UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Computer Science Report 1
KONTRA

Group 8:
Phillip Schommarz (22621105)
Nicol Visser (16986431)
Lizé Steyn (21772037)
JC Mouton (22549889)

March 29, 2022

Contents

List of Figures

Nomenclature

Acronyms and abbreviations

CMS	Content Management System
MVC	Model–View–Controller (design pattern)
URL	Uniform Resource Locator
GUI	Graphical User Interface

Chapter 1

Introduction

Connecting companies and their short-term contracts with software developers is often not a straightforward task. Many times, for a software developer to find this type of work, they would have to peruse the websites of various companies; a time-consuming and arduous process.

This project aims to solve that problem by providing a centralised web-space where software developers and companies can connect for this exact purpose. The project, going by the name **KONTRA**, allows the creation of a unique company or developer profile, and then providing a seamless and minimalist environment for interaction.

Company profiles can create job listings which summarise details about pieces of software they want developed. These contract listings allow the creating company to list specifics such as work duration, contract value, required programming languages, and whether a contract is to be performed remotely or in office.

Developer profiles can then examine all open contracts and apply for ones which interest them. After such applications have taken place, the ball is once again passed into the company's hands; allowing them to view all developers who have applied for their contracts and selecting one applicant for the job. Upon acceptance, the chosen developer is made aware of their approval and the contract is automatically listed as closed.

Chapter 2

Overall Description

This website serves as a platform for developers and companies to do business as intuitively and as quickly as possible. Such a system that allows users to so easily interact with one another requires the development of many different design components.

Firstly the needs of different user types are considered to determine possible use cases in the system. In order to implement these use cases the provided data must be efficiently modelled and stored in a database. The backend is developed to provide the ability to edit the data and access the desired information. A user interface or frontend is developed to grant users the ability to interact with the data.

This chapter describes each of these components and the different technologies and design approaches associated with them.

