Computer Science 334
Group 2 report

Johann	22690921
Ethan	22944362
Samuel	22732594
William	23605383
Liam	23913274
Keagan	22804897

March 29, 2022

Contents

1 Introduction

As the field of web development is seemingly growing everyday it is easy to imagine the complexity of the websites created are also increasing. Starting from using basic HTML to upload basic websites, to now using multiple languages to create larger and more sophisticated websites for better user experiences and overall quality improvements. Our job at DevMountain was to implement a website using 3 languages to make it possible for companies to create contracts that required certain skills to complete, could be applied for by developers and for developers to apply for already existing contracts. Firstly we used react for our front-end client side interface this is where all of the rendering and UI would be created, secondly we used MySQL for our database that would store all of our information into a database to be fetched and received by other parts of the program. Finally we used python flask as our backend solution to connect the website together and to be able to communicate between the different components of the website with ease through JSON requests.

2 Front-End React

2.1 Website Layout

2.2 Front end

The purpose of front end is to provide a graphical web based user interface. To move between different web pages we used a built in react hook called **useNavigate**. For the implementation of the forms we used a third party library called **Formik** which greatly improved our work and allowed us to make with in-depth vaildation. For styling we used css created by Jared Palmer (creater of Formik) s base. With some tweaks added in for our needs we used **Tailwind.css** , a low level CSS framework to keep styling consistent across the pages of our site and reduce overhead by applying styling to pages automatically.

Another core feature of DevMountain is to display and organize all available contracts for viewing both from a company aspect (of creating also) and a devloper aspect where they could also apply for a given contract. From the perspective of front end this was achieved through a library called **MUI**, a popular React library which inclused many usefull features such as

