KUZEY KIBRIS KAMPÜSÜ
ODTÜ
METU
NORTHERN CYPRUS CAMPUS

CNG 495
CLOUD COMPUTING

FALL 2025

CAPSTONE PROJECT PROPOSAL

Yıldız Alara Köymen - 2453389

Burak Mirac Dumlu - 2584944

Emir Canlı - 2637544

Table of Contents

1. DynaZOR: Dynamic Rendezvous & Scheduling System

DynaZOR is a cloud-based scheduling and rendezvous system designed to automate appointment management between users. The goal is to eliminate manual scheduling errors and reduce communication delays by using AWS services for storage and notifications. The system dynamically adjusts appointment times when changes occur and instantly notifies users through cloud based alerts, ensuring efficient coordination and improved time management.

One standout aspect is its clever rescheduling. If an appointment is cancelled or a dispute emerges, the system immediately intervenes. It discovers a new open time that is convenient for everyone depending on their preferences and schedules. This eliminates the need for users to manually calculate new times. The system also has a strong notification system. When an appointment is scheduled, altered, or cancelled, it provides immediate notifications to all necessary parties. This guarantees that everyone is always up to date, reducing miscommunication and missing meetings. If more information is required for an appointment, such as paperwork or specific details, the system will send a notification with contact information so that everything can be handled without having to send several emails or phone calls.

The system will use the collected appointment data to perform simple analytics and show each user when they are most likely to attend. Using Python pandas library, the system can calculate attendance probabilities for each day and time of the week based on past behavior. For example, analyzing whether a user's schedule is usually full in the morning or weekend appointments. These insights will be displayed to users as a small dashboard or chart, allowing them to see patterns like "You are most likely to be full on Tuesdays at 10 AM." This way, users can adjust their future habits and create appointment times that fit their routines more reliably.

DynaZOR improves efficiency and reliability by combining smart scheduling and transparent communication. It eliminates administrative labor, improves time management, and guarantees that all parties are prepared and informed for their appointments. The project will be implemented as a Software as a Service (SaaS) solution. Users will be able to access the system through a web interface without needing to install or manage any local software.

2. Implementation

The implementation of DynaZOR combines backend and frontend components to form a functional scheduling system. The backend manages data and storage, while the frontend, built with React, handles user interaction and display. AWS S3 provides cloud-based data storage. Using Flask for the backend and React for the frontend as separate servers ensures a clear separation of concerns and simplifies development.

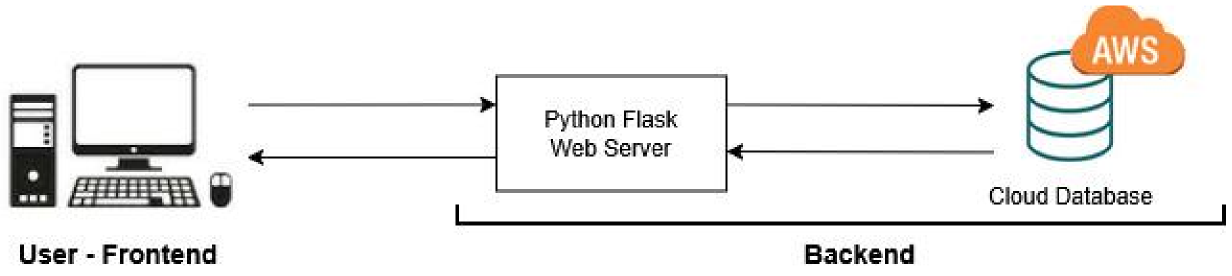


Figure 1: System architecture of DynaZOR

2.1 Backend

The backend of DynaZOR will be developed using Flask. The team is already experienced with Python and Flask, which makes it a suitable and efficient choice for implementing the core application logic. Flask will manage appointment handling, user information, and the dynamic rescheduling operations of the system.

