Eötvös Loránd University

Faculty of Informatics Department of Media and Educational Informatics

Tech Stack Planner

Supervisor:

Illés Zoltán
Associate Professor - PhD, Habilitation

Author:

Ali Bayramli
Computer Science BSC

Budapest, 2021

EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:
Student's Name: Bayramli Ali
Student's Neptun code: BOWXDM

Course Data:
Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Zoltán Illés*
Supervisor's Home Institution: *Eötvös Loránd University, Budapest*
Address of Supervisor's Home Institution: *1117 Budapest, Pázmány Péter sétány 1 / C*
Supervisor's Position and Degree: *Associate Professor, PhD, habilitation*

Thesis Title: Finding technology stack for startups

Topic of the Thesis:
(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Choosing the right technology stack can be overwhelming for any startup in the first run. One can find it very hard to choose the right pack of programming languages and tools used in software to ensure that the best service is delivered. If the team ends up settling for a wrong tech stack, it can have a devastating impact on the company as well as for the customers.
The program to be developed in the thesis will provide broad insight on which tools to choose for the project. It will ask the user several factors such as the size of the team, the location where the startup is based, and the money invested for the projects. Users will also have the option to say how fast they want to deploy their apps. Besides, using data analytics tools, the program will analyze the tech stack of its competitors as well. Based on the given criteria, the application will display the right set of tech stacks. The program will also recommend particular tools that are beneficial and have community support.
The application aims to serve startups based on web and mobile development. Users will be able to create an account, and their credentials will be kept in a database.

Budapest, 2020.05.30.

Contents

Chapter 1

Introduction

1.1 Background

In the modern area of software development, there are dozens of tech stacks on the market. A technology stack is a set of programming tools that are used by the developers to create a digital product. Thanks to the rich ecosystem, startups can scale their applications faster than before and thus reach MVP (minimum viable product) much more easily.

1.2 Motivation

As promising as software development nowadays as it sounds, there are still several factors to consider when choosing the right set of tech stacks and many startups fail to make a proper choice in the early stages of software development. A wrong decision could lead to setting the startup back by months and eventually off the market, while a well-chosen tech stack gives a robust advantage over its competitors as well as helps the startup to scale faster.

1.3 Solution to the problem

The program created aims to solve problems that are mentioned in Section 1.2. Tech Stack Planner is a web application and I decided to build it on a web platform thanks to its vast community support and being platform-independent. The Application helps startups to accelerate their products by suggesting the right set of technology stacks that are best fit to use and have excellent community support. It also gathers numerous information from the startup's owner, such as the field of the startup and the budget invested for hiring developers at the initial stage. Based on the details, startup owners will be able to pick the desired tools and start using those technologies to make awesome products/features safely and of course, rapidly.

Chapter 2

User Documentation

This chapter provides a brief description of the application, system requirements, and installation guides to run the application in the local environment.

2.1 System Requirements

To run the application locally, some system requirements have to be satisfied.

2.1.1 Hardware

Devices must have access to the internet with a speed of minimum 50Kbps. In terms of hardware specifics, it is recommended to use a device that has at least 4GB of RAM and 1.5 GHz processor speed. Since the application is written on the web, it is operating system agnostic and thus it is possible to use any operating system (Recommended: macOS 10.8 or later, Windows 7 or later, Linux 64 bit) to run. For better performance and user experience, using Chrome or Mozilla is highly recommended.

2.1.2 Software

Since the application is written in VueJS on the frontend [1], the Vue version of at least 3.0.0 with Vite Compiler (at least 2.0.0) needs to be installed [2]. Users also need to have Node js installed in their local machines. It is highly recommended to use the Node version of at least 12.0.0 [3] along with the npm version of at least 6.0.0 [4]. For database operations, the Redis version of at least 6.2 needs to be installed [5]. The application uses MongoDB as well [6], however, since MongoDB is configured on the cloud by using MongoDB Atlas, there is no need to install it locally [7]. Finally, users have to install git for version control [8].

2.2 Installation Process

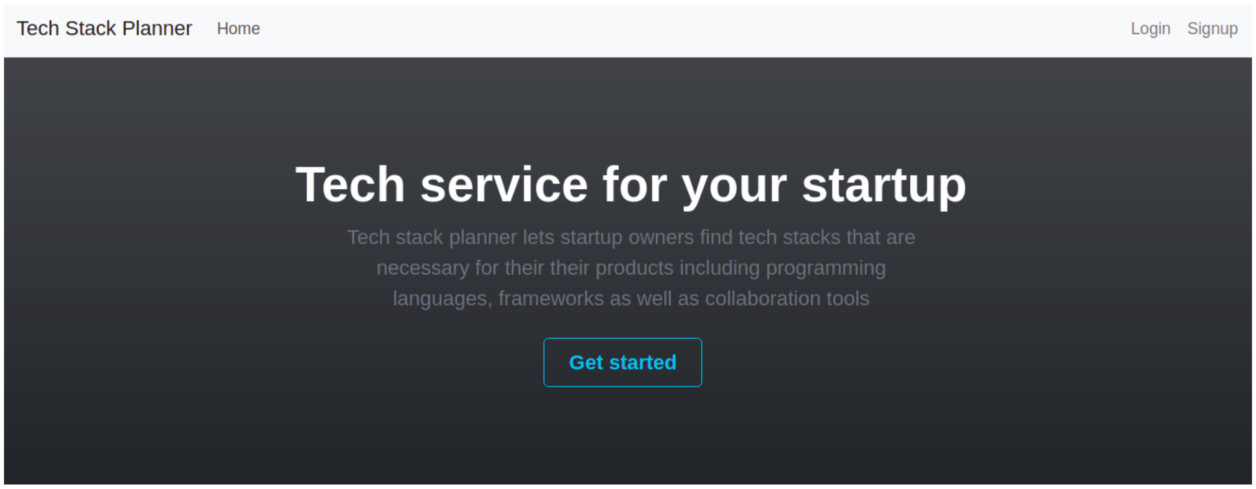
Once the hardware and the software requirements are met, users can clone the project hosted on GitHub [9]. The next step is to clone the project to the local machine by typing:
git clone <repository-url> in the terminal.
After cloning, all the package dependencies should be installed by typing *npm install*. This needs to be done in **both front-end and back-end** directories. Also, users have to set their environment variables by creating a *.env* file in the *backend/src* directory. The environment variables are needed for MongoDB Atlas usage as well as the authorization secret keys.
After navigating to directories via terminal, users have to type *npm run dev* for each one of the directories. If everything is settled properly, the web page will be available on
http://localhost:3000/

2.3 User Interfaces

In this chapter, the user interfaces will be visualized by using figures. Sections from 2.3.1 to 2.3.3 contain publicly accessible interfaces while for the rest of the sections, user authentication is required.

2.3.1 Home Page

After starting the application, the home page will be rendered by default.


Tech Stack Planner
Home
Login
Signup

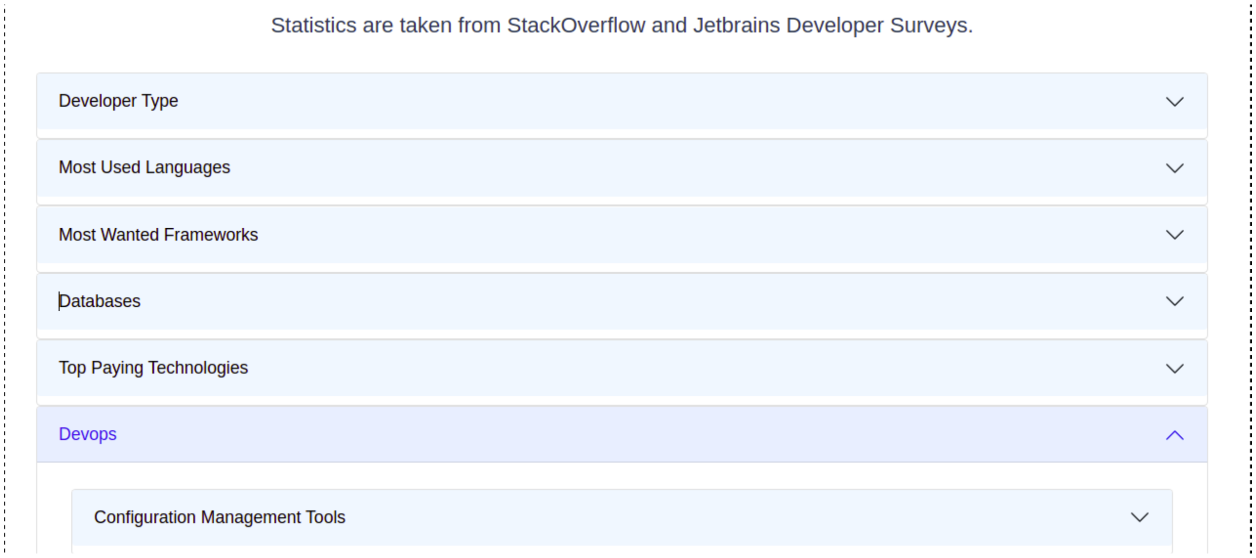
Tech service for your startup

Tech stack planner lets startup owners find tech stacks that are necessary for their their products including programming languages, frameworks as well as collaboration tools

Get started

Figure 2.3.1.1: Hero section of the home page

Under the hero section, users can get general information about popular tech tools by following details inside the statistics accordion. The statistics are taken from the StackOverflow [10] and JetBrains Developer Surveys of 2020 [11].


Statistics are taken from StackOverflow and Jetbrains Developer Surveys.

Developer Type
Most Used Languages
Most Wanted Frameworks
Databases
Top Paying Technologies
Devops
Configuration Management Tools

Figure 2.3.1.2: Statistics accordions of the home page

Statistics include two separate sections: general and team statistics. General statistics hold information of a particular programming language/framework/database, while team statistics hold tools that are widely used across teams.

General statistics examples: *Developer Type, Most Used languages, Most Wanted Frameworks, Databases, and Top Paying Technologies*.
Team statistics examples: *DevOps, Microservices, Team Tools, and Testing* Team statistics hold a nested accordion structure since they include different types based on the header (e.g DevOps). Users can observe this by clicking either one of the team statistics and the accordion will be expanded to list its sub-accordions.

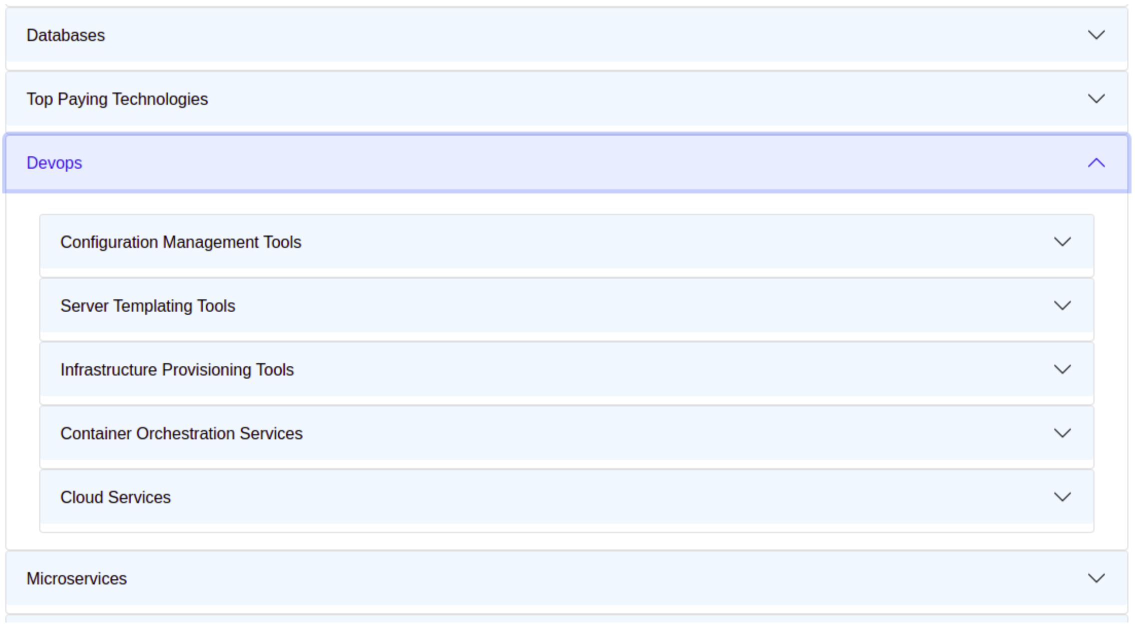
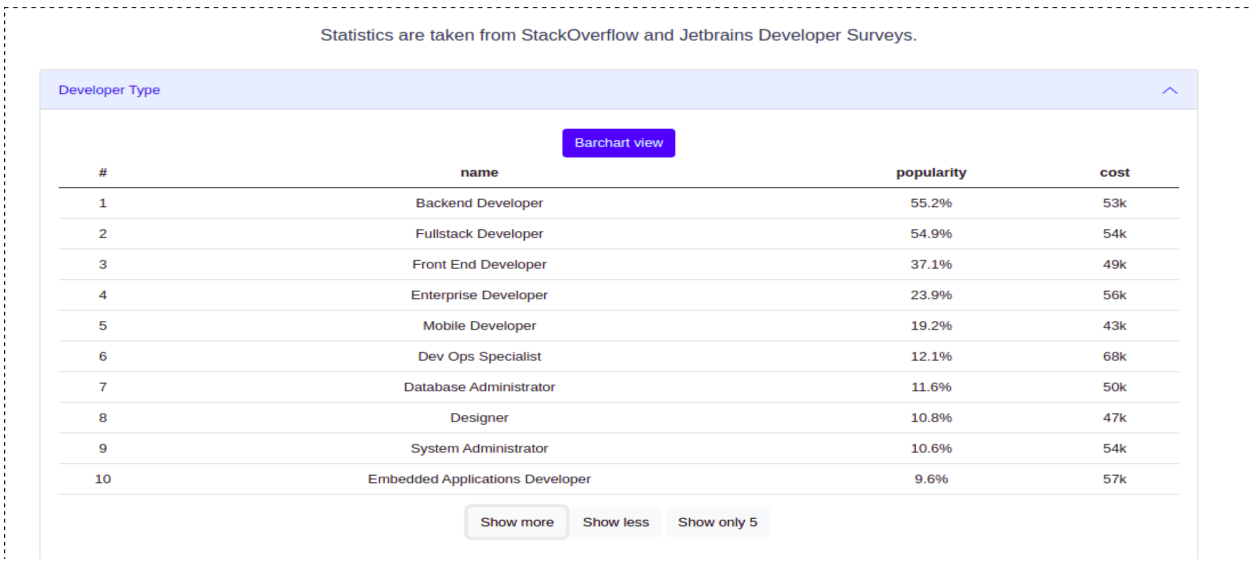

Databases
Top Paying Technologies
Devops
Configuration Management Tools
Server Templating Tools
Infrastructure Provisioning Tools
Container Orchestration Services
Cloud Services
Microservices

Figure 2.3.1.3: Nested accordion structure

Two options will be presented to the users when viewing the statistics details. Users can either see the details by the table view or by bar chart if they prefer a graphical view.



Statistics are taken from StackOverflow and Jetbrains Developer Surveys.

Developer Type

Barchart view

#	name	popularity	cost
1	Backend Developer	55.2%	53k
2	Fullstack Developer	54.9%	54k
3	Front End Developer	37.1%	49k
4	Enterprise Developer	23.9%	56k
5	Mobile Developer	19.2%	43k
6	Dev Ops Specialist	12.1%	68k
7	Database Administrator	11.6%	50k
8	Designer	10.8%	47k
9	System Administrator	10.6%	54k
10	Embedded Applications Developer	9.6%	57k

Show more Show less Show only 5

Figure 2.3.1.4: Accordion contents appearance

When table view is preferred, users can load more/less content by clicking the appropriate buttons on the bottom of the table. When the bar chart view is presented, a new page will be rendered with bar chart details of the content that the user has selected. **Figure 2.3.1.5** shows an example bar chart view when a user clicks to get information about developer-type statistics. To see specific data presented by the bar chart, users also can filter values by the labels (e.g popularity, cost) by clicking the appropriate label button.

