Project Updates Report

Med Ali Wachani - Tasnim Ferchichi - Med Aziz Ben Rejeb

April 10, 2023

1 Introduction

In this report, we will be providing an update on the progress of the project for this week. We managed to finalize our full stack web project: Task Management App (React TS, Express TS and MySQL) and to dockerize the builds into images in order to use them later on in our project.
Here is our repository: **GitHub Repository**

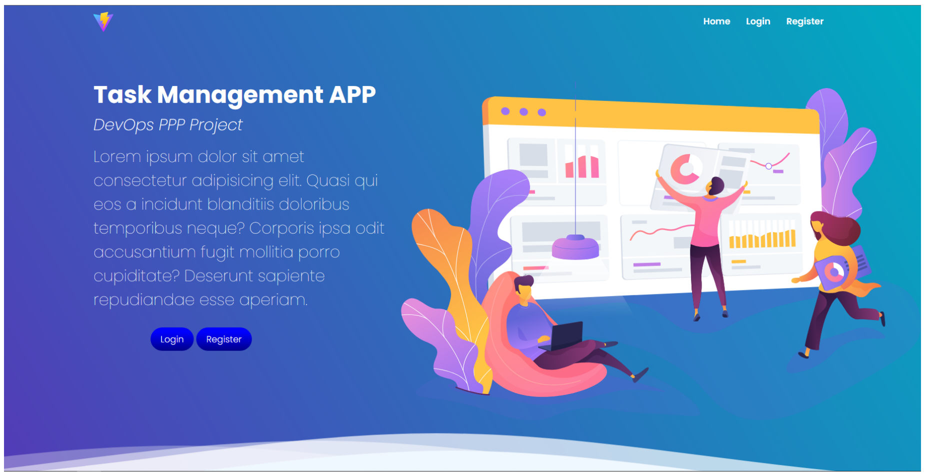

Home
Login
Register
Task Management APP
DevOps PPP Project
Lorem ipsum dolor sit amet consectetur adipisicing elit. Quasi qui eos a incidunt blanditiis doloribus temporibus neque? Corporis ipsa odit accusantium fugit mollitia porro cupiditate? Deserunt sapiente repudiandae esse aperiam.
Login
Register

Figure 1: Main Page

2 Overview

The project is on track and we have made significant progress since the last update. Our focus has been on completing the following milestones:

- **Back-end**: Implementing authentication middle-ware for every route of our back-end server.

- **Front-end Pages**: Home, Login and Register pages (authentication)
  For the authenticated user: Projects, sections and tasks pages.
- Building images using **Docker** and adding **Nginx** to the front-end image as a lightweight HTTP server to allow better control and customization of the serving process and as a reverse proxy to ensure the smooth flow of network traffic between clients and servers.

3 Milestones

3.1 Back-End Developement

In our back-end, we're using TypeORM that runs on Express TS in order to use its additional features which makes it easier dealing with database queries. In this report, we're going to use local MySQL image with docker, but in the end we're going to use the azure.com MySQL database like mentioned in our last week's report.
Since our application is going to be a task management app, we decided to use this model as our database schema:

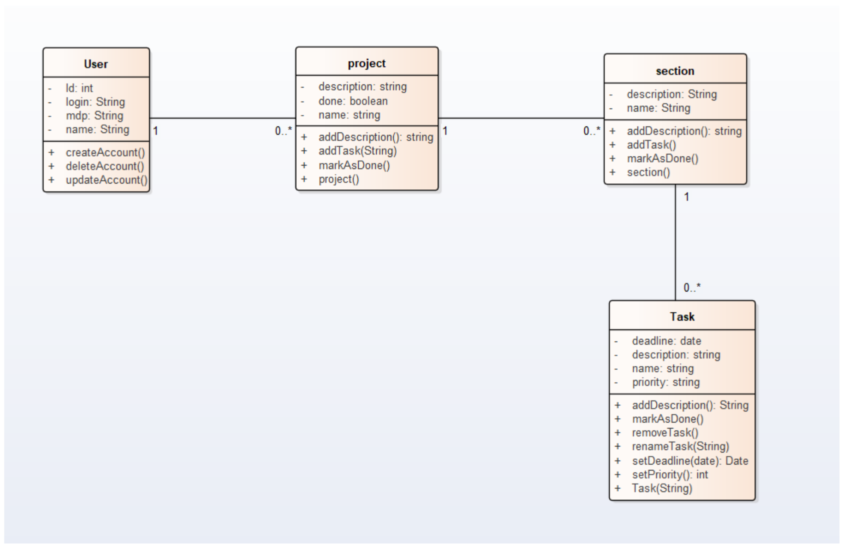


Figure 2: Schema

Furthermore, after ***coding the schema*** we're going to be implementing a simple authentication middle-ware that is going to be for every route of our back-end server. This is what we implemented in our back-end:

- **TypeORM**: We used TypeORM to handle our database connections and queries in a type-safe and organized manner, making it easy to manage our database schema and entities using TypeScript classes.
- **Dotenv**: We used dotenv to load environment variables from a .env file during development and testing, making it easy to configure our application without hardcoding sensitive information like API keys or database credentials in our code.
- **Jsonwebtoken**: We used jsonwebtoken to handle user authentication and authorization by generating and verifying JSON Web Tokens (JWTs) containing user information and permissions, making it easy to secure our API routes and protect sensitive user data.