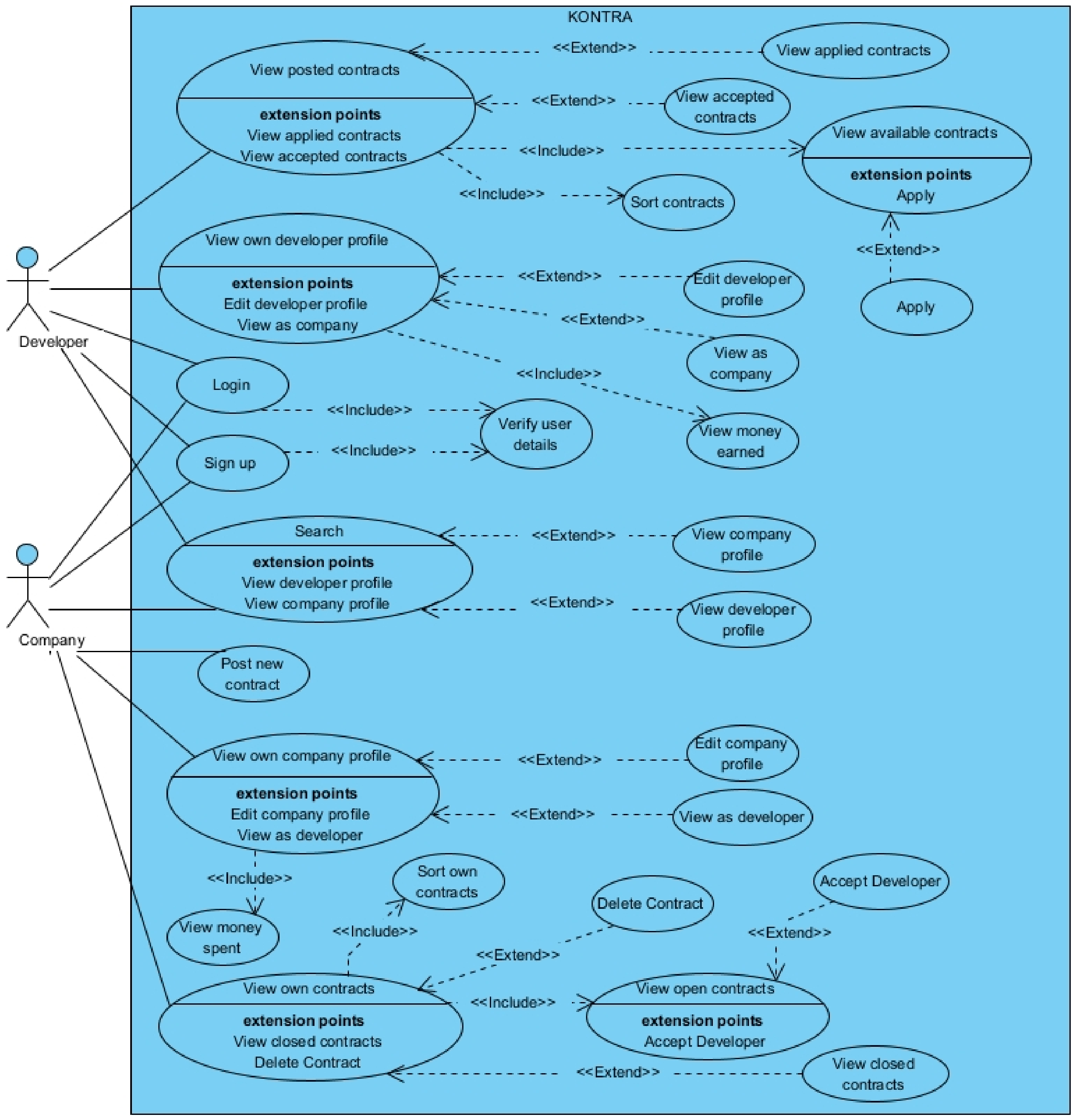
2.1. Use case

The needs of different user types are assessed to form a list of different use cases. These use cases, as illustrated in Figure 2.1, describe how users interact with the system. In this system there are two main user types or actors namely developers and companies.

Both developers and companies are first registered through a sign up process and thereafter require a login each time they wish to interact with the system. Each will be able to view their own profiles and that of other users, although each view will differ according to user types. Companies may post or delete contracts. Developers will be able to view all posted contracts, but companies will only be able to view their own. In both cases, various filtering or sorting of contracts can be done. Developers may then apply to open contracts and companies may accept a single developer from the list of applicants.

As a history of closed contracts is built up, the system will track the total money spent by companies and the total money earned by developers. This information may be viewed on their personal profiles.

Each use case will either collect, manipulate or return certain data to the user. This detail is not captured in the high level representation of a use case diagram, but the next section will provide more insight into the types of data associated with each use case.



;

Figure 2.1: Use Case Diagram

2.2. Data Modelling

In order to optimise the implementation of all the different use cases, as described in the previous section, the data must be modelled efficiently. This is achieved by normalising the database to its second normal form and thus eliminating any many-to-many relationships. This section will discuss the reasoning behind the main data modelling decisions made for the system.

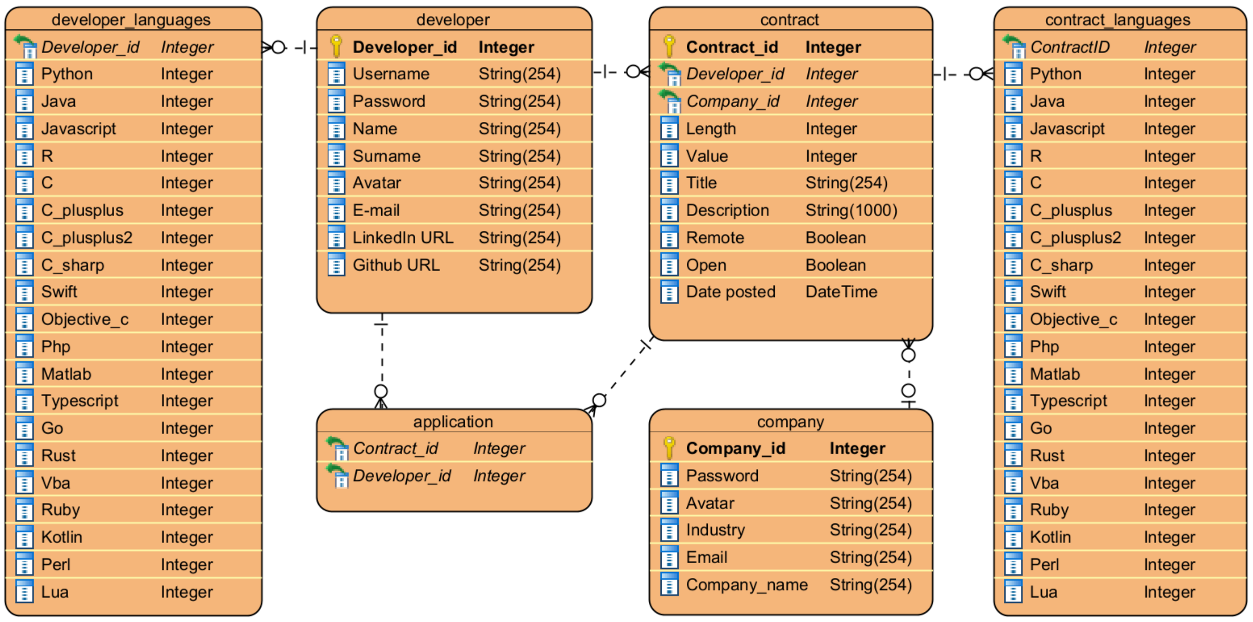
The main entities in this system are developers, companies and contracts. These entities serve as the primary keys and each is assigned various attributes. The relationships between these entities are shown in Figure 2.2.