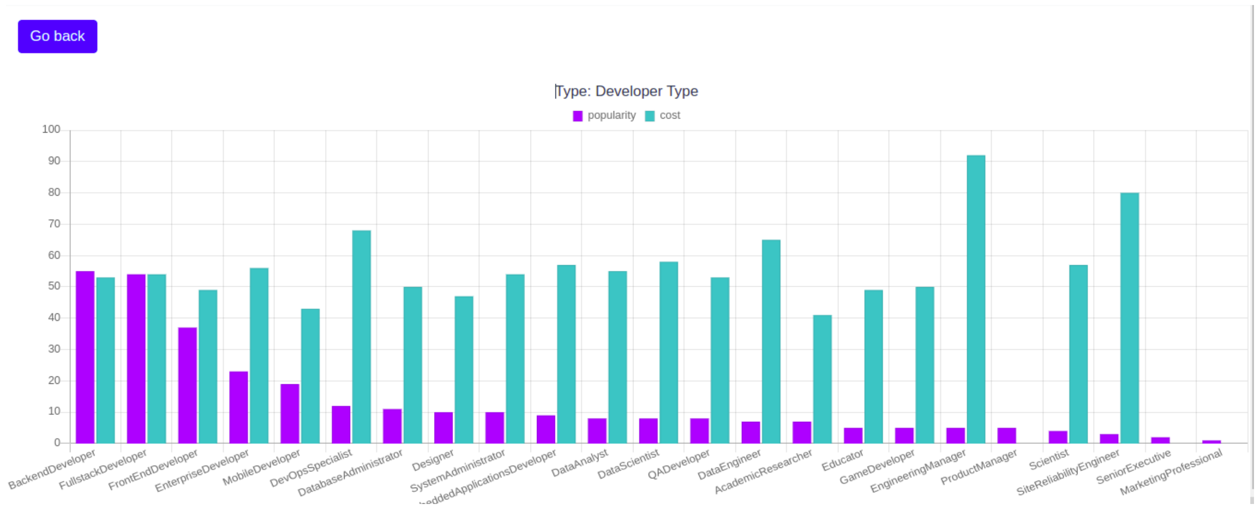
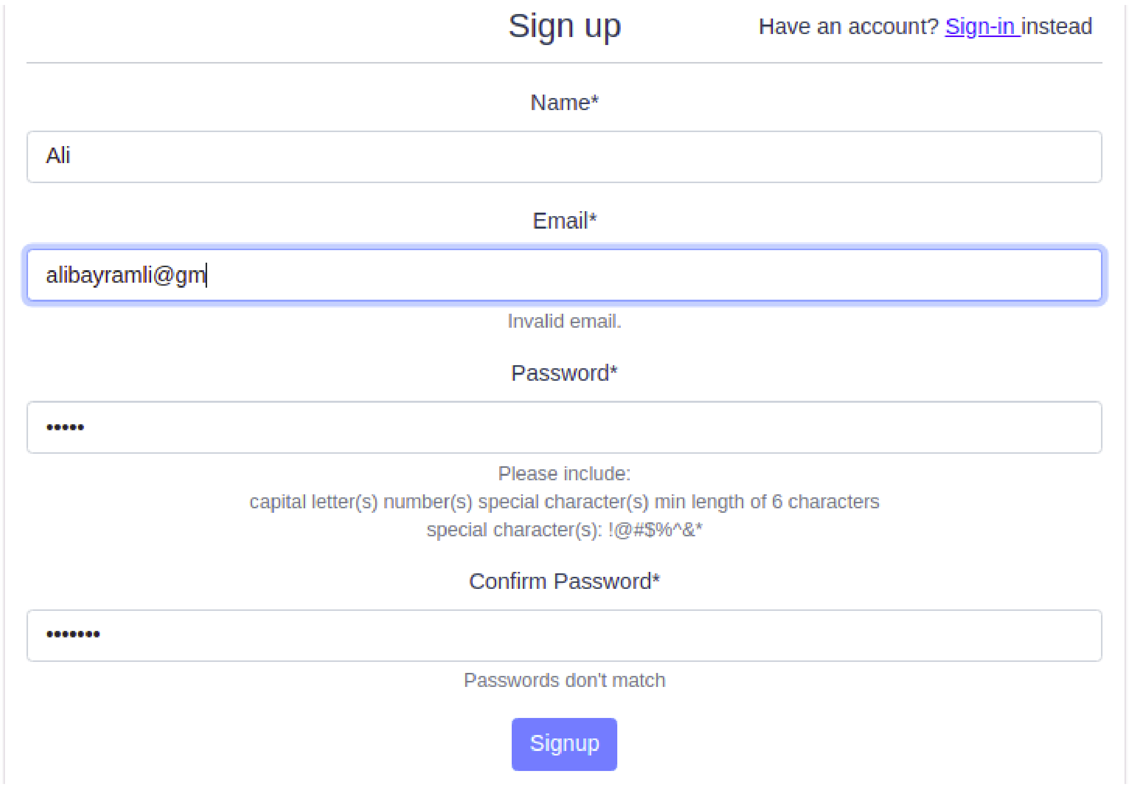


Figure 2.3.1.5: Barchart view of developer type statistics

2.3.2 Signup Page

To create a tech stack form and view suggested tools, users have to be logged in.
When users are accessing the web page for the first time, they must create an account either by clicking the signup link or navigating to the URL: *http://localhost:3000/signup*
Figure 2.3.2.1 demonstrates how the sign-up process is handled.



Sign up

Have an account? Sign-in instead

Name*

Ali

Email*

alibayramli@gm

Invalid email.

Password*

•••••

Please include:
capital letter(s) number(s) special character(s) min length of 6 characters
special character(s): !@#$%^&*

Confirm Password*

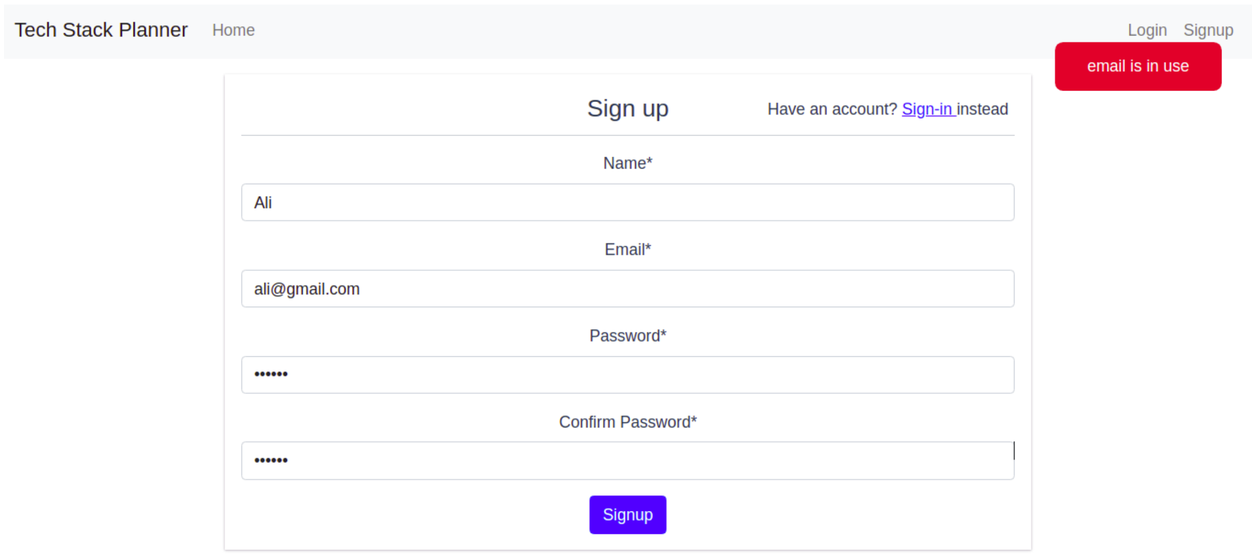
•••••••

Passwords don't match

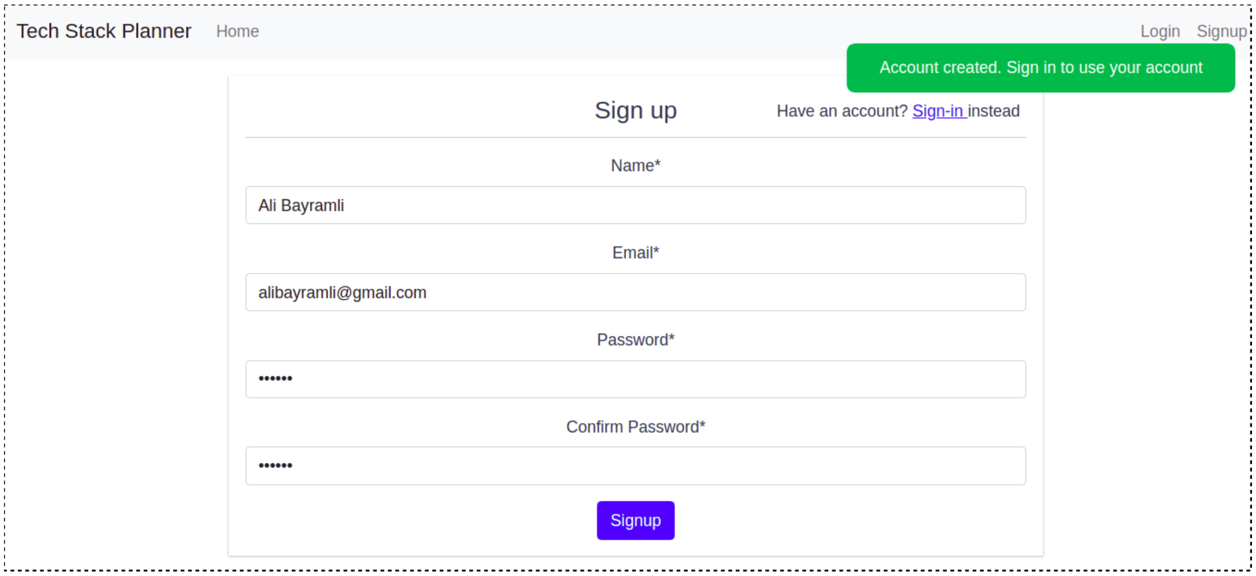
Figure 2.3.2.1: Signup process

To successfully create an account, users must fill in all the steps correctly. When adding the email account, the account must follow standard email patterns. Password should contain at least 6 characters, include a minimum of one number, one upper case letter, one lower case letter, and a minimum of one special letter from the followings: `!@#$%^&*`

If any of the requirements are not met, the signup button will stay disabled. When all the requirements are satisfied and the signup button is clicked, a toast notification will appear indicating the status of the registration. The red color in the notifications indicates errors (Figure 2.3.2-a) occurred in the request, while the green color indicates a successful request (Figure 2.3.2-b).



a) Signup failed



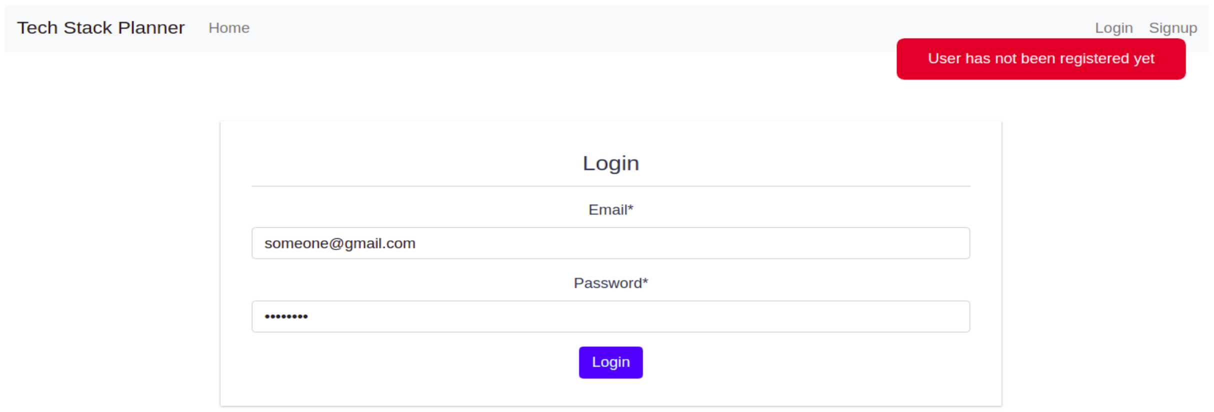
b) Signup successful

Figure 2.3.2: Signup toast notifications

In case of successful signup, the web page will be redirected to the login page after 1 second.

2.3.3 Login Page

Once the user account is created, users can log in to the application.
The login process is similar to the signup process, except users just need to provide correct email and password credentials. When the login button is clicked, a toast notification will appear indicating the status of the login process. The server will respond with an error status, if an email is not registered before logging in (Figure 2.3.3-a), or the user provides incorrect password credentials (Figure 2.3.3-b).

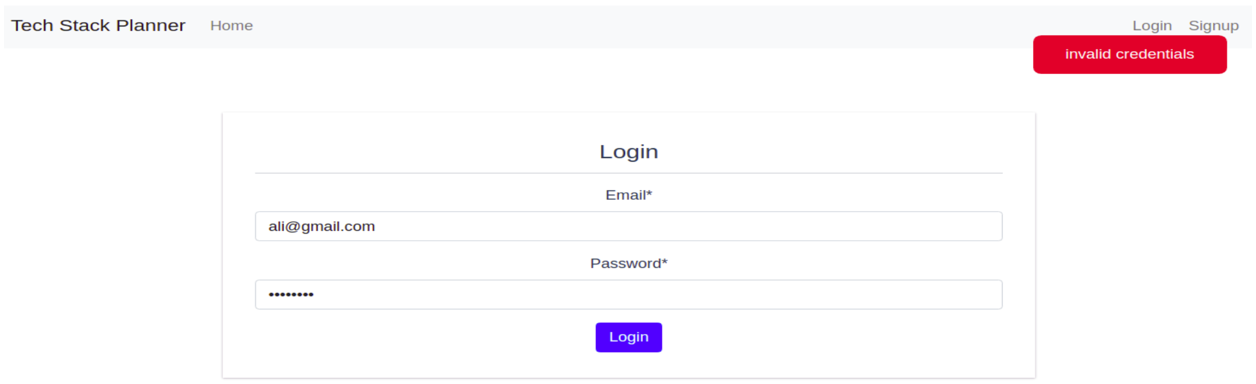


User has not been registered yet

someone@gmail.com

••••••••

a) User not found



invalid credentials

ali@gmail.com

••••••••

b) Login invalid

Figure 2.3.3: Login toast notifications

In case of a successful login, the web page will be redirected to the home page after 1 second.

2.3.4 Logout Page

Once users are logged in to the application, they will be able to log out from the system anytime by clicking on the logout link. When clicked, a pop-up will appear making sure that the user is indeed intended to log out.

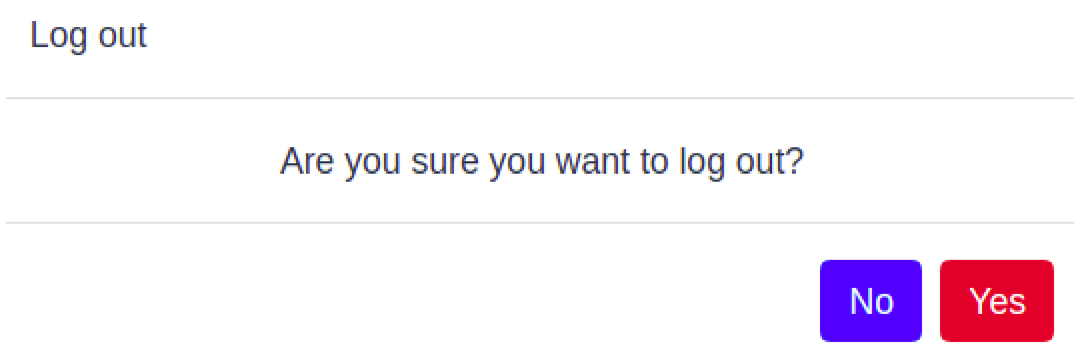
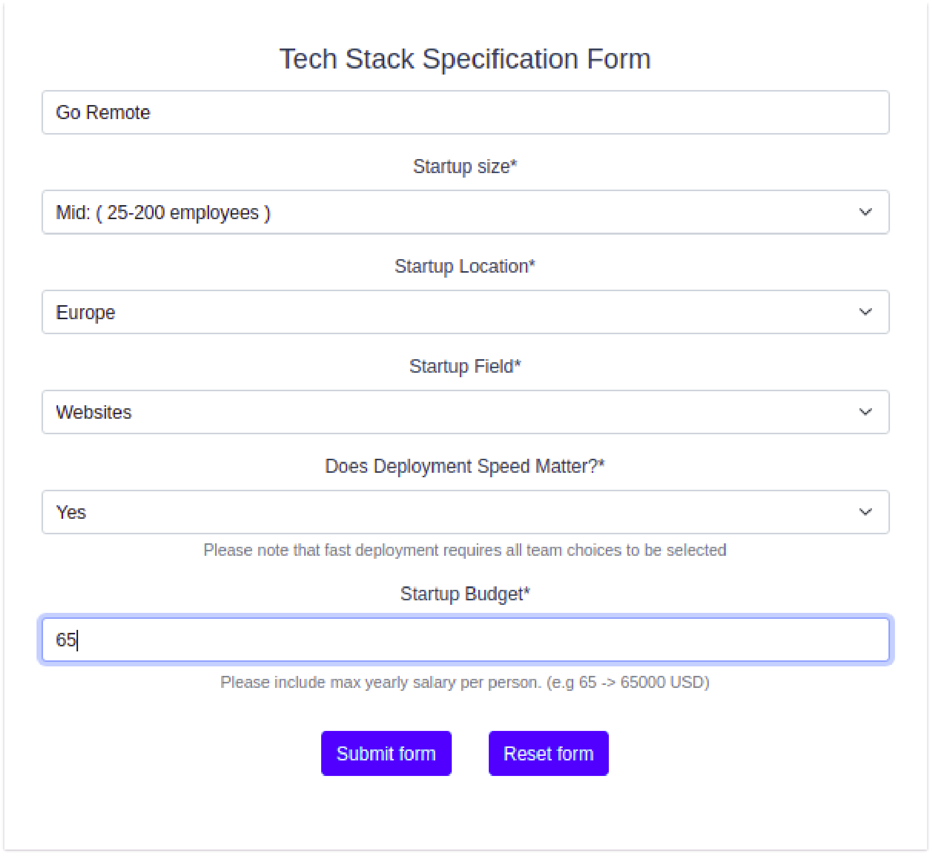


Figure 2.3.4.1: Logout

When the user clicks "Yes", the toast notification will appear indicating that the user is logged out and the page will redirect to home after one second.

2.3.5 Form Page

Logged-in users can start filling in form options to get desired tools for their startups.
Figure 2.3.5.1 demonstrates an example form interface when owners fill in their startup details.



Tech Stack Specification Form

Go Remote

Startup size*

Mid: (25-200 employees)

Startup Location*

Europe

Startup Field*

Websites

Does Deployment Speed Matter?*

Yes

Please note that fast deployment requires all team choices to be selected

Startup Budget*

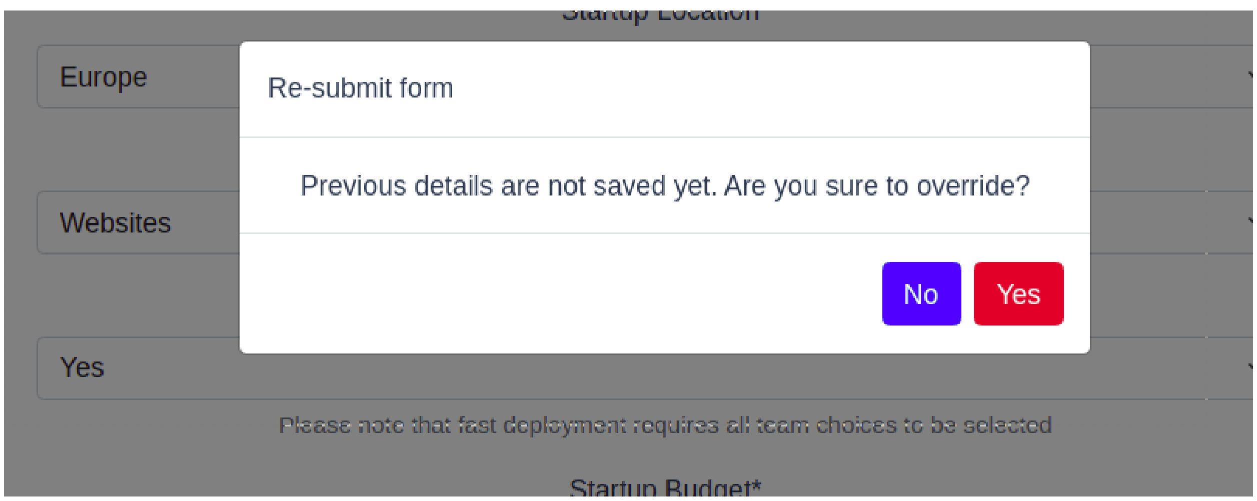
65

Please include max yearly salary per person. (e.g 65 -> 65000 USD)

Figure 2.3.5.1: Tech stack form filling

A user can submit the form when all the form options are filled in. If the user has a mid or large startup, the system makes the deployment speed matter to any startup, otherwise, it is optional. This is because bigger startups usually involve more employees to be involved and the proper DevOps configuration comes very handy when scaling [12].