Here is the middle-ware function "authentify":

```
interface RequestWithUser extends Request {
    headers: any
    user?: any
}

// authentication middleware:
function authentify(req: RequestWithUser, res:
   Response, next: Function) {
    // 1. get token from headers
    // token format: "Bearer <token>"
    const authorization = req.headers['authorization']

    if (!authorization) {
        return res.status(401).send('Unauthorized')
    }

    if (authorization.split(' ').length !== 2) {
        return res.status(401).send('Unauthorized')
    }

    if (authorization.split(' ')[0] !== 'Bearer') {
        return res.status(401).send('Unauthorized')
    }

    const token = authorization.split(' ')[1]

    try {
        // 2. validate token
        const secret = process.env.JWT_SECRET || '
            secret'
        const payload = jwt.verify(token, secret)

        // 3. if valid, continue + set req.user
        req.user = payload
        next()
    } catch (e) {
        console.log(e);
        return res.status(401).send('Unauthorized')
    }
}
```

Then, we're going to create a route that requires a POST request, that handle user authentication requests: Once a user with the correct credential logs in, we generate a token for his localStorage, else we send "Unauthorized."

3.2 Front-End Development

In this section, we're using React TypeScript with certain libraries for each specific functionality like:

- **Axios**: We used the Axios library for making HTTP requests to our backend API.
- **Jwt-decode**: We used the jwt-decode library for decoding JSON Web Tokens (JWT) in our client-side code.
- **React-router-dom**: We used the React Router library for handling client-side routing in our application.
- **Styled-components**: We used the Styled Components library for writing CSS-in-JS code in our React components.

Initially, we prepared our login, register pages and tasks pages. For the login page, Axios helped us to send HTTP requests to the backend server to authenticate the user and retrieve the JWT token. For the tasks page, Axios helped us to fetch the tasks data from the backend server and update the task status using HTTP requests.