To summarise section 2.1: companies post contracts, developers apply to the contracts and companies accept a single developer per contract. To achieve this interaction a contract is assigned both a company ID and developer ID as secondary keys. This relates the contract to the company who has posted it as well as to the developer that has been accepted by the company.

However, during the application process many different developers may apply to a single contract and a developer may apply to many different contracts. This is achieved by adding an application table that simply links developers to the contracts they have applied for.

Finally, a challenge presented itself in the form of developer's experience in different programming languages as well as the contract's suggested experience in different programming languages. The first problem is that there is an ever increasing amount of programming languages used in the world. Secondly, each language added by the developer or company has its own attribute in terms of experience.

There are numerous ways to solve this situation, but here we will describe the thought process behind our approach. To address the first problem, we decided to limit the programming languages to a rather extensive, but limited list of popular languages. This list is used as the attributes for two different tables namely, contract languages and developer languages, and is stored on the server side. The tables seem to be duplicate but are simply kept separate as they do not hold any relation to each other. The value of each field is then simply an integer value representing the number of years of experience associated with the respective language. This then solves the second problem.

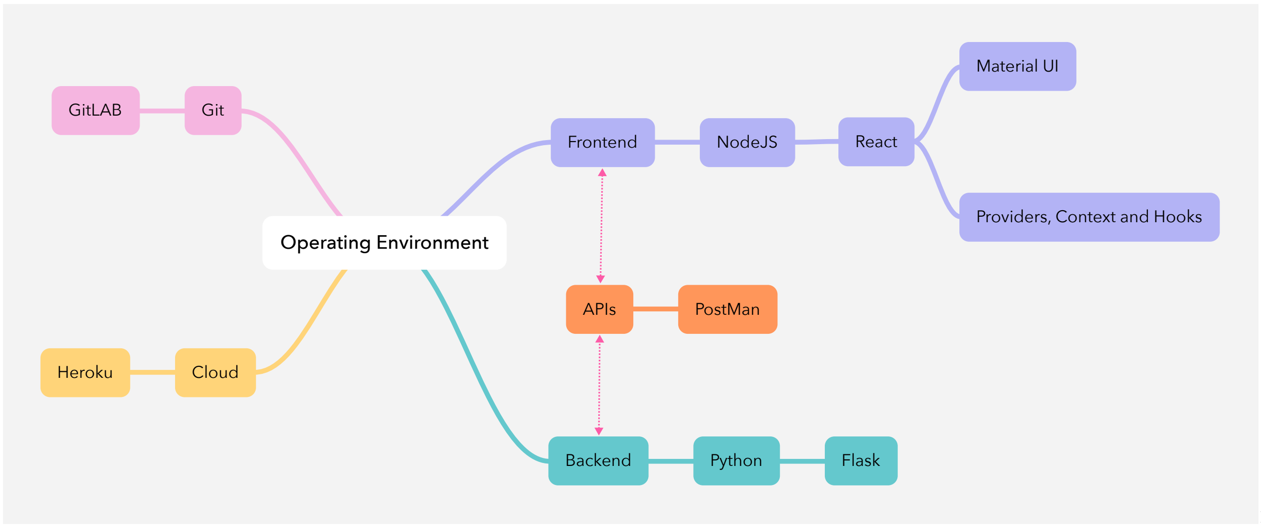


;