When adding a startup budget, the system will only accept valid numbers (integer or floating-point), otherwise invalid symbols will be wiped out from the input (e.g including two dots, starting with two zeros, etc.). If the user already submitted their form and wants to resubmit (Figure 2.3.5.2-a) or reset (Figure 2.3.5.2-b), a proper warning modal will pop up.


Startup Location

Europe

Websites

Yes

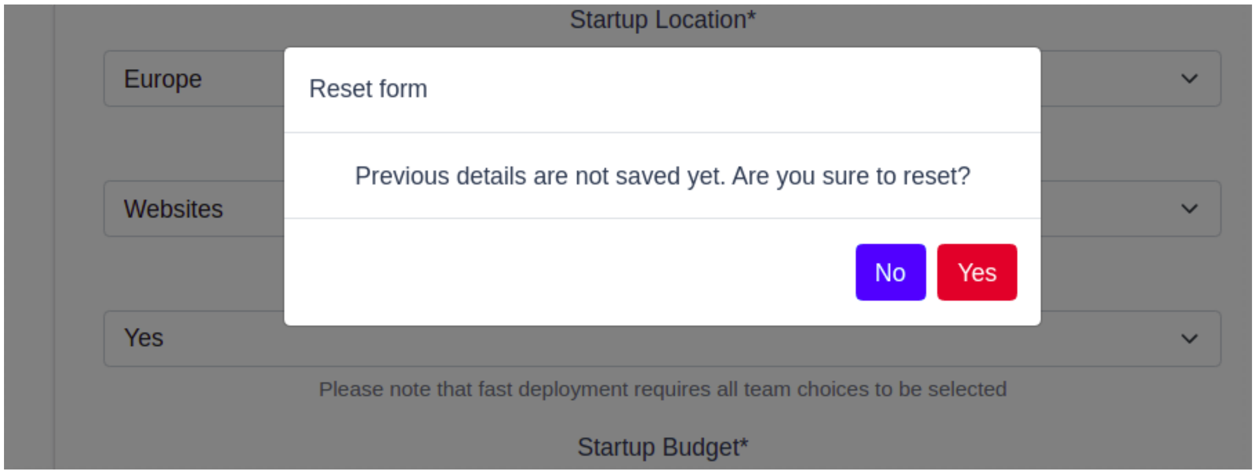
Re-submit form

Previous details are not saved yet. Are you sure to override?

Please note that fast deployment requires all team choices to be selected

Startup Budget*

a) Resubmitting the form


Startup Location*

Europe

Websites

Yes

Reset form

Previous details are not saved yet. Are you sure to reset?

Please note that fast deployment requires all team choices to be selected

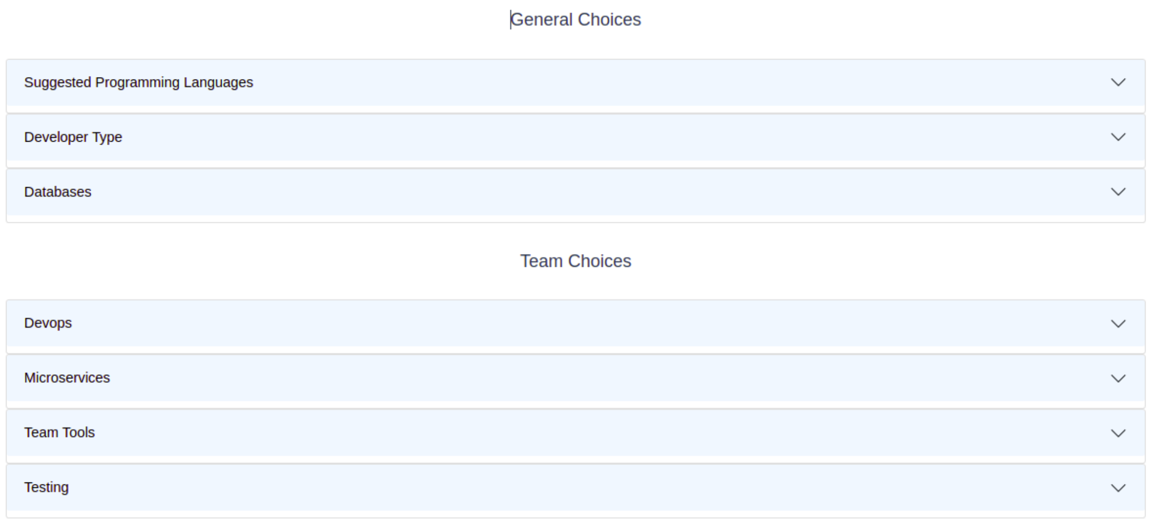
Startup Budget*

b) Resetting the form

Figure 2.3.5.2: Warning pop-ups

2.3.6 Tech Stack Page

Once the form is submitted, the application will be redirected to the tech stack page.
The page consists of two separate choices, one general and the other is team-related choices.


General Choices

Suggested Programming Languages

Developer Type

Databases

Team Choices

Devops

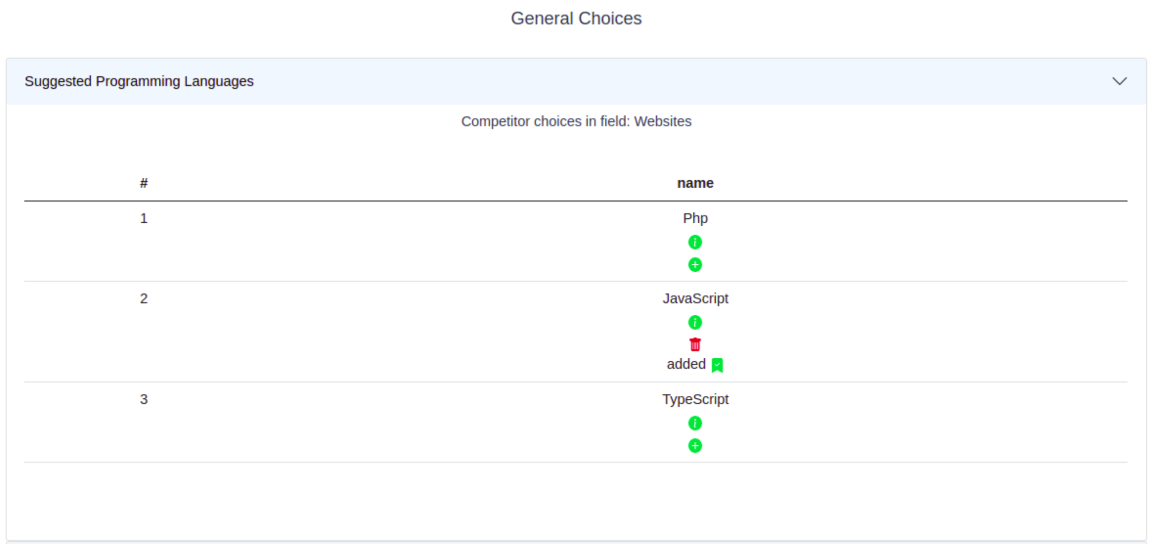
Microservices

Team Tools

Testing

Figure 2.3.6.1: Tech stack page overview

The tools in general choices are different than on the home page since the choices are more personalized based on the startup details. The first general choice is the programming languages that are used by the competitor fields. The user can pick one or more tools, or delete them if they decide not to pick anymore.


General Choices

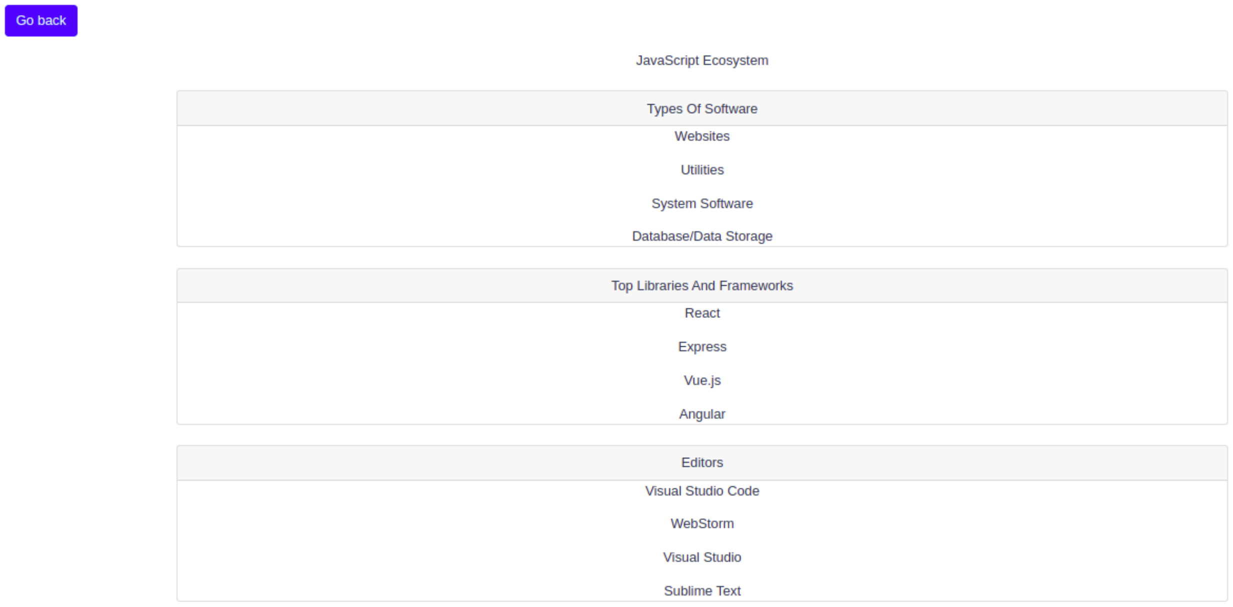
Suggested Programming Languages

Competitor choices in field: Websites

#	name
1	Php
2	JavaScript added
3	TypeScript

Figure 2.3.6.2: Suggested programming languages

To view more broad information about the programming language and its ecosystem, users should click the information icon and the system will be redirected to a new page.


Go back

JavaScript Ecosystem

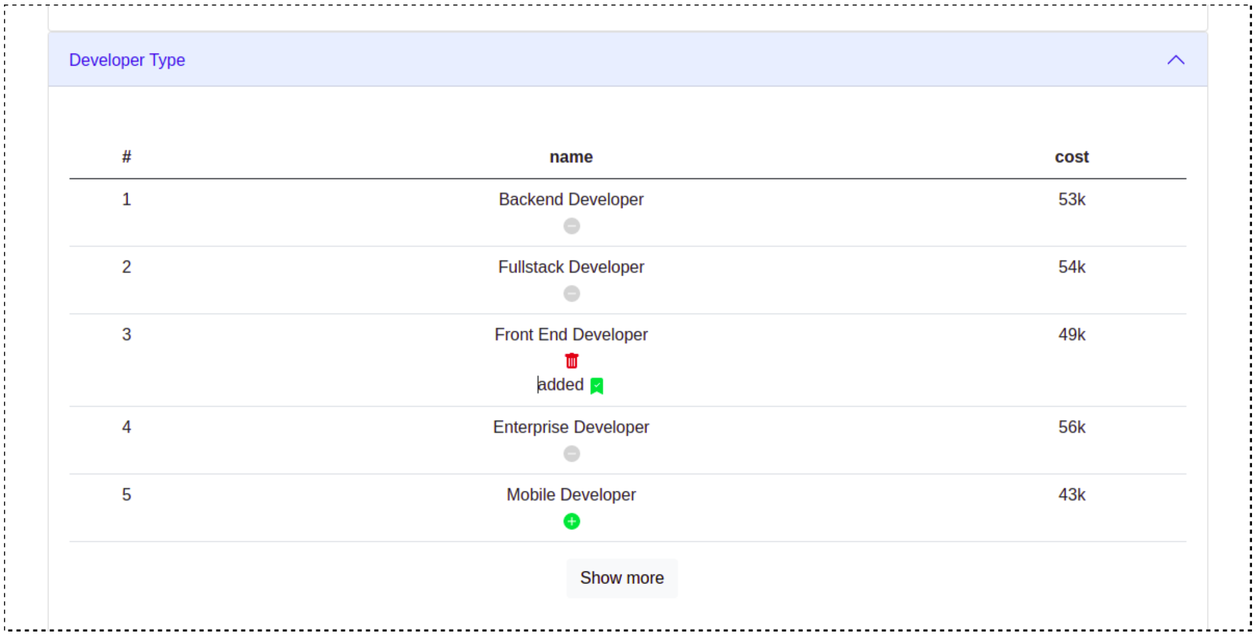
Types Of Software
Websites
Utilities
System Software
Database/Data Storage

Top Libraries And Frameworks
React
Express
Vue.js
Angular

Editors
Visual Studio Code
WebStorm
Visual Studio
Sublime Text

Figure 2.3.6.3: Programming language ecosystem

Users can not choose specific developers if their startup budget is set below the market average salary for the specific developer position.

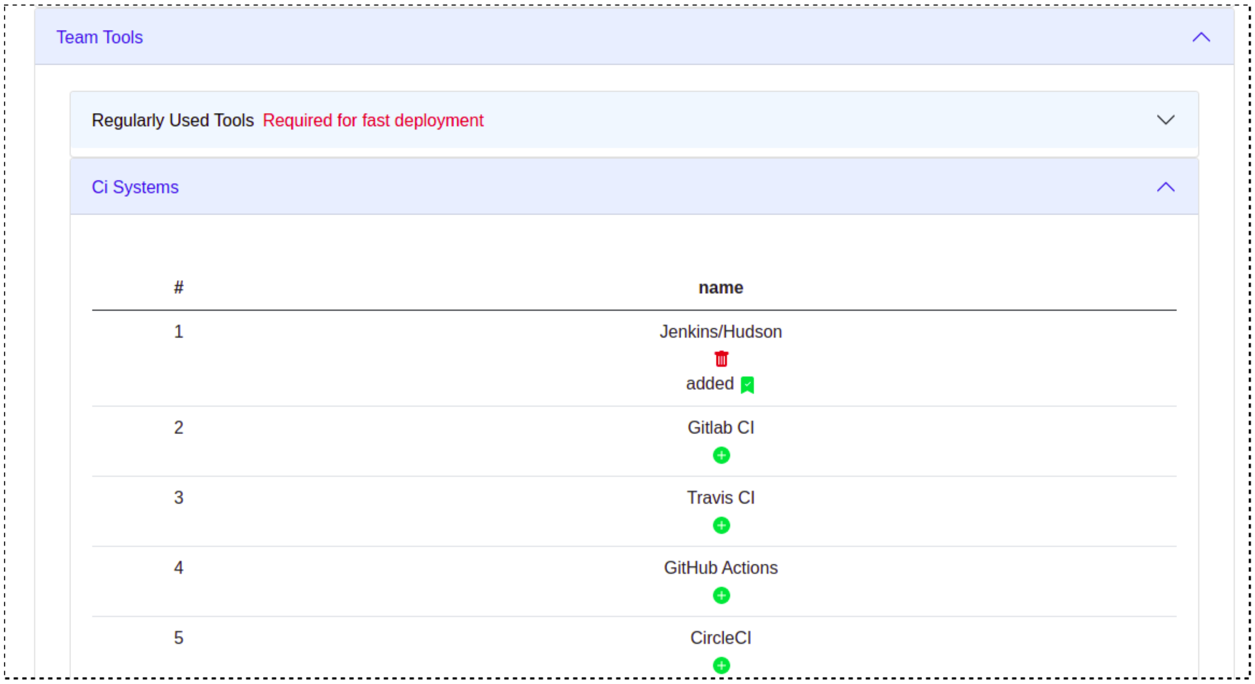

Developer Type

#	name	cost
1	Backend Developer	53k
2	Fullstack Developer	54k
3	Front End Developer added	49k
4	Enterprise Developer	56k
5	Mobile Developer	43k

Show more

Figure 2.3.6.4: Developer types

For team-related choices, it is optional for users to pick the tools unless the deployment speed matters. In that case, every sub-accordion must have at least one tool picked, otherwise a warning text will be displayed at the top of the accordion header.



