by grading the applications of the client, updating the internal models for similar jobs. Both the client and the candidate can see the results of an application in personalised feedback, presented aesthetically.

The system fits the requirements of the initial task and the development process provided invaluable experience for the team.

References

[1] Thomas J. Bulma Documentation (version 0.7.2). Bulma.io;. https://bulma.io/documentation/.

[2] What is REST API Design?; 2017. Available from: https://www.mulesoft.com/resources/api/what-is-rest-api-design.

[3] Creating a RESTful API: Django REST Framework vs. Flask; 2016. Available from: https://www.excella.com/insights/creating-a-restful-api-django-rest-framework-vs-flask.

[4] Welcome to Flask;. Available from: http://flask.pocoo.org/docs/1.0/.

[5] MongoDB;. Available from: https://docs.mongodb.com/manual/.

[6] Kaushik S, Saurav. An Introduction to Clustering different methods of clustering; 2016. Available from: https://www.analyticsvidhya.com/blog/2016/11/an-introduction-to-clustering-and-different-methods-of-clustering/.

[7] Srivastava T. Introduction to Online Machine Learning: Simplified; 2018. Available from: https://www.analyticsvidhya.com/blog/2015/01/introduction-online-machine-learning-simplified-2/5.

[8] Catlin H EC Weizenbaum N. Sass Documentation (Version 0.9.12). Sass; 2019. https://sass-lang.com/documentation/.

[9] John Resig TjT. jQuery API Documentation (Version 3.3.1). jQuery; 2019. https://api.jquery.com/.

[10] Harry Fallows, Simon Heap, Bhavik Makwana, Pavnish Mohan, Antonia Newey. CS261: Software Engineering Project Planning and Design Report - Group 23; 2019.

[11] Download Python;. Available from: https://www.python.org/downloads/.

[12] Homebrew. MikeMcQuaid;. Available from: https://brew.sh/.

[13] Migrate to MongoDB Atlas;. Available from: https://www.mongodb.com/cloud/atlas/migrate.

[14] Fully Managed MongoDB, hosted on AWS, Azure, and GCP;. Available from: https://www.mongodb.com/cloud/atlas.

[15] Web Apps;. Available from: https://azure.microsoft.com/en-gb/services/app-service/web.

[16] sklearn.neural_network.MLPClassifier;. Available from: https://scikit-learn.org/stable/modules/generated/sklearn.neural$_{n}etw$

[17] Mongoose;. Available from: https://mongoosejs.com/.

[18] A Guide to the Business Analysis Body of Knowledge (2 ed.). International Institute of Business Analysis; 2009.

[19] Harry Fallows, Simon Heap, Bhavik Makwana, Pavnish Mohan, Antonia Newey. CS261: Software Engineering Project Requirements Analysis Report - Group 23; 2019.

[20] Eggert E AZS. How to Meet WCAG 2 (Quickref Reference). W3C;. https://www.w3.org/WAI/WCAG21/quickref/.

[21] What Are Your Obligations?;. Available from: https://eugdprcompliant.com/what-are-your-obligations/.

[22] Microsoft Azure;. Available from: https://www.microsoft.com/en-us/TrustCenter/CloudServices/Azure/GDPR.

[23] GDPR Compliance;. Available from: https://www.mongodb.com/cloud/trust/compliance/gdpr.