Figure 2.2: Entity Relationship Diagram

2.3. Operating environment

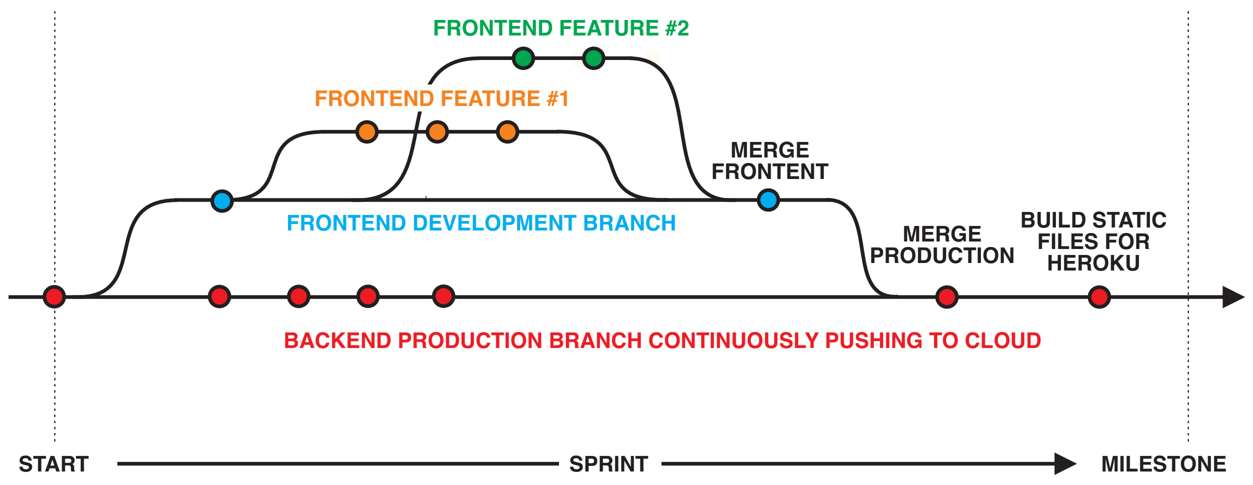
There are a wide variety of technologies available to build a website. Figure 2.3 shows the operation environment of our website with some of the technologies we have decided to incorporate.



;

Figure 2.3: Operating Environment of our Web App

We have source control via git, hosted on GitLab. We have found the git workflow as depicted in Figure 2.4 to suit the needs of our team best. After meeting up and discussing requirements, we would have a sprint of work in order to reach a milestone. During each of these sprints, the backend team implements the APIs as soon as possible and pushes them to Heroku such that they are accessible to all developers. The frontend team will branch away from the main production branch and further branch out to develop independent features. As soon as all features are implemented and tested, there is a sequence of merging (and further testing) until all functionality ends up in the main branch. After that, the static files are built and pushed to Heroku. Final testing is then done on the cloud and the milestone is marked as reached.


FRONTEND FEATURE #2
FRONTEND FEATURE #1
MERGE FRONTENT
FRONTEND DEVELOPMENT BRANCH
MERGE PRODUCTION
BUILD STATIC FILES FOR HEROKU
BACKEND PRODUCTION BRANCH CONTINUOUSLY PUSHING TO CLOUD
START
SPRINT
MILESTONE

;

Figure 2.4: Git Workflow during Development

2.3.1. Frontend operating environment

JavaScript is commonly used to build the frontend of websites instead of plain HTML and CSS. When using JavaScript the user is able to interact with the website without having to load a completely new HTML page on every interaction.

React is a JavaScript library for developing powerful and responsive web apps. React's state management system allows developers to efficiently split up the web application in reusable components. If developers follow React design principles, it will ensure that only relevant parts of the web app get reloaded on each interaction from the user.

We decided to go with React rather than other frameworks due to its slightly lower learning curve and the wide amount of information available due to its popularity. In Section 3.1 we will further discuss our web app's design from a React perspective.

We used Create React App to create our React workspace in a NodeJS environment. In this workspace we set up ESLint globally such that the developers can quickly identify problems with their code and also to enforce a code style and formatting that is consistent over the workspace.

From the start, we wanted our web app to be deployable to the cloud. We decided to go with Heroku to host our backend server and to route the user to the static files built by React. By setting up Heroku from the start, we were able to identify bugs that happen only on the cloud early on. This would have been harder to debug if we only deployed our web app near the end of the project. The backend was also able to host the database and the latest version of the API to the cloud, such that frontend developers did not have to run the backend development server to be able to use the API during development.

Our web app can be accessed by going to cs334proj1group8.herokuapp.com. Please note that we are using a free tier of Heroku. This means that the server will go into a sleep mode and that the first time loading the site might take a while.