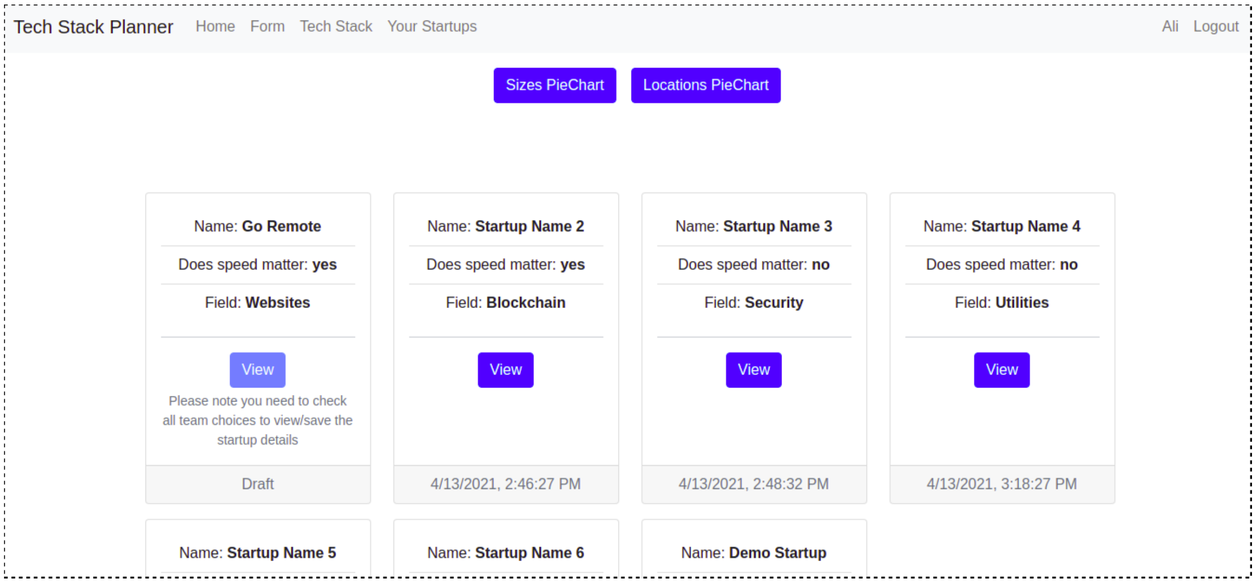
#	name
1	Jenkins/Hudson added
2	Gitlab CI
3	Travis CI
4	GitHub Actions
5	CircleCI

Figure 2.3.6.5: Compulsory team tools when deployment speed matters

2.3.7 User Startups Page

Figure 2.3.7.1 demonstrates how the actual user startups page looks like. All startups created by the specific user will be shown in cards. The cards will display the name, field, and deployment speed of the startup. Once the form is submitted, a draft startup[1] card will be generated on the user startups page. If the deployment speed matters and not all team tools are selected according to the rules (Figure 2.3.6.6), the view button will stay disabled.
In case the startup is drafted, the word "Draft" will appear on the bottom of the card, otherwise, the creation date will be displayed.

[1] Draft Startup refers to the startup that has not been saved in the database yet.


Tech Stack Planner Home Form Tech Stack Your Startups Ali Logout

Sizes PieChart Locations PieChart

Name: **Go Remote**
Does speed matter: **yes**
Field: **Websites**
View
Please note you need to check all team choices to view/save the startup details
Draft

Name: **Startup Name 2**
Does speed matter: **yes**
Field: **Blockchain**
View
4/13/2021, 2:46:27 PM

Name: **Startup Name 3**
Does speed matter: **no**
Field: **Security**
View
4/13/2021, 2:48:32 PM

Name: **Startup Name 4**
Does speed matter: **no**
Field: **Utilities**
View
4/13/2021, 3:18:27 PM

Name: **Startup Name 5**

Name: **Startup Name 6**

Name: **Demo Startup**

Figure 2.3.7.1: User startups

If there is no startup created by the user, an empty cart icon will be displayed on the user startups page.

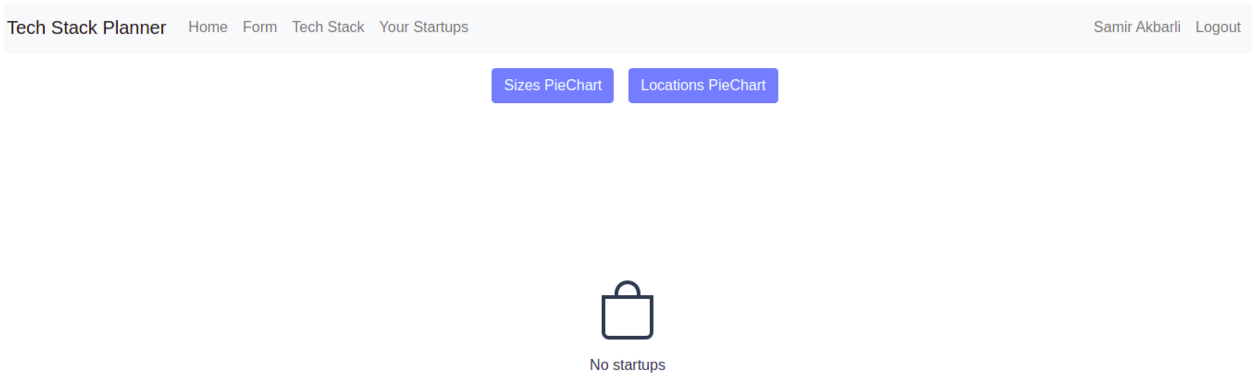


Figure 2.3.7.2: No startups created

Two buttons on the top will display startup sizes and the locations created by the user in a pie chart/accordion format. Once clicked, piecharts will be shown on a new page.

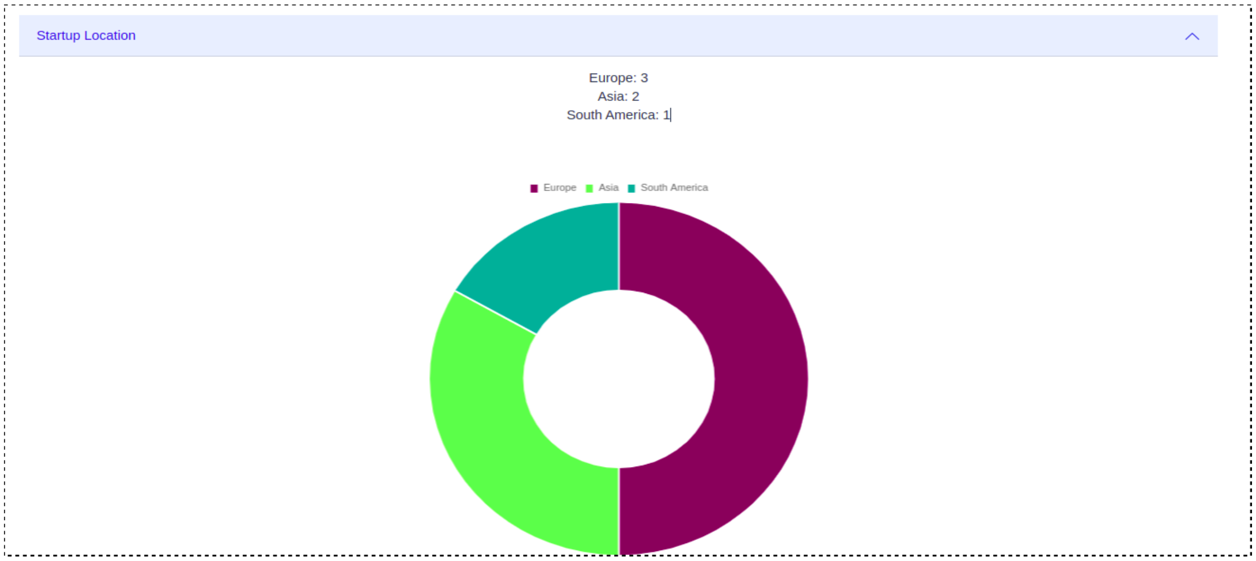
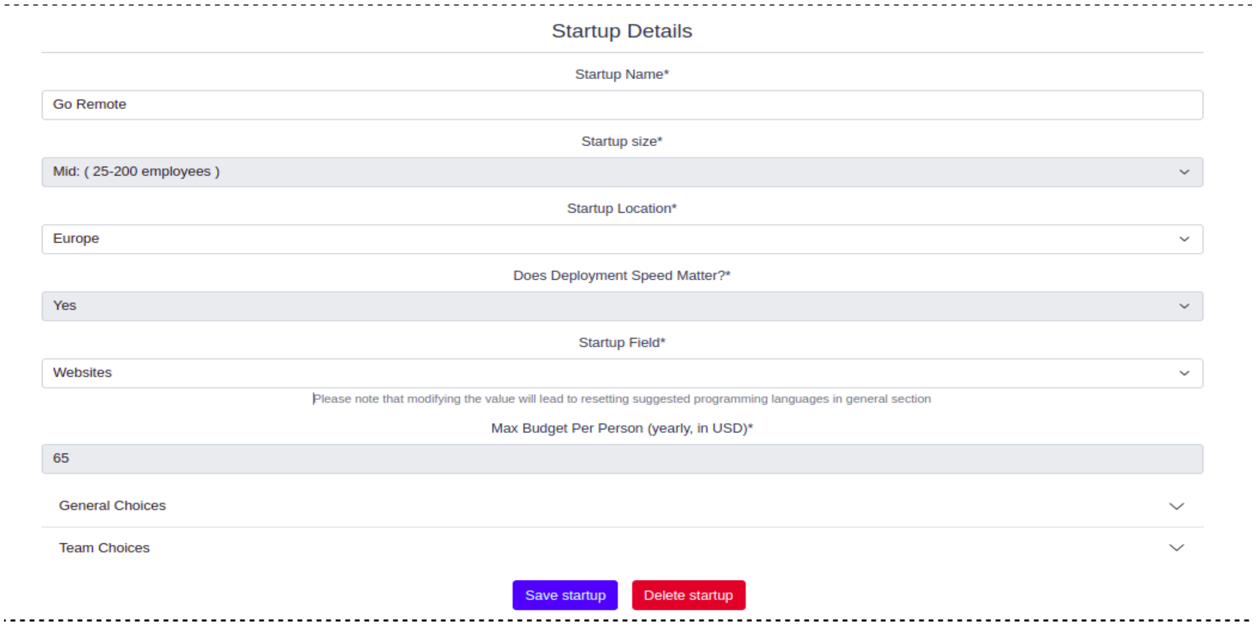


Figure 2.3.7.3: Startups by locations

2.3.8 Startup Details Page

Startups that are draft or already saved (Figure 2.3.7.1) can be viewed by opening the proper startup card and the following figure illustrates contents when a card is opened.



Startup Details

Startup Name*
Go Remote

Startup size*
Mid: (25-200 employees)

Startup Location*
Europe

Does Deployment Speed Matter?*
Yes

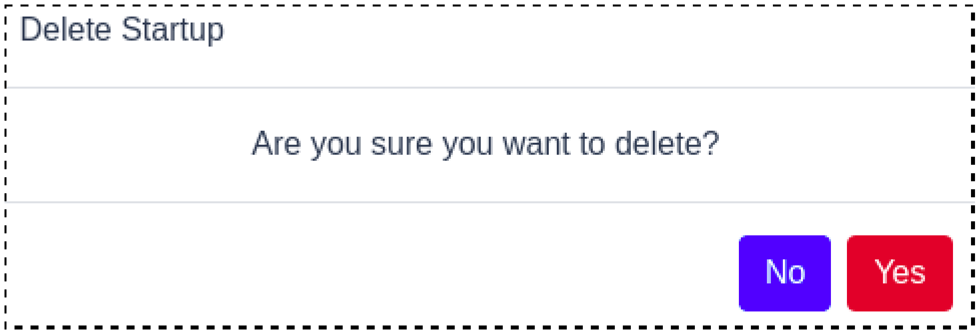
Startup Field*
Websites

Please note that modifying the value will lead to resetting suggested programming languages in general section

Max Budget Per Person (yearly, in USD)*
65

Figure 2.3.8.1 Startup details

Users can save in case the startup is drafted, update an existing startup, or delete it from the database.
When deleting the startup, a pop-up will appear making sure that the user is indeed intended to delete.

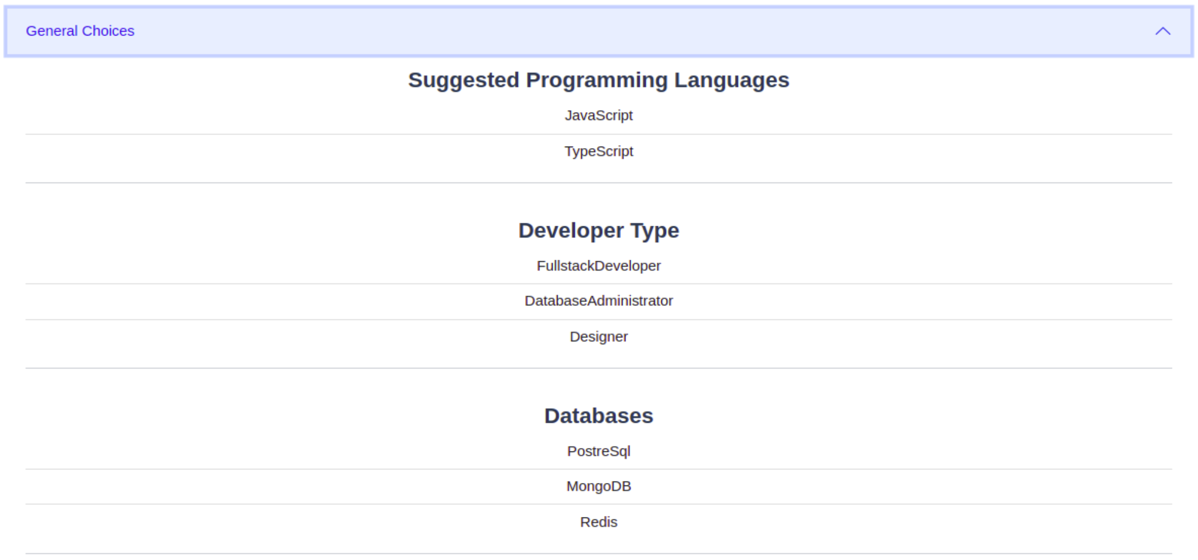


Are you sure you want to delete?

Figure 2.3.8.2: Deleting a startup

Since startup size, startup budget, and the deployment speed strictly yield to generating programming tools, users **can not** update those values later but can reset the form to start over. Users can update startup field values, however, if updated, suggested programming language choices for the specific field **will be reset**.

General (Figure 2.3.8.3-a) and team choices (Figure 2.3.8.3-b) picked by the users will be shown in accordions. Once the user clicks either one of the accordions, they will be expanded to list the tools picked by the users, or "No tools" text will be displayed in case there is no general/team choice picked by the user.


General Choices

Suggested Programming Languages

JavaScript

TypeScript

Developer Type

FullstackDeveloper

DatabaseAdministrator

Designer

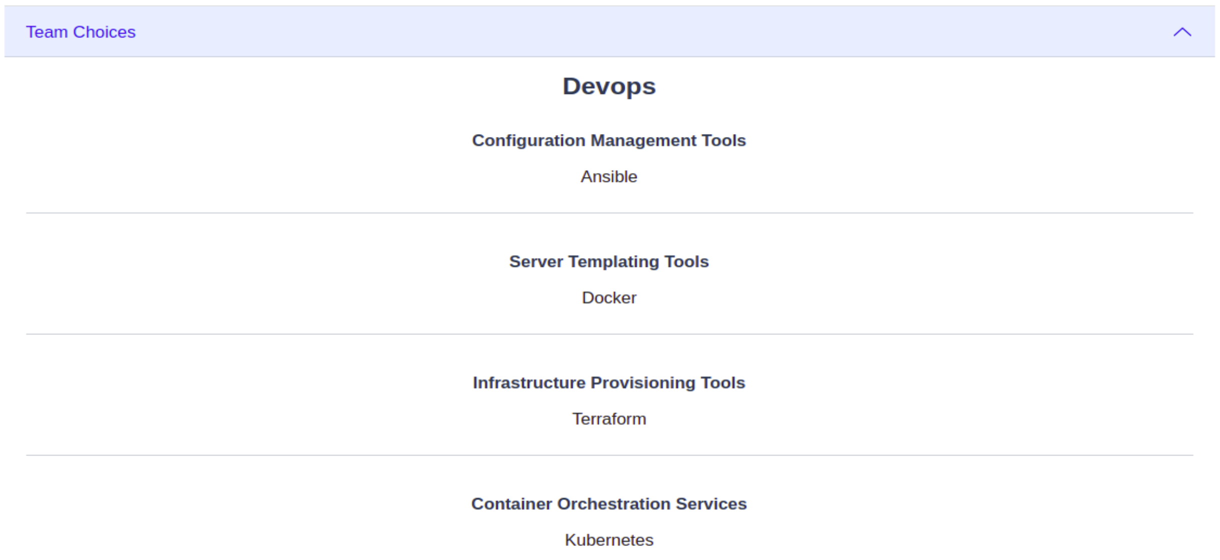
Databases

PostreSql

MongoDB

Redis

a) General choices


Team Choices

Devops

Configuration Management Tools

Ansible

Server Templating Tools

Docker

Infrastructure Provisioning Tools

Terraform

Container Orchestration Services

Kubernetes

b) Team choices

Figure 2.3.8.3: User picked choices

Chapter 3

Developer Documentation

Application structure, code snippets of the most important development logic, and use-case diagrams will be demonstrated in the chapter by using figures. Also, the relationship between the front end and the back end, the responsive design system, code quality, and the authorization details will be explained thoroughly.

3.1 Technologies used

The web application is split into two big sections: the client-side and the server-side.

3.1.1 Client-side technologies

The client-side application is developed by VueJS. Vue is a component-based progressive framework designed to create single-page applications (SPA). Thanks to its popularity and thus large community support, its ecosystem is rich with powerful libraries. One of them that is used by the application is *Vuex* which serves as a centralized store for the components within the application and ensures data changes in a predictable and safe approach [13]. The application also uses a router library called *Vue Router* which makes single-page applications fast and with the best user interface [14]. The latest version of Vue (Vue 3) along with its ecosystems are used throughout the development lifecycle.
As a UI framework, the latest version of *Bootstrap* (Bootstrap 5) has been used [15]. For data visualizations, the *Chart.js* library has been used [16].

3.1.2 Server-side technologies

The server part has been mainly developed by ExpressJS [17] which is a part of the NodeJS ecosystem. For database operations, two main databases have been used: MongoDB and Redis. Mongoose as a framework of MongoDB is used for getting user-specific details such as user startups, while Redis is used for blacklisting refresh tokens. To make the application highly secure, some popular libraries such as bcrypt [18] and JSON web token [19] have also been implemented. Finally, for testing purposes, the supertest library [20] has been used.

3.2 Application structure

To communicate between the client-side and the server-side, a connection through *HTTPS* needs to be established.