The backend will implement RESTful endpoints to handle client requests and exchange data in JSON format. After processing, it will use the AWS SDK (Boto3) to interact with Amazon S3 for data storage. AWS S3 (Simple Storage Service) is a scalable and reliable cloud storage service that allows storing and retrieving data objects in the form of files. In the DynaZOR system, AWS S3 will be used to store structured data such as JSON files containing appointment records, user availability, and system logs.

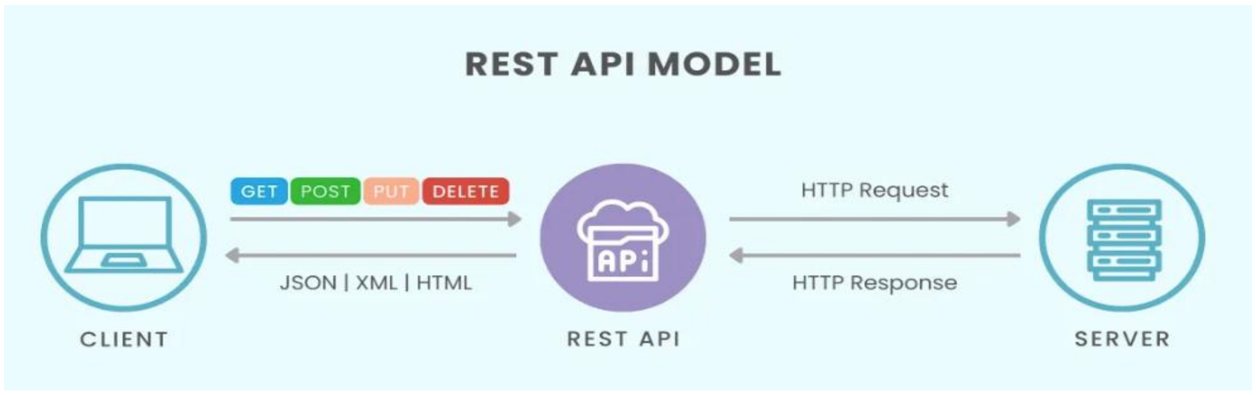


Figure 2: Architecture of REST API model (Yu, 2023)

Flask will use Boto3 to read, write, and update data objects within S3 buckets. Each operation, such as adding or updating an appointment, will correspond to creating or modifying a file in S3. This setup provides a scalable backend structure suitable for the scope of this project.

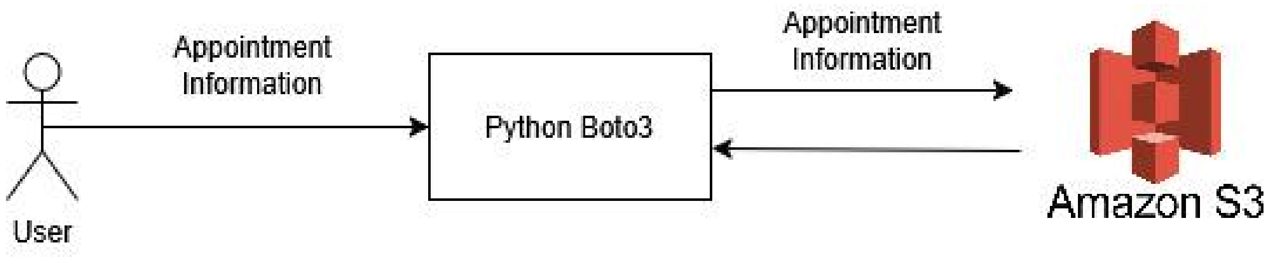


Figure 3: Data flow between the user, Python Boto3, and Amazon S3

2.2 Frontend

The frontend of DynaZOR will be developed using React. React is chosen because it allows the reuse of components, which helps in building features like calendars, time-slot grids, and appointment lists efficiently. The team is already familiar with JavaScript, so using React simplifies the development process and reduces the learning curve.