2.3.2. Backend operating environment

Python was chosen as the developing language for the backend as it is easy to learn, has simple syntax and a large set of pre-written libraries to work with. Building a backend from scratch is time consuming and unnecessary for a small project, therefore a framework was used. The Flask python framework is ideal for minimalist projects, it gives the developers the essential tools for web design without cluttering the workspace with unnecessary features. Flask is a lightweight framework (microframework) that can easily churn out a simple project unlike, Django that is better suited for larger projects that require more functionality.

Database management

PostgreSQL is currently one of the best database management systems. It is free, reliable, and open-source. For these reasons PostgreSQL was picked over MySQL. To be able to view and mangage our database (hosted on Heroku) in a non-terminal environment, pgaAdmin was used to provide a GUI.

Postman

In order to accommodate communication between frontend and backend, specifically what format API calls should take, Postman was used during development. This API platform was populated by our backend developer, allowing the frontend to quickly and easily recognise what kind of HTTP request to use, what data to pass in the body of the request, as well as in what format data will be returned. The full API can be viewed on https://documenter.getpostman.com/view/18073555/UVyoXdhA.

Chapter 3

Solution

3.1. Frontend Solution

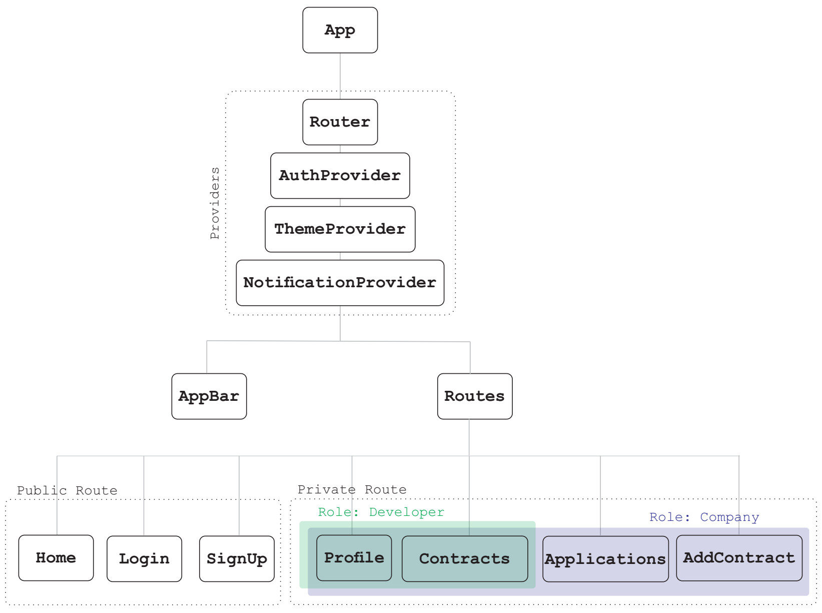
On a high level, we tried to follow two main design methodologies while developing the frontend. Most importantly we wanted modular, reusable code. Furthermore we wanted our code to reflect a Model-View-Controller (MVC) design pattern.

Since we tried to keep our code modular and reusable we have, for example, an *AvatarPicker*, a *LanguagePicker* and an *ExperiencePicker*. These are reused in various places in the web app: When signing up, when editing a profile and when creating a contract. Any improvements we made to these components over time, reflected everywhere we used it. This also meant that we had to do proper testing after making changes to such a component, ensuring that it works everywhere it was implemented.

We tried as far as possible to separate our code into parts that are purely responsible for the user interface (view) and parts that are purely responsible for updating the data in the backend (model). This was a big challenge, especially considering how new we were to using the React library. But as we will discuss later, we figured out how to extensively use React's context providers and hook to get closer to a purely MVC design pattern.

We made use of various technologies in the frontend such as React, React Router, Custom React Hooks, Material UI, Axios and the ImgBB Image hosting API. Each of these will be discussed in detail in the following sections.

3.1.1. React Library



;

Figure 3.1: React App Structure

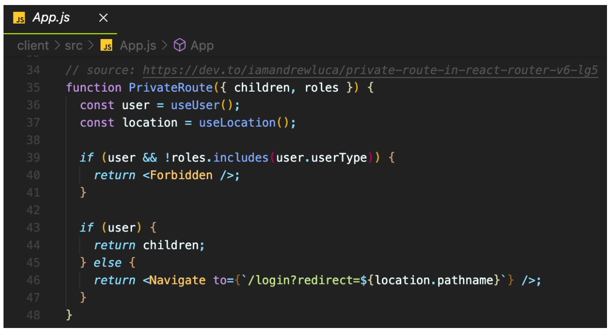
At the heart of the frontend solution lies the React Library. Due to it's maturity and industry-wide use, this framework was chosen. Its component-based design allowed us modularity and re-usability in the development stage. Our React app has a structure as shown in Figure 3.1.