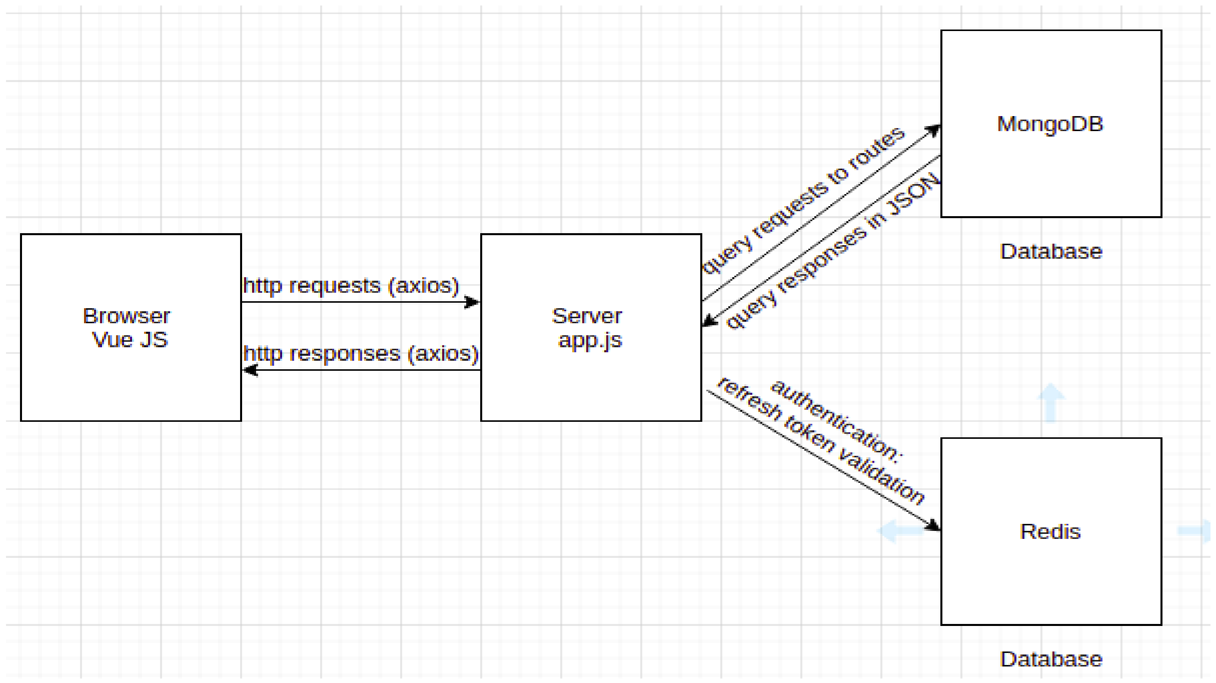
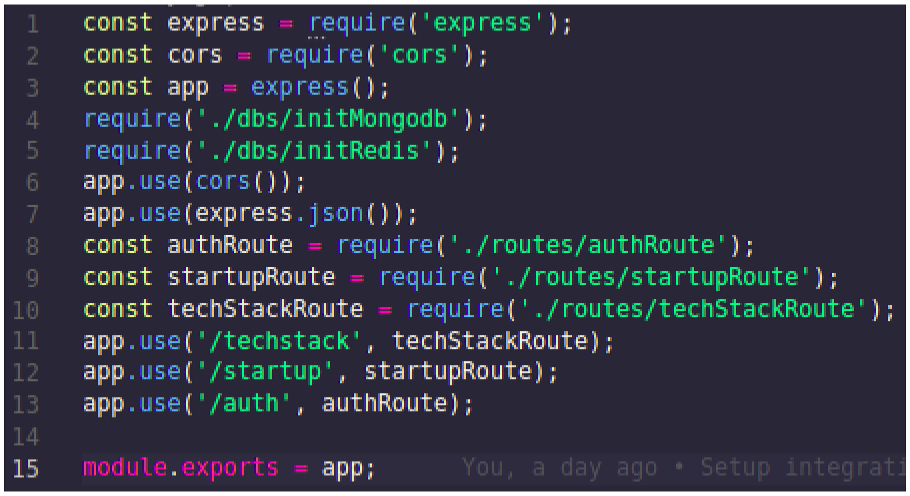


Figure 3.2.1: Application structure

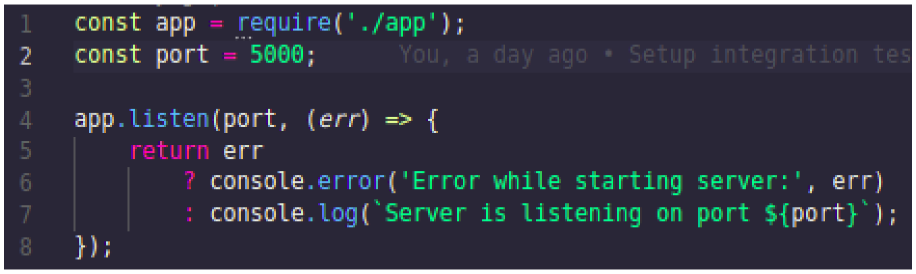
When running *npm run dev* in the backend, Express framework sets up the *app.js* file and creates a connection to both databases.



```
const express = require('express');
const cors = require('cors');
const app = express();
require('./dbs/initMongodb');
require('./dbs/initRedis');
app.use(cors());
app.use(express.json());
const authRoute = require('./routes/authRoute');
const startupRoute = require('./routes/startupRoute');
const techStackRoute = require('./routes/techStackRoute');
app.use('/techstack', techStackRoute);
app.use('/startup', startupRoute);
app.use('/auth', authRoute);

module.exports = app;      You, a day ago • Setup integrati
```

Figure 3.2.2: Backend app.js setup

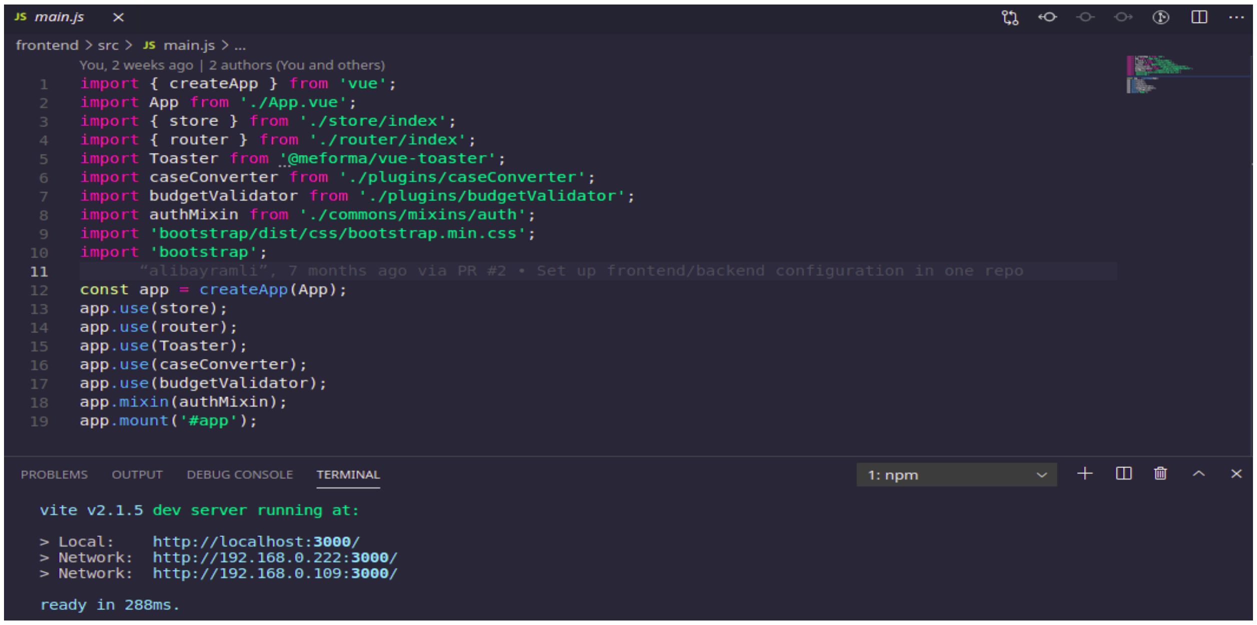
Then *server.js* listens continuously for every connection coming from the client and handles requests on a specific port.



```
const app = require('./app');
const port = 5000;      You, a day ago • Setup integration tes

app.listen(port, (err) => {
    return err
        ? console.error('Error while starting server:', err)
        : console.log(`Server is listening on port ${port}`);
});
```

Figure 3.2.3: Backend server.js setup

When running *npm run dev* in the frontend, The *main.js* will create the front-end application and mount *App.Vue* file and the sub-modules (state-management, router, plugins, etc.) on the DOM.



main.js

frontend > src > main.js > ...

You, 2 weeks ago | 2 authors (You and others)

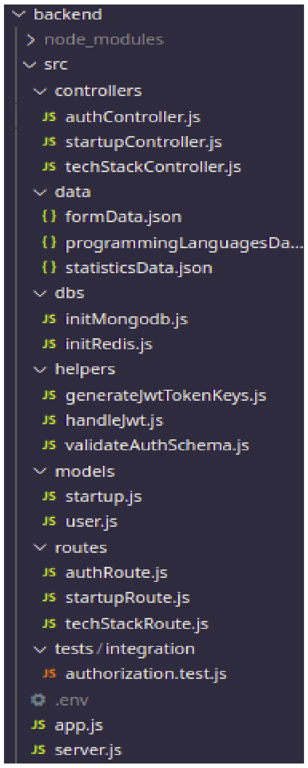
```
import { createApp } from 'vue';
import App from './App.vue';
import { store } from './store/index';
import { router } from './router/index';
import Toaster from '@meforma/vue-toaster';
import caseConverter from './plugins/caseConverter';
import budgetValidator from './plugins/budgetValidator';
import authMixin from './commons/mixins/auth';
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
        "alibayramli", 7 months ago via PR #2 • Set up frontend/backend configuration in one repo
const app = createApp(App);
app.use(store);
app.use(router);
app.use(Toaster);
app.use(caseConverter);
app.use(budgetValidator);
app.mixin(authMixin);
app.mount('#app');
```

```
  vite v2.1.5 dev server running at:

  > Local:    http://localhost:3000/
  > Network:  http://192.168.0.222:3000/
  > Network:  http://192.168.0.109:3000/

  ready in 288ms.
```

Figure 3.2.4: Vite compiler starts the client

3.3 Folder structure

3.3.1 Frontend



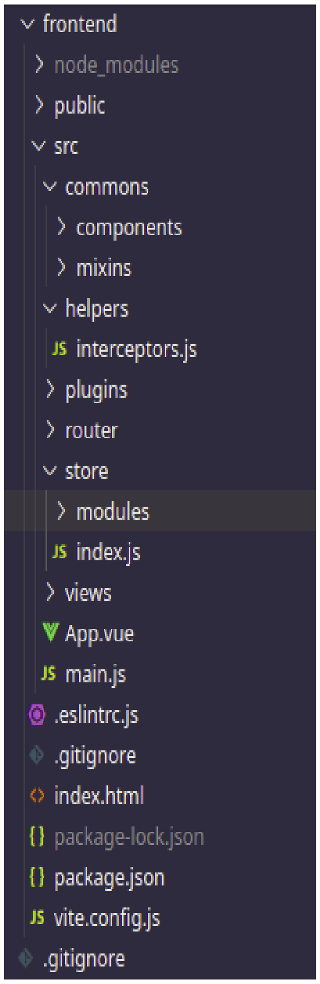
```
backend
  node_modules
  src
    controllers
      authController.js
      startupController.js
      techStackController.js
    data
      formData.json
      programmingLanguagesDa...
      statisticsData.json
    dbs
      initMongodb.js
      initRedis.js
    helpers
      generateJwtTokenKeys.js
      handleJwt.js
      validateAuthSchema.js
    models
      startup.js
      user.js
    routes
      authRoute.js
      startupRoute.js
      techStackRoute.js
    tests / integration
      authorization.test.js
  .env
  app.js
  server.js
```

In the backend, the application follows the model-view-controller pattern (known as MVC). Incoming requests are handled to routes by using Express route middlewares. The routes then pass the specific information to controllers where it handles data interactions with databases. Each of the databases has a script file that creates connections to the database drivers. The Mongoose framework creates models on the Mongoose schema which then defines the structure of the document and sets validators. The *data* folder contains developer survey results (languages, tools, etc. Figure 2.3.1.2) and also available options for the form fields (fields, locations, etc.) Both contents are stored in JSON format. For authentication, helpers methods such as handling and validating jwt tokens, validating authentication when logging in, and signing up have been used. The secret keys to MongoDB atlas drivers and jwt tokens are stored inside the .env file. Finally, the test files are located under the *tests* folder.

(**Figure 3.3.1.1** Backend folder structure)

The application uses code quality tools such as ESLint in both the front-end and the back-end which will be explained in section 3.6.

3.3.2 Frontend



```
v frontend
  > node_modules
  > public
  v src
    v commons
      > components
      > mixins
    v helpers
      JS interceptors.js
    > plugins
    > router
    v store
      > modules
      JS index.js
    > views
    App.vue
    JS main.js
  .eslintrc.js
  .gitignore
  index.html
  {} package-lock.json
  {} package.json
  JS vite.config.js
.gitignore
```

Here is how the front-end application looks like.

The Commons folder includes different components and mixins (reusable components).

Helpers folder includes the *interceptors.js* file that deals with every HTTP request and response using Axios. [21]

Plugins folder has helper files that are used when validating user inputs such as adding budget or converting words to start case letters.

The *router* folder defines routes that map to a specific component. It also sets up authentication guards to prevent unauthorized users from accessing specific pages. [22]

Store folder creates containers that hold the application state. States are grouped inside the modules folder. [23]

The components that can be used as routes are listed under the *views* folder.

Since the application uses the Vite compiler, the configuration details are listed in the *vite.config.js* file

Figure 3.3.2.1 Frontend folder structure

3.4 Use Case Diagram

Since the user interfaces are getting more in-demand in modern software development, the UX of the application has been set to be easy and simple to follow. Clear and straightforward instructions (warnings as the user types form input values, updating them, etc.) let the user focus on the software's main areas, preventing the user from wasting valuable time figuring out how the application flow is designed. **Figure 3.4.1** demonstrates how the user is interacting with the web application. A direct arrow between the circles means that the source case is a prerequisite while the target case depends on the source.

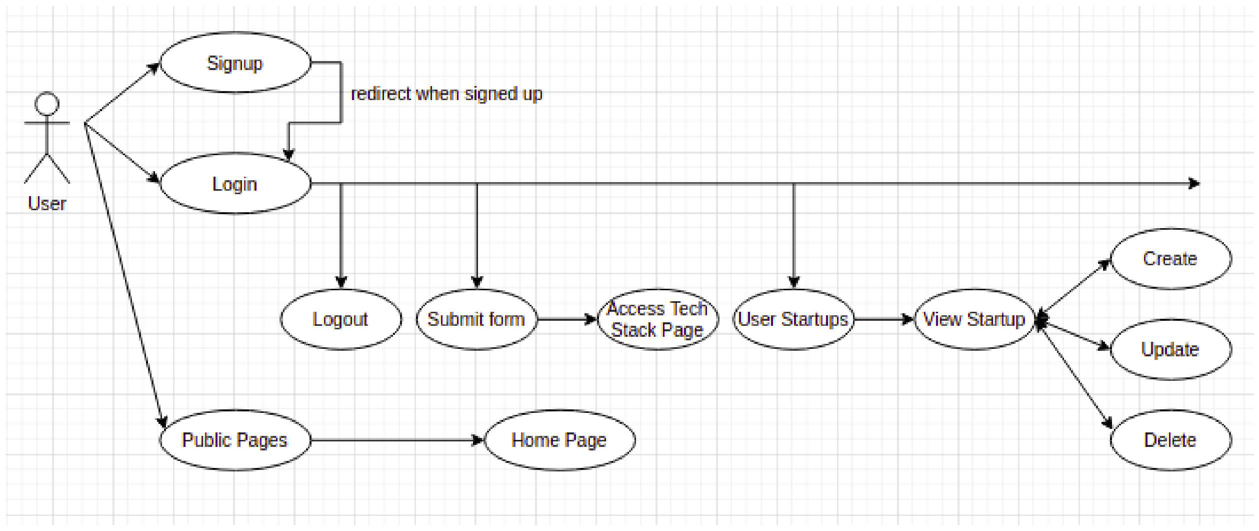
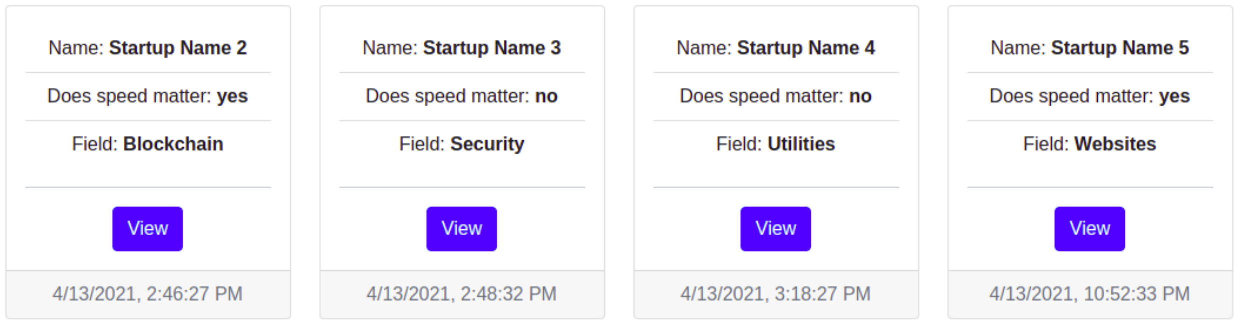


Figure 3.4.1 Use case diagram

3.5 Responsive Design

One of the main focuses of the application is to enable users to access the web application from various screen resolutions. That means a user can view the same content both in laptop size and the mobile size screens without losing any content or corrupting the UI. For this purpose, Bootstrap's grid system has come very handily to use. The following demonstrates when startup cards are viewed on laptops (Figure 3.5.1-a), tablets (Figure 3.5.1-b), and mobile devices (Figure 3.5.1-c).