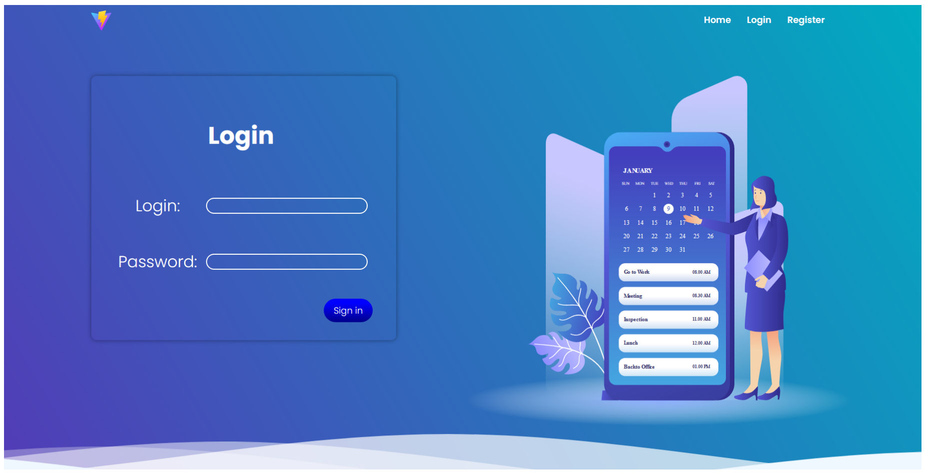


Figure 3: Login Page

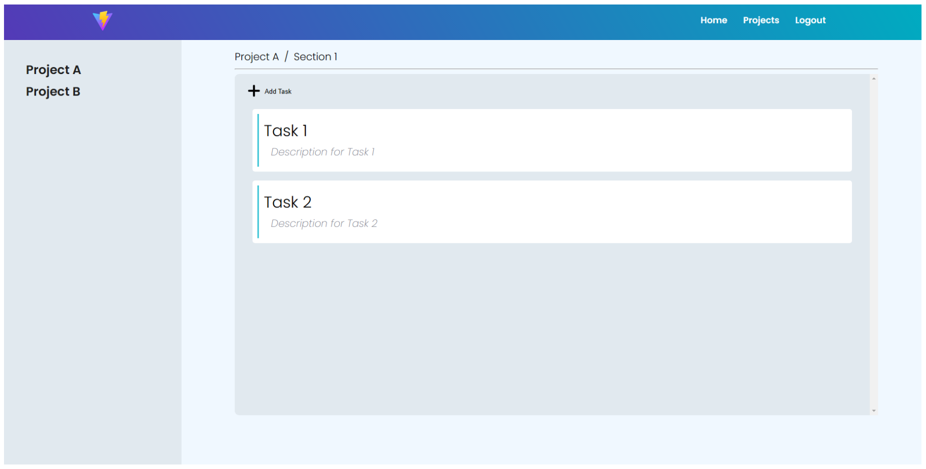


Figure 4: Tasks Page

3.3 Configuring Nginx and DockerFile for Front-end

We used Nginx on our front-end image and configured it to serve our React application. We also made sure to set it up as a reverse proxy to redirect requests to the backend API. In order to configure it, we created this nginx.conf configuration file :

```
upstream app {
    server app:5000;
}

server {
    listen 80;
    server_name localhost;

    location /api/ {
        proxy_pass http://app/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
            $proxy_add_x_forwarded_for;
        proxy_cache_bypass $http_upgrade;
    }

    # Handling Error Pages
    error_page   500 502 503 504  /50x.html;
    location = /50x.html/ {
        root   /usr/share/nginx/html;
    }

    # Handling / pages to our index.html from our
       front-end build
    location / {
        root /usr/share/nginx/html;
        index index.html;
        try_files $uri /index.html;
    }
}
```

Once we had our Nginx configuration working locally, we created a Dockerfile for our frontend. In this Dockerfile, we started with a base image that included Node.js, and then copied our React application into the image.

Next, we added the Nginx installation and configuration steps to our Dockerfile. We installed Nginx and copied our Nginx configuration file into the image.

Finally, we made sure that our Dockerfile included instructions to start Nginx when the container was started.

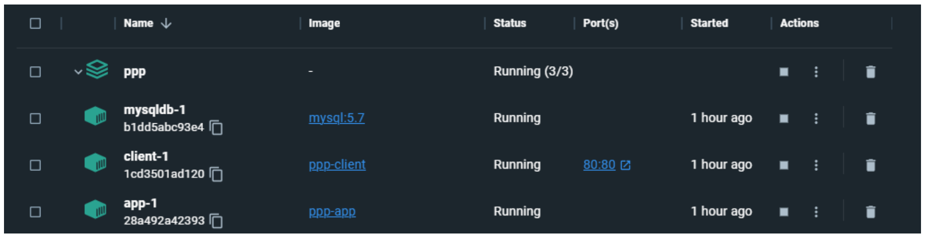
With these steps completed, we were able to build and run a Docker image for our frontend that included Nginx as a lightweight HTTP server and reverse proxy.

```
FROM node as build

WORKDIR /app

COPY package.json ./

COPY package-lock.json ./

RUN npm install

COPY . .

RUN npm run build

FROM nginx

COPY ./nginx.conf /etc/nginx/conf.d/default.conf

COPY --from=build /app/dist /usr/share/nginx/html
```

3.4 Building Docker Images with Docker Compose

The frontend and backend applications were containerized using Docker images, and the MySQL database was also run as a container. We defined the configuration of these containers in a YAML file called docker-compose.yml.

With Docker Compose, we were able to define and manage the relationships between these containers, such as linking the back-end and MySQL containers, and exposing the front-end container to the host machine's port using Nginx configuration for easy access.



Name	Image	Status	Port(s)	Started
ppp	-	Running (3/3)		
mysqldb-1 b1dd5abc93e4	mysql:5.7	Running		1 hour ago
client-1 1cd3501ad120	ppp-client	Running	80:80	1 hour ago
app-1 28a492a42393	ppp-app	Running		1 hour ago

Figure 5: Docker Containers

Using Docker Compose allowed us to easily manage and deploy our application, as we were able to spin up the entire environment with a single command. It also ensured that our application would run consistently across different machines, as we were able to define the dependencies and configuration for each container in a single file.

```yaml
version: "3.8"

networks:
  app-tier:
    driver: bridge

services:
  mysqldb:
    image: mysql:5.7
    expose:
      - "3306"
    networks:
    - app-tier
    environment:
      MYSQL_DATABASE: ppp
      MYSQL_USER: user
      MYSQL_PASSWORD: passworduser
      MYSQL_ROOT_PASSWORD: passwordroot

  app:
    depends_on:
      - mysqldb
    build:
      context: ./serverORM/
      dockerfile: Dockerfile
    command: bash -c 'while !</dev/tcp/mysqldb/3306;
       do sleep 1; done; npm start'
    ports:
      - "5000"
    networks:
      - app-tier
    volumes:
      - .:/app
      - '/app/node_modules'

  client:
    depends_on:
      - app
    build:
      context: ./client/
      dockerfile: Dockerfile
    ports:
      - 80:80
    networks:
      - app-tier
    volumes:
      - .:/app
      - '/app/node_modules'
```

4 Conclusion

To summarize, we developed a full-stack web application with React, Express, and MySQL. To enhance the performance of our application, we utilized Nginx as a load balancer and a reverse proxy. We also leveraged Docker and Docker Compose to create images and containers for our frontend, backend, and MySQL. These tools helped us streamline the deployment process and improve the scalability of our application. By using technologies such as Axios, JWT and TypeORM, we were able to build a robust and secure application with a clean and modern user interface.