3.1.2. React Router

The React Router library provides a way to perform nested routing. By using React Router properly, we could keep some components such as the AppBar and the Notification-Provider (the notification displaying system explained in Section 3.1.3) active, while the user appears to be changing pages. This makes for a far more enjoyable and smooth user experience.

If we look again at Figure 3.1, on the lowest level of the App structure, we have components that correspond to the different pages the user is visiting. The React Router library allows us to show the correct component based on the route (URL) the user is visiting.

Some of these pages should only be shown to the user under certain conditions. Therefore, we have the notion of public and private routes. *Public routes* are visible to any user, regardless of whether they are logged in or not. *Private Routes* are only visible to users that are currently logged in. Furthermore, we only want users with certain roles to access certain pages. Developers should not be able to see applications made to a contract or to add a contract themselves. Therefore, we restrict certain pages only to the *role* of the logged in user.

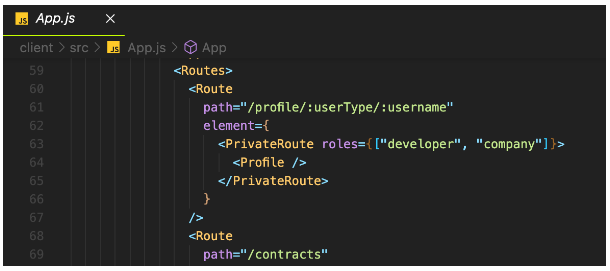


```
// source: https://dev.to/iamandrewluca/private-route-in-react-router-v6-lg5
function PrivateRoute({ children, roles }) {
  const user = useUser();
  const location = useLocation();

  if (user && !roles.includes(user.userType)) {
    return <Forbidden />;
  }

  if (user) {
    return children;
  } else {
    return <Navigate to={`/login?redirect=${location.pathname}`} />;
  }
}
```

(a)



```
<Routes>
  <Route
    path="/profile/:userType/:username"
    element={
      <PrivateRoute roles={["developer", "company"]}>
        <Profile />
      </PrivateRoute>
    }
  />
  <Route
    path="/contracts"
```

(b)

Figure 3.2: (a) *PrivateRoute* Component (b) Routing with *PrivateRoute* Component.

With help from an article by Andrew Luca [2] we had to extend the React Router to allow for such functionality. Instead of simply passing the component which we want the router to render, we wrapped the element in a newly created PrivateRoute component as shown in Figure 3.3b. This PrivateRoute component, whose code can be seen in Figure 3.3a, ensures that its children are only rendered under the correct permissions. Otherwise it will redirect you to the Log In page, or to a 403 Forbidden page.

3.1.3. React Context Providers and Hooks

One of the most powerful features of the React library we found was to use Context Providers and custom Hooks. Context allows you to pass data down to components without sending them component-by-component via props. Custom hooks can then be written to access this data or even to access functions to manipulate this data. Combined these provide a way to reuse stateful logic without changing component hierarchy. In Figure 3.1 you can see a list of these context providers that we used in our project.

Authenticated User Provider

After logging in or signing up, we need some details of the user to be stored in react state in order to show content relavent to the user. We have an *AuthProvider* that stores this information as state. The *AuthProvider* is in charge of synchronising this state with the session storage of the user's browser. This keeps the user logged in if they happen to hit refresh or navigate to a specific URL manually. In addition, this component has various functions that relate to authentication of the user such as logging in, logging out and verifying a password. Any other component can now access the authenticated user data and call these functions via the custom hooks. For example, the AppBar can access the user avatar, render its menu items based on whether a user is logged in and it can sign the user out if the user clicks on 'log out' in the menu. This allows for a completely separated model, view and controller in the MVC design pattern that we are aiming for.