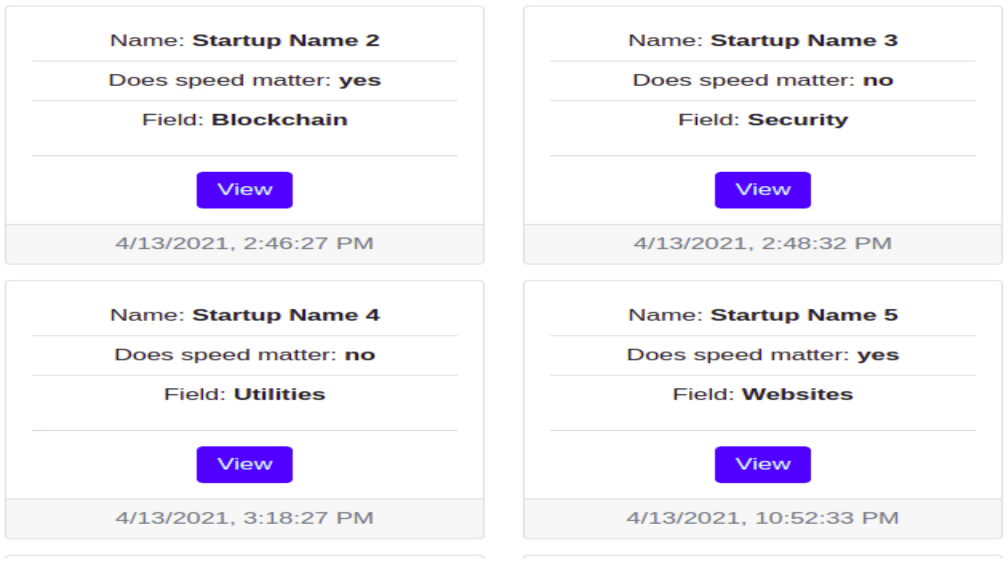

Name: **Startup Name 2**
Does speed matter: **yes**
Field: **Blockchain**
4/13/2021, 2:46:27 PM

Name: **Startup Name 3**
Does speed matter: **no**
Field: **Security**
4/13/2021, 2:48:32 PM

Name: **Startup Name 4**
Does speed matter: **no**
Field: **Utilities**
4/13/2021, 3:18:27 PM

Name: **Startup Name 5**
Does speed matter: **yes**
Field: **Websites**
4/13/2021, 10:52:33 PM

a) Laptop view

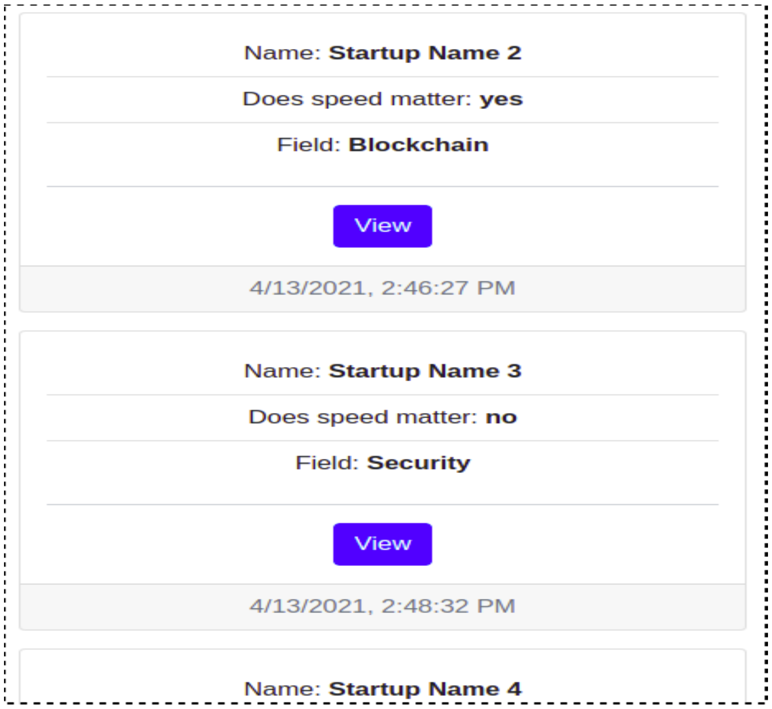

Name: **Startup Name 2**
Does speed matter: **yes**
Field: **Blockchain**
4/13/2021, 2:46:27 PM

Name: **Startup Name 3**
Does speed matter: **no**
Field: **Security**
4/13/2021, 2:48:32 PM

Name: **Startup Name 4**
Does speed matter: **no**
Field: **Utilities**
4/13/2021, 3:18:27 PM

Name: **Startup Name 5**
Does speed matter: **yes**
Field: **Websites**
4/13/2021, 10:52:33 PM

b) Tablet view


Name: **Startup Name 2**

Does speed matter: **yes**

Field: **Blockchain**

4/13/2021, 2:46:27 PM

Name: **Startup Name 3**

Does speed matter: **no**

Field: **Security**

4/13/2021, 2:48:32 PM

Name: **Startup Name 4**

c) Mobile view

Figure 3.5.1 Startup cards in different screen resolutions

3.6 Code Quality

Adhering to code quality standards is quite essential for the software to scale easily and less prone to wasting money and time when it comes to debugging. The application aims to achieve some of the most important code quality standards such as:

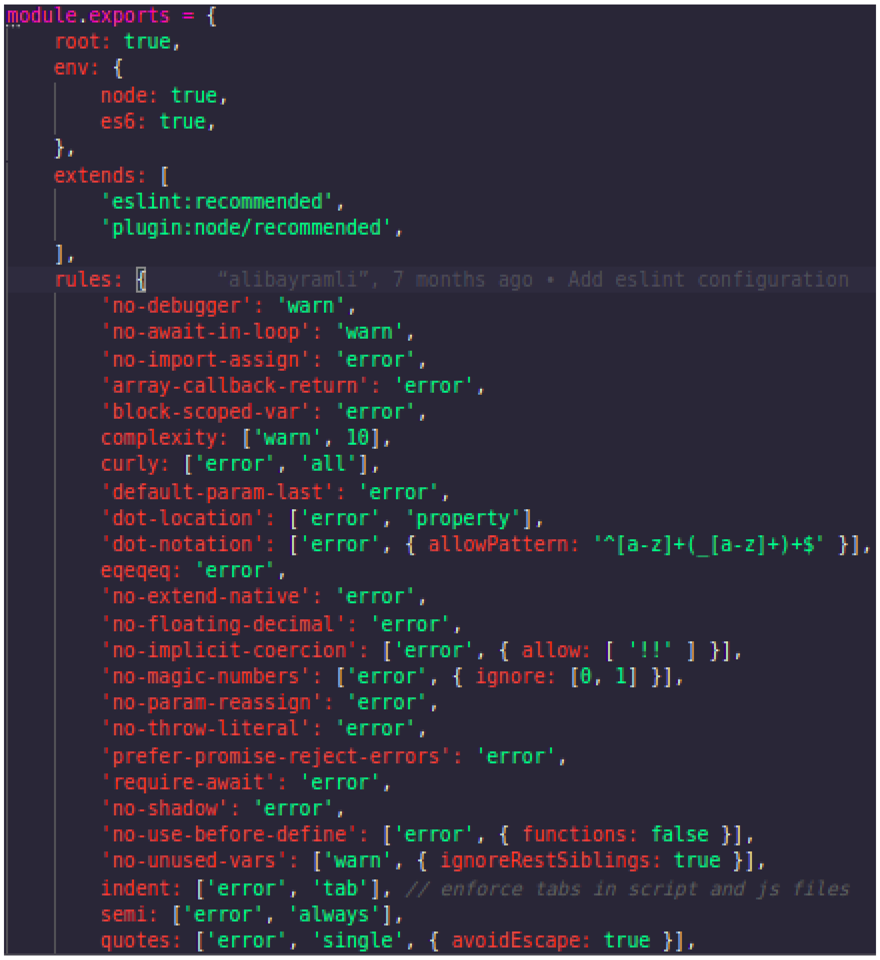
Readability: The application makes sure that the code written is easy to read and includes simplicity.

Predictability: The behaviors of the codes are predictable, meaning they are less prone to hidden bugs.

Maintainability: fixing, updating, improving, in other words scaling the application is not complex.

To achieve high quality, the application uses the proper coding standards set by ESLint. [24]

Figure 3.6.1 illustrates some of the customs rules that have been applied in the front-end part of the application. If any of the rules are violated while the developer is coding, ESLint will highlight the specific part with the status of the rule (warning, error) and may suggest auto-fixes or alternatives for refactoring.



```
module.exports = {
    root: true,
    env: {
        node: true,
        es6: true,
    },
    extends: [
        'eslint:recommended',
        'plugin:node/recommended',
    ],
    rules: {      "alibayramli", 7 months ago • Add eslint configuration
        'no-debugger': 'warn',
        'no-await-in-loop': 'warn',
        'no-import-assign': 'error',
        'array-callback-return': 'error',
        'block-scoped-var': 'error',
        complexity: ['warn', 10],
        curly: ['error', 'all'],
        'default-param-last': 'error',
        'dot-location': ['error', 'property'],
        'dot-notation': ['error', { allowPattern: '^[a-z]+(_[a-z]+)+$' }],
        eqeqeq: 'error',
        'no-extend-native': 'error',
        'no-floating-decimal': 'error',
        'no-implicit-coercion': ['error', { allow: [ '!!' ] }],
        'no-magic-numbers': ['error', { ignore: [0, 1] }],
        'no-param-reassign': 'error',
        'no-throw-literal': 'error',
        'prefer-promise-reject-errors': 'error',
        'require-await': 'error',
        'no-shadow': 'error',
        'no-use-before-define': ['error', { functions: false }],
        'no-unused-vars': ['warn', { ignoreRestSiblings: true }],
        indent: ['error', 'tab'], // enforce tabs in script and js files
        semi: ['error', 'always'],
        quotes: ['error', 'single', { avoidEscape: true }],
```

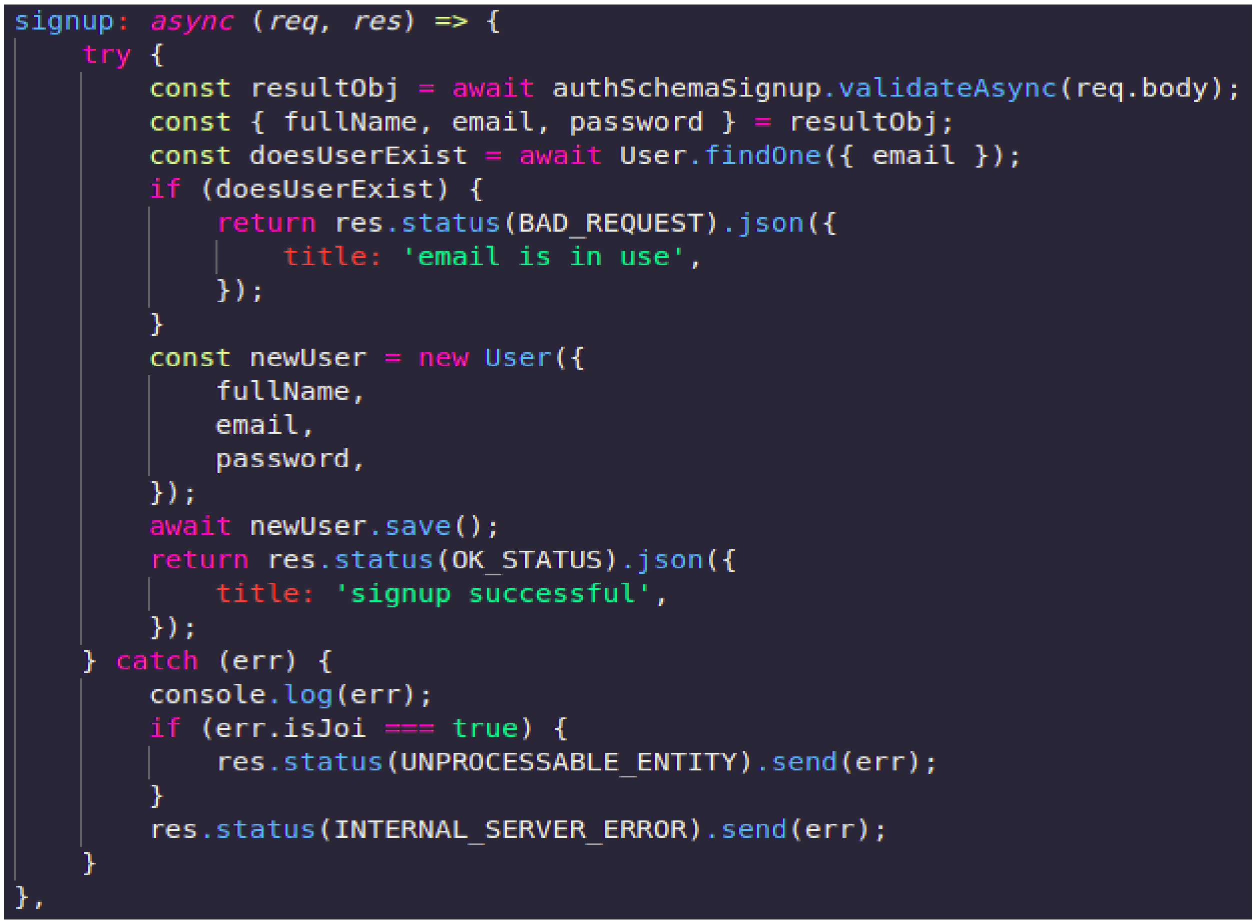
Figure 3.6.1: coding rules set in front-end .eslintrc.js

3.7 Authorization

The core part of application development relies heavily on authorization. In this part, security details related to the authorization, public/protected contents will be explained elaborately [25].

3.7.1 Registration

User registration is processed under the "*/auth/signup*" route that passes it to the authentication controller. When a user sends the request, the request object is already validated on the client-side (Section 2.3.2), however, due to security reasons, validation will take place on the back-end side as well. The signup process is illustrated on the back-end.



```
signup: async (req, res) => {
    try {
        const resultObj = await authSchemaSignup.validateAsync(req.body);
        const { fullName, email, password } = resultObj;
        const doesUserExist = await User.findOne({ email });
        if (doesUserExist) {
            return res.status(BAD_REQUEST).json({
                title: 'email is in use',
            });
        }
        const newUser = new User({
            fullName,
            email,
            password,
        });
        await newUser.save();
        return res.status(OK_STATUS).json({
            title: 'signup successful',
        });
    } catch (err) {
        console.log(err);
        if (err.isJoi === true) {
            res.status(UNPROCESSABLE_ENTITY).send(err);
        }
        res.status(INTERNAL_SERVER_ERROR).send(err);
    }
},
```

Figure 3.7.1.1: Registration process on the back-end

The object retrieved from the *req.body* is verified via the method *validateAsync* that uses a library called *JOI* for authenticating the user credentials. If any error occurs, the library will throw that in the catch block and the process will be stopped. If the user is already registered, meaning the email address has been found in the database, an *"e-mail is in use"* error message will be sent to the client. Otherwise, creating a new user process will be started. Before saving, Mongoose has a middleware function called pre which hashes the passwords via the bcrypt library and then saves the generated hash password to the database.

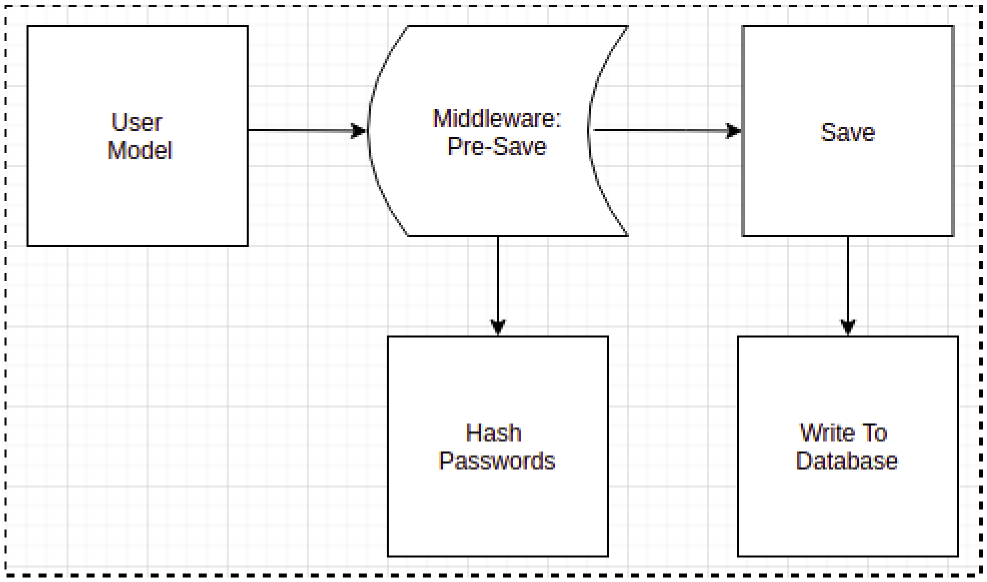
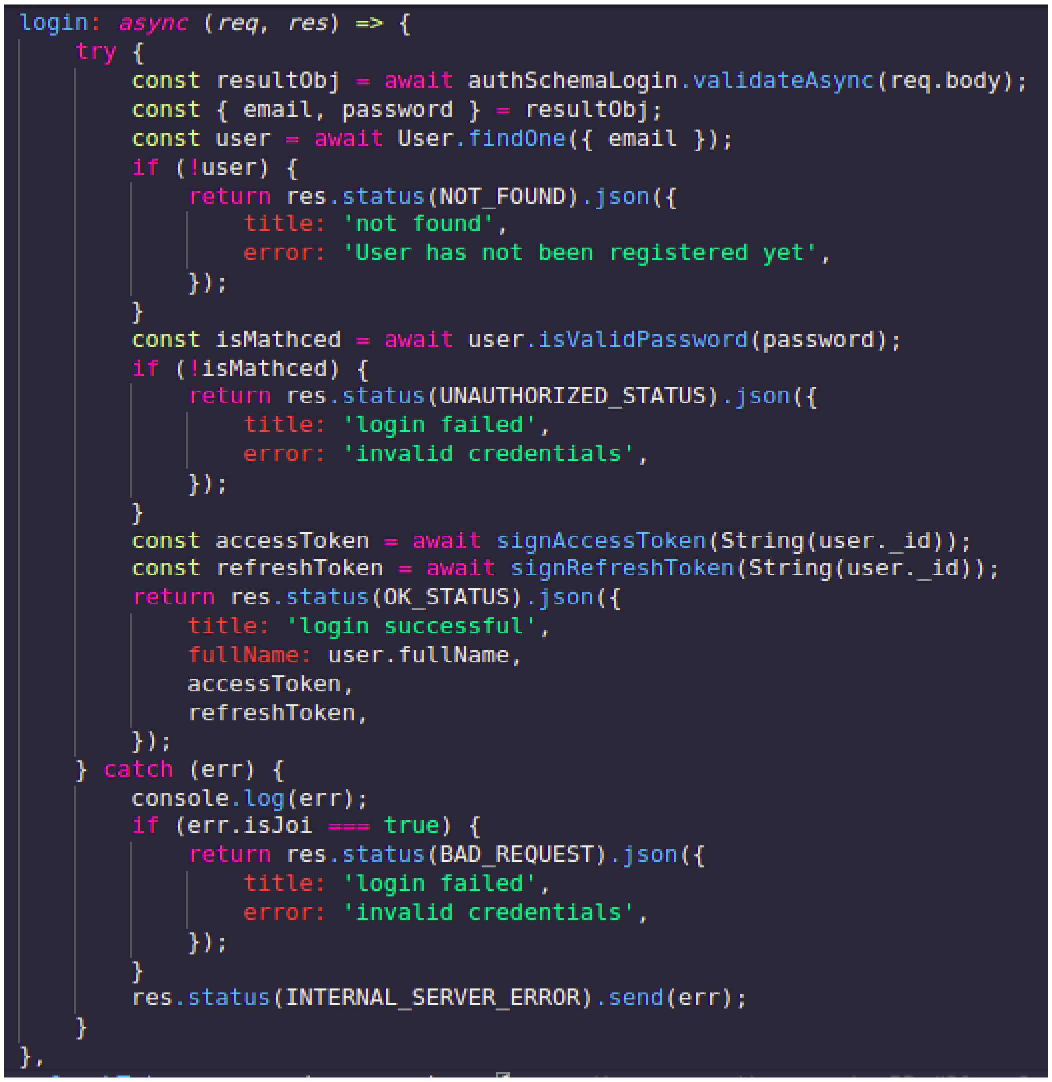


Figure 3.7.1.2: Mongoose middleware for saving passwords

After saving, a *"signup successful message"* will be sent to the client and the user can start the login process afterward.