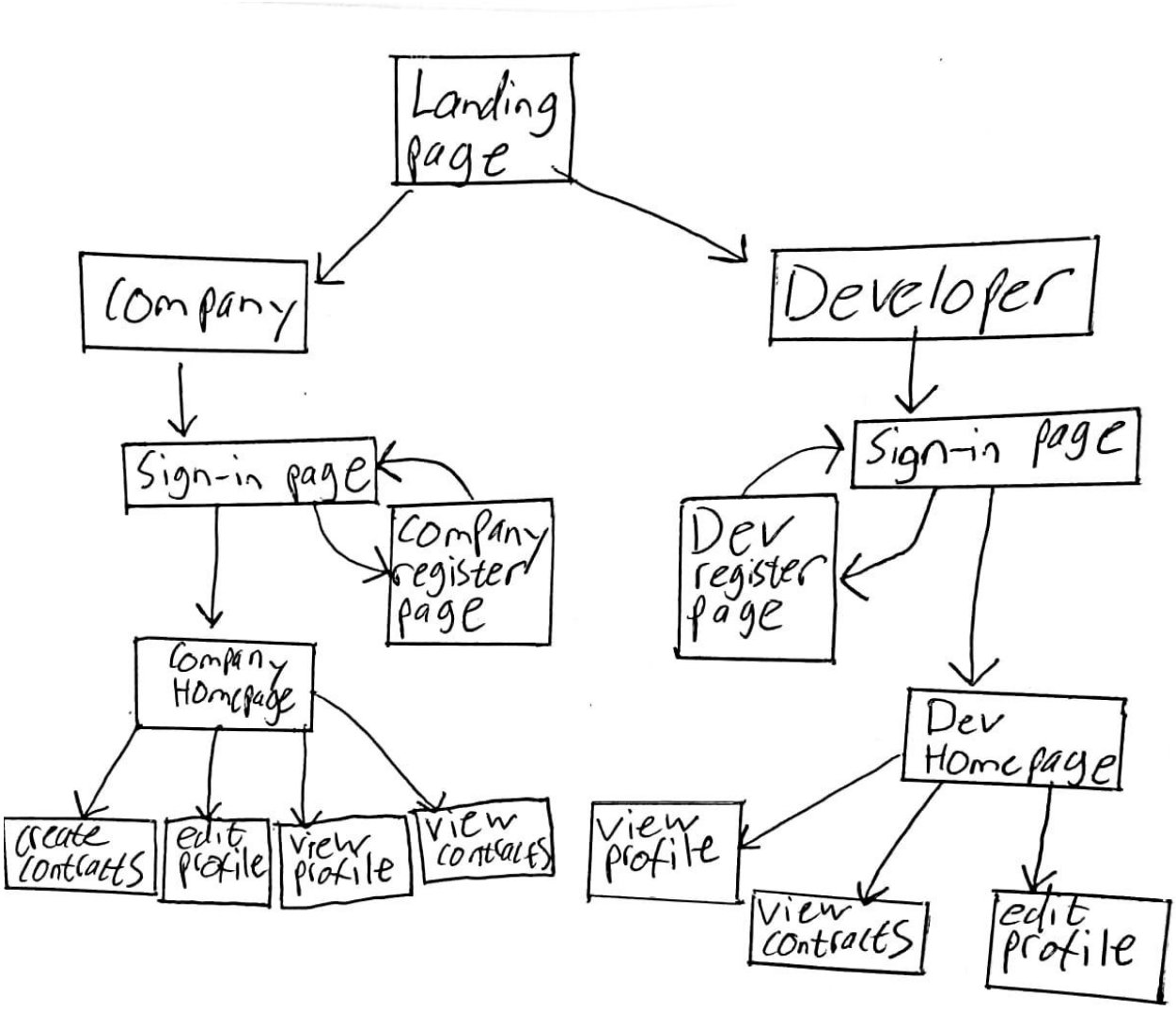

Figure 1: Website path layout

```
const Navbar = (props) => {
    return (
        <div className="flex gap-4 p-3 mb-3 bg-green-800  ">
        {props.isDev ?
            <>
            <Link to="/dev_contracts" className="text-2xl">Contracts</Link>
            <Link to="/dev_profile" className="text-2xl">Profile</Link>
```

Figure 2: A snippet of our navbar

buttons, check boxes etc. From this library we made use of the button feature.

```
<Button variant="text"> Create Contract </Button>
```

This allowed for the creation of JSON requests from buttons which would be sent the back end server. Through the use of MUI we could style and "beautify" our tables and UI. We selected to use "StyledTableCell" which fit in errorlessly and painlessly.

```
conts StyledTableCell - styled(TableCell) (({ theme }) -> ({
    ['&'.\${tableCellClasses.head}']: {
        backgroundColor: theme.palette.common.black,
        color: theme.palette.common.white,
    }
}));
```

Figure 3: An example of mui library

3 Database

3.1 Database Schema

The database schema was designed to be flexible to reduce the productive consequences of changes in the layout of the website or a change in the overall structure of how requests are handled. To maximise flexibility as much data as possible is stored in the "base tables" of the database ie. those tables which store data pertaining to users, companies and the contracts. To connect these tables we make use of lookup tables that represents the many-to-many relationships in the project. For example one design decision was to implement a fixed amount of "languages" the contract could be specified as using or that the user could be specified as proficient in. This leaves oppurtunity for a malicious attacker (or simply a error in transmission) to submit a contract or a user using a language which is not in the fixed set of allowed "langauges". To add a layer of security to check the input a seperate table for languages is created and initialized to contain only the languages allowed. And this table is then placed in many-to-many relationships with the user and contract table. Addition to the language table will not intefer with the design of the rest of the website.

3.2 Object Relational Mapper

To establish the connection between the database and the flask web framework we used the **Flask SQL-Alchemy** ORM. This is used to simplify the process of interacting with the database, which is a **MySQL** server. The ORM allows the database interaction to be done in the same language that the web framework is in (Python). This makes for ease of development and our backend developers need not know the specifics of writing SQL for a web server. For example the database queries sanitation is done automatically in SQL-Alchemy if a string is passed to it.

Figure 4: Display of the database schema

```
CONT = contract.query.filter_by(id=CONT_ID).first()
```

Figure 5: A query filtering contracts by id

```
class language(database.Model):
__tablename__='languages'
id = database.Column("id", database.Integer,
↪  primary_key=True)
dev_id = database.Column(database.Integer,
↪  ForeignKey('developer.id'), nullable=False)
language = database.Column(database.String(30),
↪  nullable=False)

def __init__(self, dev_id, language):
    self.dev_id = dev_id
    self.language = language
```

An example of a table class created in SQL-Alchemy. The class has an initialization method associated which can be used to create new rows. A foreign key is also specified with which to create a relationship with the developer table.