Notification Provider

We wanted notifications to pop up on both the failure and success of a user's actions such that the website feels very responsive. We wanted them to interfere minimally on the success of an action, but also be noticeable enough in the case of an error. We quickly ran into issues where for example after signing in, the notification shows only for a split second before the user is redirected to another component. We decided put the notifications inside a top level component called the NotificationProvider. Any lower level component can then use the *useNotifyError* or *useNotifySuccess* hooks to display messages that will pop up and persist for a couple of seconds even if the component who generated the notification's gets discarded after changing views. We found the Snackbar component of the Material UI library as the perfect way to display these responsive notifications. Material UI will be discussed further in Section 3.1.4.

ContractList Component

In several places on our web app we show a list of contracts. For example when the developer might look at available contracts, the company might look at a list of open contracts, or some user might look at their own profile and want to see list of their own contracts. The contract list component therefore had to be very reusable. We primarliy used props in order to control the behaviour of the list. For example a 'condensed' prop to render the list in a table format instead of a large card format with media. We also pass a 'status' prop to determine what kind of contracts to be shown. The *ContractList* component is in charge of loading the data (via API calls) and displaying the data. We only realised near the end of the project that we could leverage Context providers and custom hooks to split these concerns up into components that will follow the MVC design pattern more closely. In the future we will be using this design pattern extensively, but for now the *ContractList* component worked well enough and we did not want to break any changes.

3.1.4. Material UI

The React Material UI (MUI) library provides numerous, high quality user-interface components. We used the MUI library as an extensive crutch in our development as it saved countless hours of work which could have been lost to basic CSS coding. In addition, due to MUI's flexibility and easy customisability, we were able to create a beautiful interface for our client side.

3.1.5. Axios

For API calls from the frontend we used the Axios library. It allowed us to establish promise-based HTTP requests; enabling an even more digestible client experience. In addition, the library provides a way to automatically convert JavaScript objects into JSON format, making coding API calls on the frontend a breeze.

3.1.6. ImgBB Image Hosting and API

One of the requirements for the project was that images (used for developer and company avatars) should not be stored directly in the database. We realise that a Content Management System (CMS) could be a very elegant solution to this requirement. Web applications that make use of a CMS can very easily be scaled for large amounts of traffic. However we decided to go with a much simpler approach. There are several image hosting websites such as ImgBB and Imgur that allow anyone with an account to upload images via the service's API. We decided to go with ImgBB's API which is very simple to use with minimal authentication requirements.

ImgBB's API has a POST request at the endpoint https://api.imgbb.com/1/upload. You have to pass it two parameters, the API key associated with the account that will be storing the image and the image (encoded in base64 format) that you want to update. If the request is a success the response includes a URL to where the image is stored. We have a resuable *AvatarPicker* component, that allows the user to upload an image from their filesystem. This image is encoded to base64 and uploaded via the above API. On sign up or profile edit, we then only send the URL to the database.

On a side note, we realise that we forgot to hide the API token associated with our ImgBB account and to not publish it to GitLAB. In the future we will take care to use environment variables in both development and production instead of exposing our API key. You live and you learn.

3.2. Backend Solution

When designing an API, certain constraints need to be in place to prevent it from turning too messy. The most common set of architecture constraints is REST or a RESTful API. REST APIs provide a flexible, lightweight technique to integrate applications; it has emerged as the most widely used method for connecting components in microservice architectures [3]. The key operations of a RESTful API are 4 basic HTTP methods namely GET, POST, DELETE and PUT. These HTTP methods are directly related to CRUD (Create, Read, Update, Delete) operations which are performed on the database. A high-level example would be if the client request to add a user to the database. The client would send a request to the API of method POST along with a JSON object that contains the new user's details. The API will then determine which method is being used and consequently apply the create operation on the database to add the user to the user table. In Figure 3.3 below we can see this process and how CRUD and HTTP methods relate.