3.7.2 Login

User registration is processed under the "/*auth/login*" route that passes it to the authentication controller. Similar server-side validation in the registration part (except checking registered emails and verifying hashed passwords) will take place in the login part as well and the corresponding results will be sent to the client. However, the main difference is in case the login process is successful, **access and refresh tokens** are generated by the JSON Web Token library (in other words, *jwt*) and sent to the client.

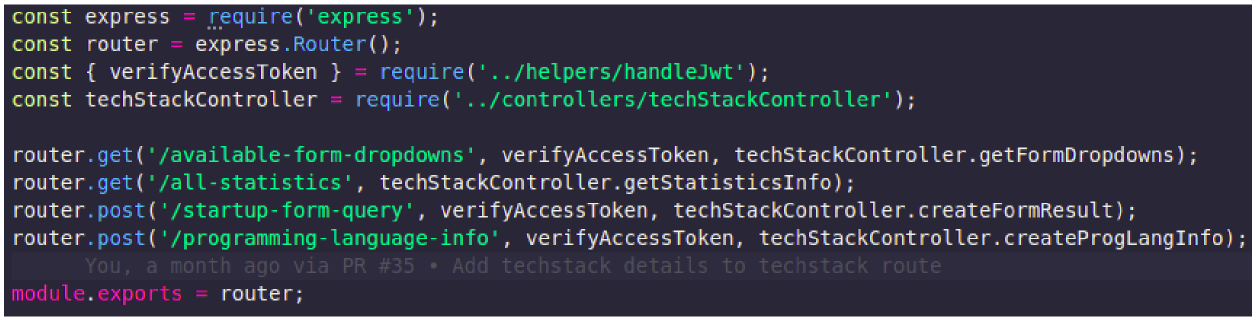


```
login: async (req, res) => {
    try {
        const resultObj = await authSchemaLogin.validateAsync(req.body);
        const { email, password } = resultObj;
        const user = await User.findOne({ email });
        if (!user) {
            return res.status(NOT_FOUND).json({
                title: 'not found',
                error: 'User has not been registered yet',
            });
        }
        const isMathced = await user.isValidPassword(password);
        if (!isMathced) {
            return res.status(UNAUTHORIZED_STATUS).json({
                title: 'login failed',
                error: 'invalid credentials',
            });
        }
        const accessToken = await signAccessToken(String(user._id));
        const refreshToken = await signRefreshToken(String(user._id));
        return res.status(OK_STATUS).json({
            title: 'login successful',
            fullName: user.fullName,
            accessToken,
            refreshToken,
        });
    } catch (err) {
        console.log(err);
        if (err.isJoi === true) {
            return res.status(BAD_REQUEST).json({
                title: 'login failed',
                error: 'invalid credentials',
            });
        }
        res.status(INTERNAL_SERVER_ERROR).send(err);
    }
},
```

Figure 3.7.2.1: Login process on the back-end

3.7.3 Access and Refresh tokens

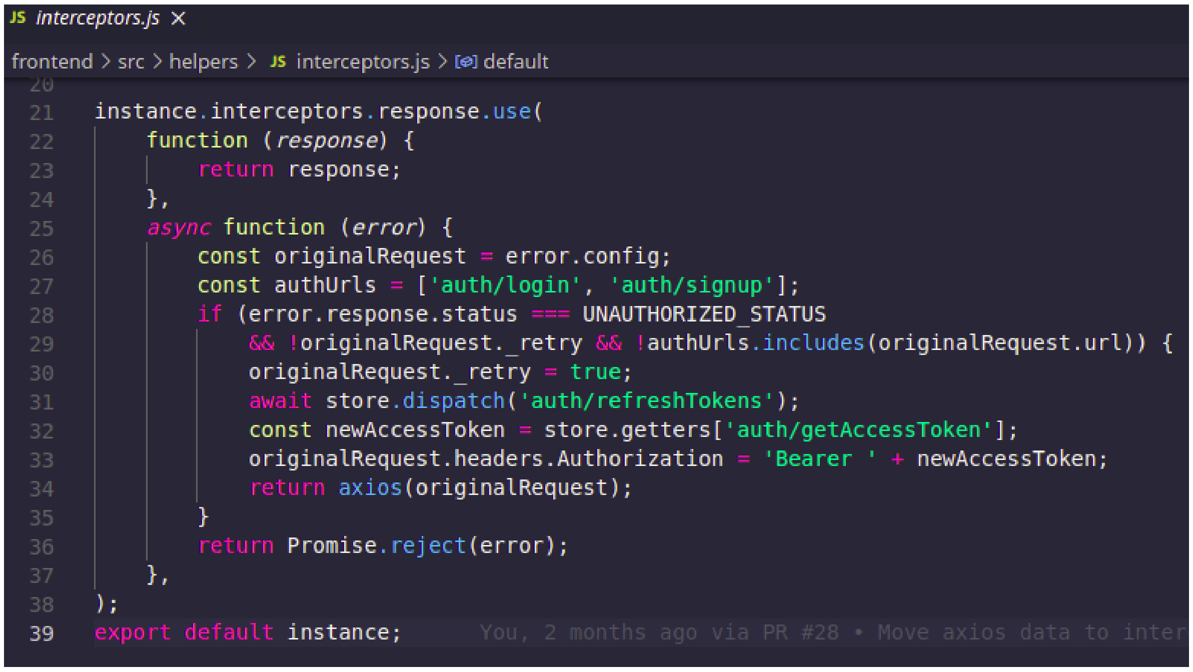
Access tokens enable users to retrieve protected contents that apply only to the user.
Each time private content is requested, access tokens are verified and the verification happens in the *handleJwt* file. *verifyAccessToken* is a middleware function and moves to the next iteration if the access token is valid, otherwise throws an error. In case the route is public, the middleware function is not passed to the header.



```
const express = require('express');
const router = express.Router();
const { verifyAccessToken } = require('../helpers/handleJwt');
const techStackController = require('../controllers/techStackController');

router.get('/available-form-dropdowns', verifyAccessToken, techStackController.getFormDropdowns);
router.get('/all-statistics', techStackController.getStatisticsInfo);
router.post('/startup-form-query', verifyAccessToken, techStackController.createFormResult);
router.post('/programming-language-info', verifyAccessToken, techStackController.createProgLangInfo);
    You, a month ago via PR #35 • Add techstack details to techstack route
module.exports = router;
```

Figure 3.7.3.1: Accessing protected routes

Access tokens have a shorter lifespan due to security reasons, and once they are expired, the protected content can no longer be reached. However, upon the request, the system can generate a new access token by using refresh tokens. This happens when Axios response interceptors in the client detect that the server response code is **401** requests new tokens.



interceptors.js ×

frontend > src > helpers > interceptors.js > default

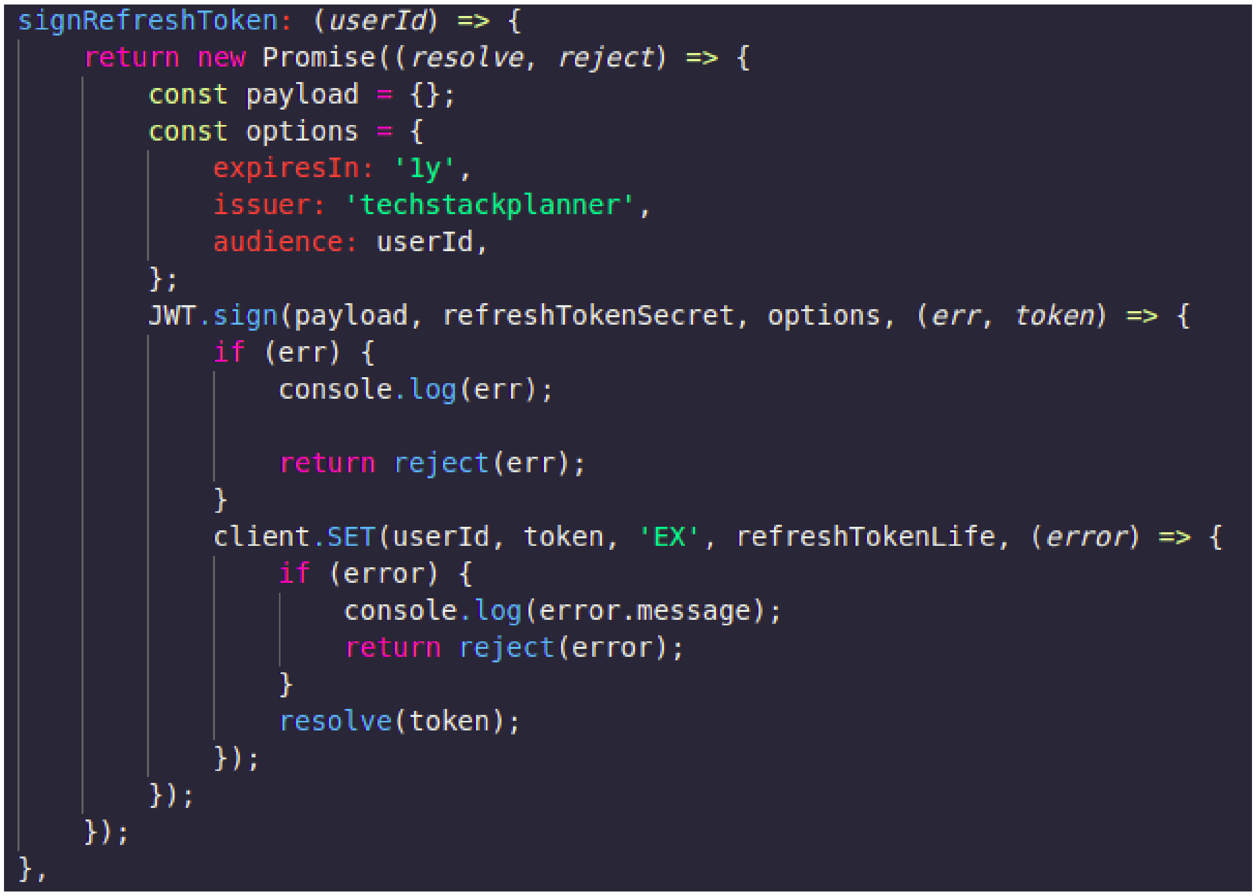
```
instance.interceptors.response.use(
    function (response) {
        return response;
    },
    async function (error) {
        const originalRequest = error.config;
        const authUrls = ['auth/login', 'auth/signup'];
        if (error.response.status === UNAUTHORIZED_STATUS
            && !originalRequest._retry && !authUrls.includes(originalRequest.url)) {
            originalRequest._retry = true;
            await store.dispatch('auth/refreshTokens');
            const newAccessToken = store.getters['auth/getAccessToken'];
            originalRequest.headers.Authorization = 'Bearer ' + newAccessToken;
            return axios(originalRequest);
        }
        return Promise.reject(error);
    },
);
export default instance;      You, 2 months ago via PR #28 • Move axios data to inter
```

Figure 3.7.3.2: Refreshing expired token and requesting again

The new request in the back-end verifies the refresh token to check whether the token was generated by jwt. If it is a valid one, the system signs a new pair of access and refresh tokens and sends it back to the client.

3.7.4 Using Redis

The refresh tokens have a much longer lifespan and the valid refresh tokens are stored in Redis. Redis is an in-memory database that gives quite fast access to the refresh tokens. The reason to store refresh tokens is to invalidate a user's long-lived tokens, leading to big security issues. Redis is used when signing a new refresh token or verifying the already existed ones.



```
signRefreshToken: (userId) => {
    return new Promise((resolve, reject) => {
        const payload = {};
        const options = {
            expiresIn: '1y',
            issuer: 'techstackplanner',
            audience: userId,
        };
        JWT.sign(payload, refreshTokenSecret, options, (err, token) => {
            if (err) {
                console.log(err);

                return reject(err);
            }
            client.SET(userId, token, 'EX', refreshTokenLife, (error) => {
                if (error) {
                    console.log(error.message);
                    return reject(error);
                }
                resolve(token);
            });
        });
    });
},
```

Figure 3.7.4.1: Signing a refresh token and storing it in Redis

Client.set method stores the token in the database and removes it once the lifespan of the token is expired.

3.7.5 Logout

When the user logs out of the system, a logout request is made on the path "/auth/*logout*". After checking the validity of the refresh token, the system deletes the refresh token stored in Redis and logs out the user.

3.7.6 Inaccessible Pages

If the user enters a URL that is not valid, the Vue router will redirect the application to the *"not found"* view where the user will see an image containing a text stating the page is not found [26].

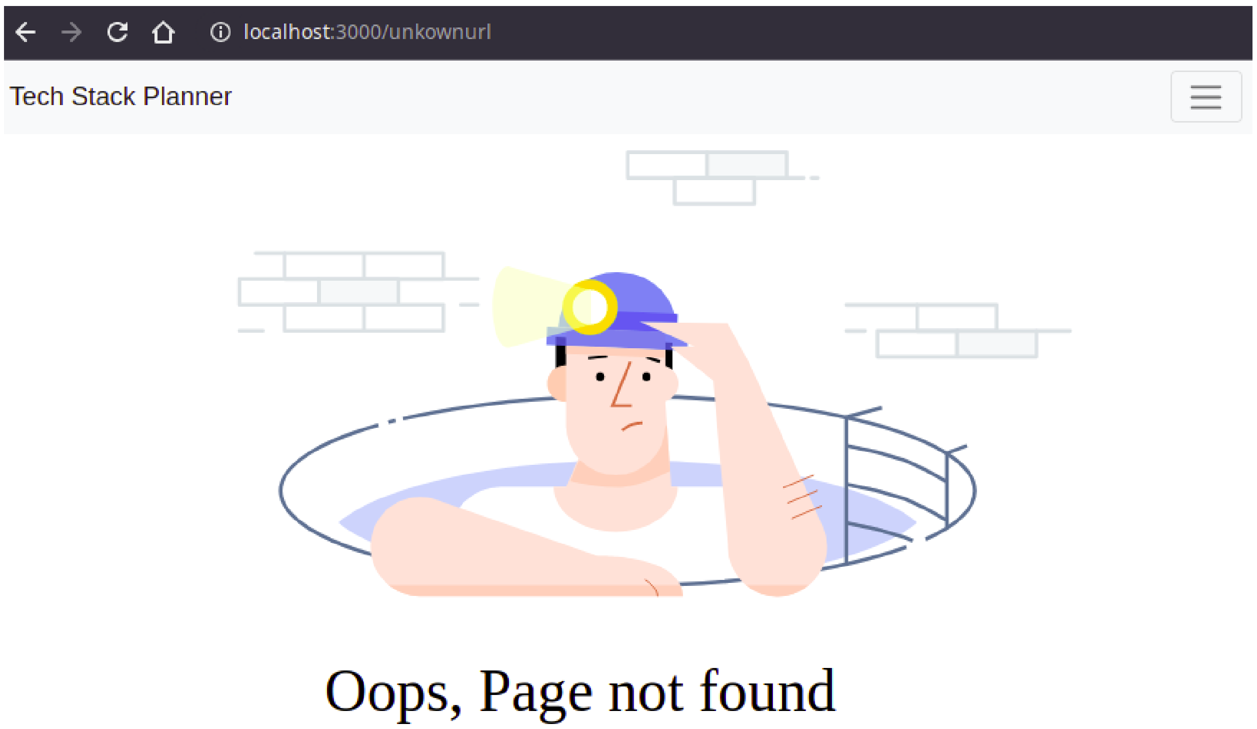
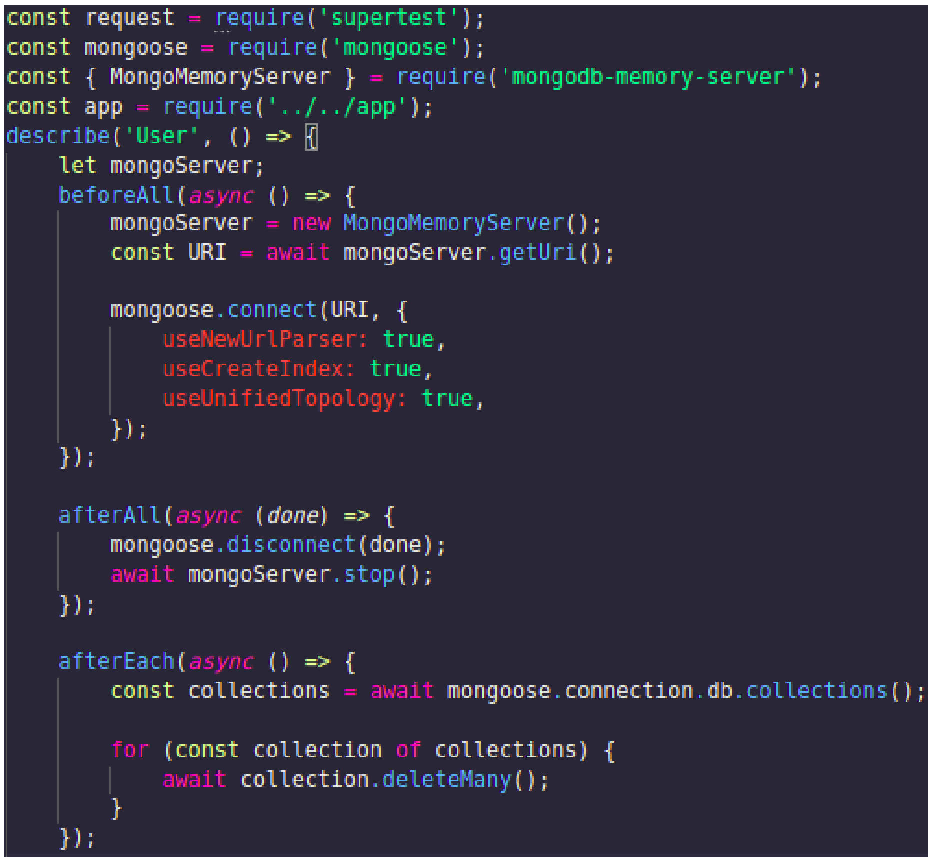