3.3 Queries in SQL-Alchemy

The object level abstraction that SQL-Alchemy provides provides query functionality as functions which are callable on the database table classes. In this project the query all with a simple "filter_by" id was used extensively.

But of course sometimes more advanced filters were required such as returning a list of contracts to which the user has applied by using back references and a lookup table.

```
if get_type == "pending_contracts":
    return {"contracts":conts_schema.dump(DEV.applied_contracts)}
```

Figure 6: Return a list of contracts to which developer has applied which is still pending note: ”cont_schema.dump” is explained in the Marshellow section

4 Python-Flask

4.1 Routes

In DevMountain different routes have to be provided for user/company login and user/company register etc. The flask method of handling such routes are with decorators, yet these decorators only specify which function us used for each route. The flask method of handling the specifics of the request is the request object, which contains the full HTML request sent to the server.

```
@app.route('/devSignup',methods=['POST','GET'])
```

Figure 7: A route decorater specifying which methods it can handle

```
response = request.json
username = response['username']
```

Figure 8: How the JSON content of a response is retrieved

```
if request.method == "POST":
```

Figure 9: How the HTML method is captured

4.2 Handling the JSON

Another consideration is the situation where a function might need to handle multiple GET requests. As an example a company might request different types of contract lists like ”all contracts” or ”all contracts with developers who has applied to them”. The method by which to distinguish between different types of GET requests in DevMountain is to check some value in the JSON itself which the requester needs to specify.

```
REQ = request.json
get_type = REQ["get_type"]

if get_type == "company_contracts":
```

Figure 10: Example of a GET specific request identified with the ”get_type” variable, note the REQ object represents the JSON input of the request

The above figure also demonstrates the method by which the requests content is casted to useable python variables. This allows for the manipulation of the variables such as inputting them into the database or using them for queries.

```
length = REQ["length"]
val = REQ["value"]
description = REQ["description"]
in_office = REQ["is_in_office"]
name = REQ["name"]
Language = REQ["language"]
```

Figure 11: An example of a series of variables retrieved from the request's JSON

4.2.1 Marshmellow

The manual retrieval of each variable in the JSON is primitive and was only made possible in the contract creation page since there is only one such a page the manual retrieval of variables is not such a laborious task. However in contract retrieval where lists of contracts sometimes are to be expected, assigning each variable in a database object to a variable in python and then converting those variables to a JSON request changes that situation. A usefull library in this regard is **Marshmellow**. With this library it is possible to specify which fields to return when a database table row is queried and then to automate the process of converting the database objects content to a JSON response.

```
class developerSchema(ma.Schema):
    class Meta:
        fields = ("id","username","email","languages","email","experience")

dev_schema = developerSchema()
devs_schema = developerSchema(many=True)
```

Figure 12: An example of the developer schema which will output all the specified fields when invoked

```
return {"devs":devs_schema.dump(CONT.applied_devs)}
```

Figure 13: An example of a JSON responses created with marshmallow which returns a list of developer contents

4.3 Session data

To handle company login Flask must somehow keep track of user data through the use of cookies. To do this the Flask "session" dictionary object is used. This object keeps a private cookie on the user's web browser. This cookie is encrypted so there is no need to fear tampering either by the user self or a malicious attacker. In DevMountain whenever a user logs in their id (which is their primary key in the database) is stored in the session. When the user moves away from the login page to some other page which requires content specific to the user this cookie variable is retrieved and the database is queried for the user to see whether such an id exists. Because this id is assigned to a user at login by the backend and since it is encryted an attacker will never be able to login with another user's id without their password.

```
dev_id = int(dev.id)
session["dev_id"] = dev_id
```

Figure 14: A user's primary key id is added to the session

```
REQ = request.json
try:
    COMP_ID = session['comp_id']
except Exception:
    return {"status":False,"details":error["NotLoggedIn"]}
COMP = company.query.filter_by(id=COMP_ID).first()
```

Figure 15: Before any requests are handled it is checked whether a company is logged in note the error handling explained in error handling.