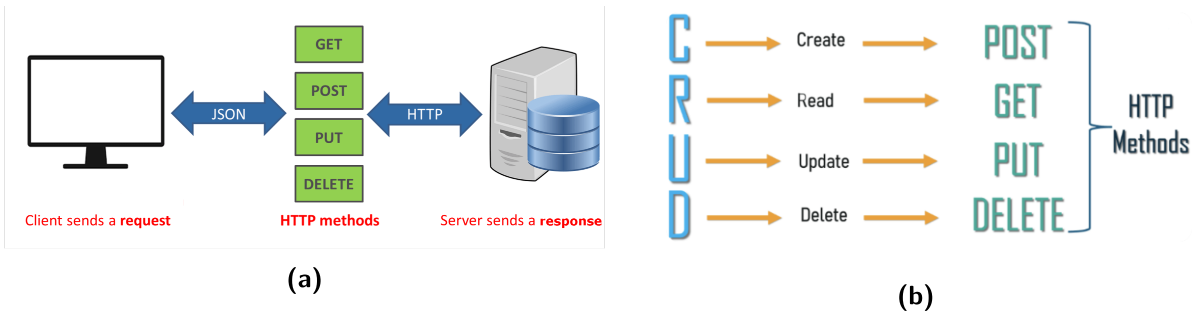
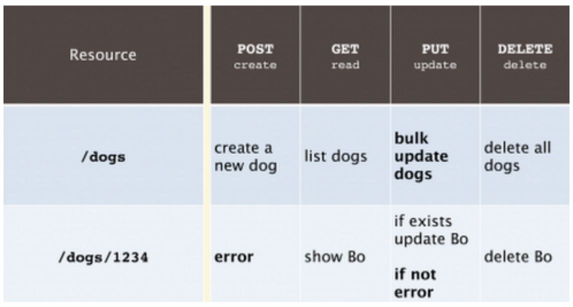


Figure 3.3: (a) HTTP methods [3] (b) CRUD operations [4]

To keep our API routes easy to read and interpret, we used our own standard. To work with an object one first specifies the class, followed by an unique identifier for that class. For example, to request a developer one would use *baseUrl/developer/nicolvisser*. We also decided that verbs shall not be used in the routes as this increases the number of unique routes and makes it harder to implement a RESTful system. With a single route having the ability to perform up to 4 different functions on the same data, we are able to greatly reduce the number of routes and unnecessary code. Figure 3.4 shows how our API would be used. For a full list of routes refer to the link in the Postman section 2.3.2.



Resource	POST create	GET read	PUT update	DELETE delete
/dogs	create a new dog	list dogs	**bulk update dogs**	delete all dogs
/dogs/1234	**error**	show Bo	if exists update Bo **if not error**	delete Bo

Figure 3.4: RESTful API implementation [1]

3.2.1. Technology

Object Relational Mapping (ORM)

Since python is an object oriented language, it only made sense to treat the data in the database as objects as well. A thorough description was given by Mia Liang: "An object-relational mapper provides an object-oriented layer between relational databases and object-oriented programming languages without having to write SQL queries. It standardizes interfaces reducing boilerplate and speeding development time" [5]. Flask has a great ORM library SQLAlchemy which allowed for a flawless implementation of an ORM in our backend.

Password Hashing

Ensuring the safety of our user's data requires their passwords to be protected in some way. Traditional encryption is not safe enough these days and a more secure method is required: hashing. Hashing of a password refers to essentially transforming a string (password) into a random string of characters. The problem is that every string has its own unique hash code, therefore rainbow tables can be used to crack a password using previously known string-hash code pairs. To reduce this, one can add salt to the password string, changing the string and hence the hash. The salt must also be stored which could be a problem as ideally every password should have its own salt. Hashing algorithms (such as the SHA family) have become very efficient and GPUs can crack them fairly easily, therefore, for password hashing, one wants a "slower" algorithm. Now considering that we want a fairly slow and salt friendly hashing algorithm we chose the Argon2 algorithm (python version) as it stores its own salts and salts multiple times.

Bibliography

[1] Brian Mulloy, "Restful api design: nouns are good, verbs are bad," 2011. [Online]. Available: https://cloud.google.com/blog/products/api-management/restful-api-design-nouns-are-good-verbs-are-bad

[2] Andrew Luca, "Private route in react-router v6," 2021. [Online]. Available: https://dev.to/iamandrewluca/private-route-in-react-router-v6-lg5

[3] Amit Kulkarni, "Ultimate crud vs rest guide: Operations simplified 101," 2021. [Online]. Available: https://hevodata.com/learn/crud-vs-rest/#ir

[4] Avelon Pang, "Crud operations explained," 2021. [Online]. Available: https://medium.com/geekculture/crud-operations-explained-2a44096e9c88

[5] Mia Liang , "Understanding object-relational mapping: Pros, cons, and types." [Online]. Available: https://www.altexsoft.com/blog/object-relational-mapping/

Appendix A

Appendix

A.1. Work done by each group member

Name	Student Number	Work done
Nicol Visser	16986431	Client-side solution, Report
Phillip Schommarz	22621105	Client-side solution, Demo Video, Report
Lizé Steyn	21772037	Client-side solution, Database Solution, Report
JC Mouton	22549889	Back-end API, Database Solution, Report

Table A.1: Work done by each group member.