Figure 3.7.6.1: Page not found

3.8 Testing

This chapter demonstrates the testing methods used for the application.
In the backend, integration tests have been used for different API endpoints [27]. Test files are located under the *backend/src/tests* folder and the tests can be run by navigating to the backend directory and typing *the npm run test* command in the terminal. The script determines the file ".test.js" suffix and runs all the test suites. First, an in-memory MongoDB database is set up before the testing is started. This process happens inside *beforeAll* hook with the help of the *mongodb-memory-server* library. Each one of the test case impacts is wiped out of the database via afterEach hook. Once all the test cases proceeded, the connection to the MongoDB server is shut down in the *afterAll* hook.



```
const request = require('supertest');
const mongoose = require('mongoose');
const { MongoMemoryServer } = require('mongodb-memory-server');
const app = require('../../app');
describe('User', () => {
    let mongoServer;
    beforeAll(async () => {
        mongoServer = new MongoMemoryServer();
        const URI = await mongoServer.getUri();

        mongoose.connect(URI, {
            useNewUrlParser: true,
            useCreateIndex: true,
            useUnifiedTopology: true,
        });
    });

    afterAll(async (done) => {
        mongoose.disconnect(done);
        await mongoServer.stop();
    });

    afterEach(async () => {
        const collections = await mongoose.connection.db.collections();

        for (const collection of collections) {
            await collection.deleteMany();
        }
    });
```

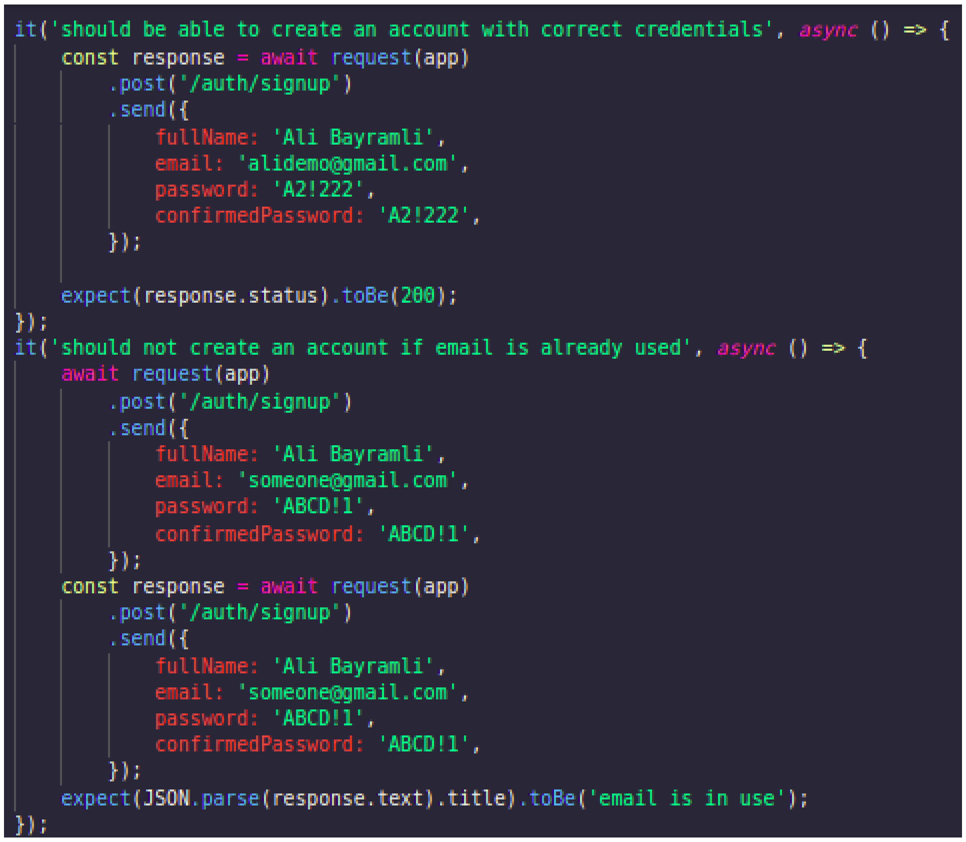
Figure 3.8.1: MongoDB in-memory setup in testing

Once the initial setup is done, the tests can be executed within ***it*** block.

Related to signup, two edge cases are tested:

1. When the user can signup with correct credentials, this time a status code of **200** should appear as a response to the API request.

2. If the user email is already used, the server response should contain a title as **"email is in use"**.



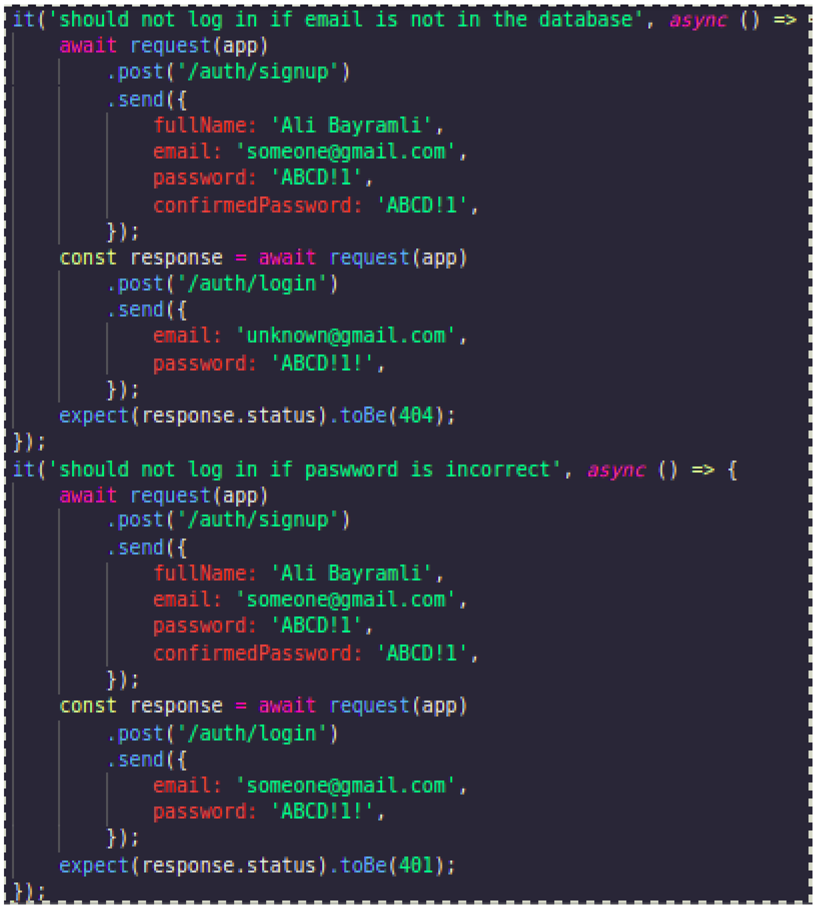
```
it('should be able to create an account with correct credentials', async () => {
    const response = await request(app)
        .post('/auth/signup')
        .send({
            fullName: 'Ali Bayramli',
            email: 'alidemo@gmail.com',
            password: 'A2!222',
            confirmedPassword: 'A2!222',
        });

    expect(response.status).toBe(200);
});
it('should not create an account if email is already used', async () => {
    await request(app)
        .post('/auth/signup')
        .send({
            fullName: 'Ali Bayramli',
            email: 'someone@gmail.com',
            password: 'ABCD!1',
            confirmedPassword: 'ABCD!1',
        });
    const response = await request(app)
        .post('/auth/signup')
        .send({
            fullName: 'Ali Bayramli',
            email: 'someone@gmail.com',
            password: 'ABCD!1',
            confirmedPassword: 'ABCD!1',
        });
    expect(JSON.parse(response.text).title).toBe('email is in use');
});
```

Figure 3.8.2: Testing signup endpoints

Related to log in, a few edge cases are tested, some of them include:

1. If the user email is not registered before, the login attempt should fail and a **404** response code should be sent back.

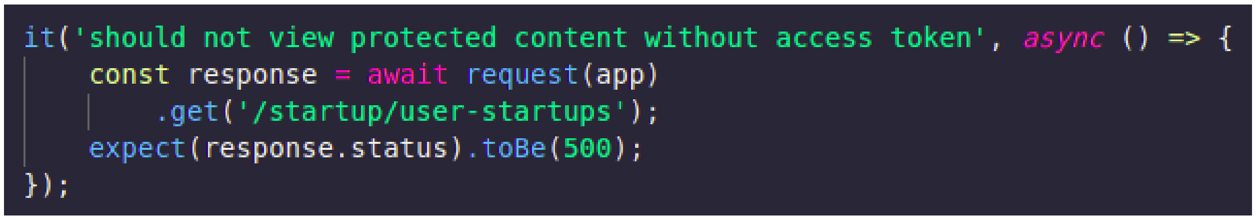
2. If the user provides an incorrect password, a **401** response code should be sent back.



```
it('should not log in if email is not in the database', async () =>
    await request(app)
        .post('/auth/signup')
        .send({
            fullName: 'Ali Bayramli',
            email: 'someone@gmail.com',
            password: 'ABCD!1',
            confirmedPassword: 'ABCD!1',
        });
    const response = await request(app)
        .post('/auth/login')
        .send({
            email: 'unknown@gmail.com',
            password: 'ABCD!1!',
        });
    expect(response.status).toBe(404);
});
it('should not log in if password is incorrect', async () => {
    await request(app)
        .post('/auth/signup')
        .send({
            fullName: 'Ali Bayramli',
            email: 'someone@gmail.com',
            password: 'ABCD!1',
            confirmedPassword: 'ABCD!1',
        });
    const response = await request(app)
        .post('/auth/login')
        .send({
            email: 'someone@gmail.com',
            password: 'ABCD!1!',
        });
    expect(response.status).toBe(401);
});
```

Figure 3.8.3: Testing login endpoints

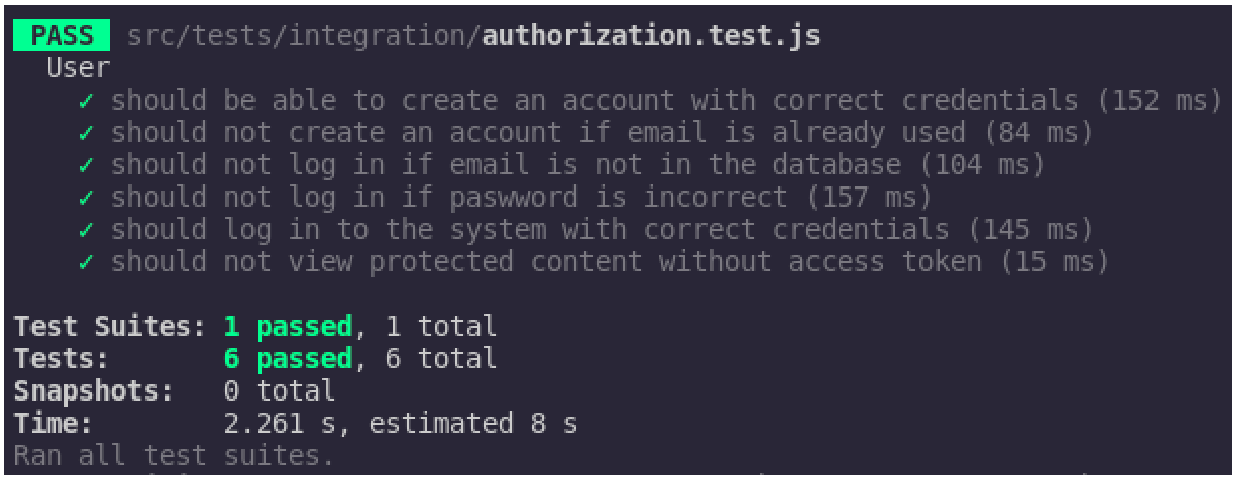
Finally, a test case is written related to the security of the protected routes. If a user sends an *HTTP request* to startup endpoints but doesn't have a valid access token, the server should respond with a **500** status code.



```
it('should not view protected content without access token', async () => {
    const response = await request(app)
        .get('/startup/user-startups');
    expect(response.status).toBe(500);
});
```

Figure 3.8.4: Testing login endpoints

Once all the tests are executed, the result table will be shown in the terminal which contains the status of the test: showing several tests that passed as well as the failed ones with proper error messages.



```
 PASS  src/tests/integration/authorization.test.js
  User
    ✓ should be able to create an account with correct credentials (152 ms)
    ✓ should not create an account if email is already used (84 ms)
    ✓ should not log in if email is not in the database (104 ms)
    ✓ should not log in if paswword is incorrect (157 ms)
    ✓ should log in to the system with correct credentials (145 ms)
    ✓ should not view protected content without access token (15 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        2.261 s, estimated 8 s
Ran all test suites.
```

Figure 3.8.5: Test results

Chapter 4

Conclusions and future work

4.1 Conclusion

This thesis was inspired by the vast importance of technology stacks in the software development industry that was introduced in chapter 1. The purpose of this thesis was to understand why different technology stacks could result in various performance and scalability issues and how to raise awareness by choosing the set of tools for startup owners as well as facilitate the selection process by providing modern and relevant examples. Based on the analysis contacted, the thesis work aims that the startups can make a huge impact on the market while scaling without wasting valuable resources.

4.2 Future work

Some sections provide limited research or features and can be extended as recommendations for further work. One of them would be adding steps for deployment and hosting it on a web service such as AWS. Before the deployment phase, more testing methods, such as unit and end-to-end tests should be provided. Eventually, more developer survey results can be obtained to have a highly customized tech stack for specific startup fields and locations.

Bibliography

[1] Vue 3 Documentation,

URL: https://v3.vuejs.org/guide/introduction.html

[2] ViteJS Documentation,

URL: https://vitejs.dev/guide/

[3] NodeJS Documentation,

URL: https://nodejs.org/en/docs/

[4] Node Package Manager (npm) Documentation,

URL: https://docs.npmjs.com/

[5] Redis Documentation,

URL: https://redis.io/documentation

[6] MongoDB Documentation,

URL: https://docs.mongodb.com/

[7] MongoDB Atlas Documentation,

URL: https://docs.atlas.mongodb.com/

[8] Git Documentation,

URL: https://git-scm.com/doc/

[9] Ali Bayramli, "Tech Stack Planner" Github repository,

URL: https://github.com/alibayramli/vue-node-techstack-planner

[10] Stack Overflow Developer Survey 2020,

URL: https://insights.stackoverflow.com/survey/2020

[11] JetBrains Developer Survey 2020,

URL: https://www.jetbrains.com/lp/devecosystem-2020/

[12] An article, "Why are Big Companies Embracing DevOps"

URL: https://www.infosys.com/iki/insights/companies-embracing.html

[13] Vuex Documentation,

URL: https://next.vuex.vuejs.org/guide/

[14] Vue Router Documentation,

URL: https://next.router.vuejs.org/introduction.html

[15] Bootstrap 5 Documentation,

URL: https://getbootstrap.com/docs/5.0/getting-started/introduction/

[16] ChartJS Documentation,

URL: https://www.chartjs.org/docs/latest/

[17] ExpressJS Documentation,

URL: https://expressjs.com/

[18] Bcrypt Library,

URL: https://www.npmjs.com/package/bcrypt

[19] JSON Web Token Documentation,

URL: https://jwt.io/introduction

[20] Supertest Library,

URL: https://www.npmjs.com/package/supertest

[21] Axios Client,

URL: https://github.com/axios/axios

[22] Authentication guards on the client,

URL: https://router.vuejs.org/guide/advanced/navigation-guards.html

[23] Vuex Modules,

URL: https://next.vuex.vuejs.org/guide/modules.html

[24] ESLint Documentation,

URL: https://eslint.org/

[25] Trulymittal, “API-Authentication-NodeJs”, Github repository,

URL: https://github.com/trulymittal/API-Authentication-NodeJs

[26] IlluStatus, “not-found svg animation”,

URL: https://github.com/blairlee227/IlluStatus

[27] Jpedroschmitz, “jest-mongodb” Github repository,

URL: https://github.com/jpedroschmitz/jest-mongodb