4.4 Error handling in Flask

To create neat code for error handling in Flask an error dictionary is used. Here all the details of the different types of errors can be posted.

```
errors = {
    "NotLoggedIn" : {
        "message":"Not logged in",
        "code":404
    },
```

Figure 16: An error in the errors dictionary

```
if CONT.company.name != COMP.companyname:
    return {"status":False,"details":errors["Unauthorized"]}
```

Figure 17: Contract code example where company error is returned if a company tries to change a contract which is not it's own

5 REST API

5.1 A brief explanation of REST

A RESTful web service does not refer to a specific technology stack but rather to a set of rules by which the transfer of information and the seperation of logic is adheres. The six rules for a RESTful web service is

1. **Uniform Interface**
2. **Client-Server Decoupling**
3. **Statelessness**
4. **Cahcheability**
5. **Layered System Architecture**
6. **Code on demand**

The specifics of these rules and their advantages are outside the scope of this project report. In short the web service is divided into layers, each layer communicates with the layer below and on top of it with a agreed upon protocol that is designed to reduce traffic. It also provides the business logic of the previous layer as an abstraction to the layer that interfaces with it.

5.2 Advantages of being RESTful

In this project the greatest advantage to using a REST approach is the parallelizing of the back end and front end development.If the information transfer protocol has been agreed upon by each developer in their respective layer (in this case the React developers and the Flask developers) then each layer can be developed independently from the rest. This is a big advantage of using a front end server like Node JS and React in comparison to using a simple templating engine like Jinja. One of the requirements of being RESTful is that each layer in the service operate on it's own server so that should the Flask server be run on the local machine or on

a completely different server it should make no difference to the front end. This can not be accomplished with Jinja templating (or at least not easily The seperation of logic also avoids confusion as the code for each layer is in a seperate file. This means that two developers code will not intermingle unless they work in the same layer and this greatly reduces conflicts not just in terms of the code being easy to understand for each developer but also in terms of the project behaving well in the version control system being used (Git). Each layer (Flask en Node) can create it's own branch and develop on that branch and all work , when ready, can be merged with the master branch for deployment.

6 Operating Environment

Our solution stack for this website was a mixture of React (in combination with MUI), JSS, python (flask), SqlAlchemy and Tailwind for more styling options. All instances were either run locally or on Git, with multiple sub branches of everyone's work that eventually got merged into a master. When run locally we used npm which is a package manager for javascript files and node. These two packages helped us run our site by using "npm install" which installed all necessary dependencies that were essential for the site to run. After that we used "npm start" which simply ran the code and started the site on a local port.

7 Appendix

7.1 Description of work done by each member

Ethan Worked on front end applications, mainly the contracts (dev and company) including routing and styling and the report.

Samuel Supplied the front end guys with my react course and resources. Set up the styling and forms across the frontend. Helped plan and organize the team and ensured that everyone had tasks and that all the work was being done. Wrote dev profile, dev registration, dev sign in, did the linking for most of the site using react router, set up the forms using formik, did the majority of git merging and cleaning up the file structure of the repo.

William I wrote our frontend landing page and helped link other pages. Initially I spent a lot of time learning flask,json,flask routes and how to link up frontend with backend.

Johann Wrote the backend implementation for the contracts (devcontract and compcontract). Implemented Marshmallow. Designed the database schema. Did sections 3 (Database), 4 (Python-Flask) and 5 (REST API) of the report. Also typed out the report in latex.

Keagan Developed the whole zenoffer.py file with all the database, routes and password hashing implementations. As well as implementing the frontend retrieval of backend information (i.e. the getDeveloperInfo() , getCompanyInfo() , getEmail() , logout() etc. functions) and navigating to the necessary pages.

Liam Worked on front end login in page for developers and all bits of work to do with login. Assisted with routing and various miscellaneous tasks on the front end.

8 Demo Videos

Video 1 `https://stellenbosch-my.sharepoint.com/:v:/g/personal/22732594_sun_ac_za/Ea74S-owmm`
`e=43PeSo`

Video 2 `https://stellenbosch-my.sharepoint.com/:v:/g/personal/22804897_sun_ac_za/EeuyaHpi6j`
`z-RVch8V0Zvg?e=O5UiLb`