React's component-based structure also makes it easier to maintain a clear and organized codebase. It supports efficient updates when appointment data changes, which is important for DynaZOR's dynamic scheduling and rescheduling features.

2.3 Cloud Notification System

Amazon Simple Notification Service (SNS) will be integrated into the Flask backend to handle user notifications for rescheduled and cancelled appointments. Whenever an appointment is modified or cancelled, Flask will trigger an SNS publish event that sends a notification to the corresponding users. This ensures that users are promptly informed about any changes in their schedules without manually checking the system. Because this approach is scalable, we are allowed to include other notifications based on the development evolution of the system. SNS topics will be configured to deliver messages through email. AWS SDK (Boto 3) will be used here to access Amazon SNS services.

3. Diagrams

3.1 Data-Flow Diagram:

The data flow diagram illustrates how information moves through the DynaZOR system. Its purpose is to give a clear overview of how data is processed and transferred within the system.

3.1.1 Level-0:

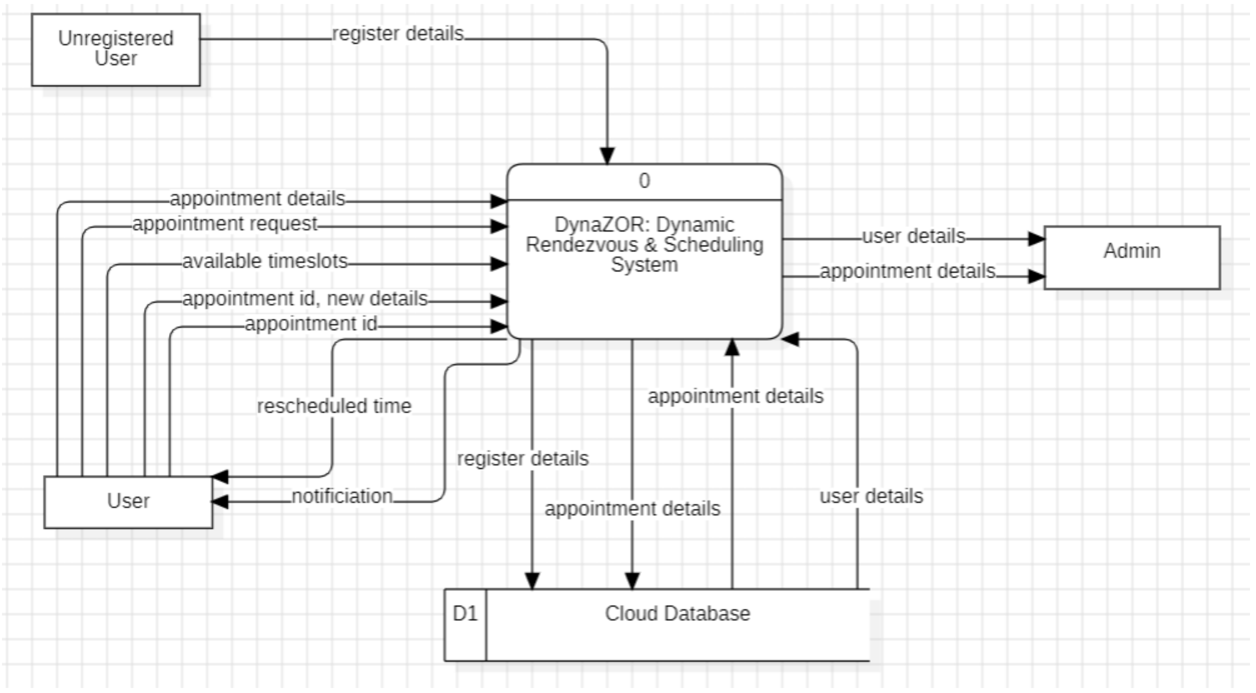


Figure 4: Level-0 Data-Flow Diagram of DynaZOR

3.1.2 Level-1:

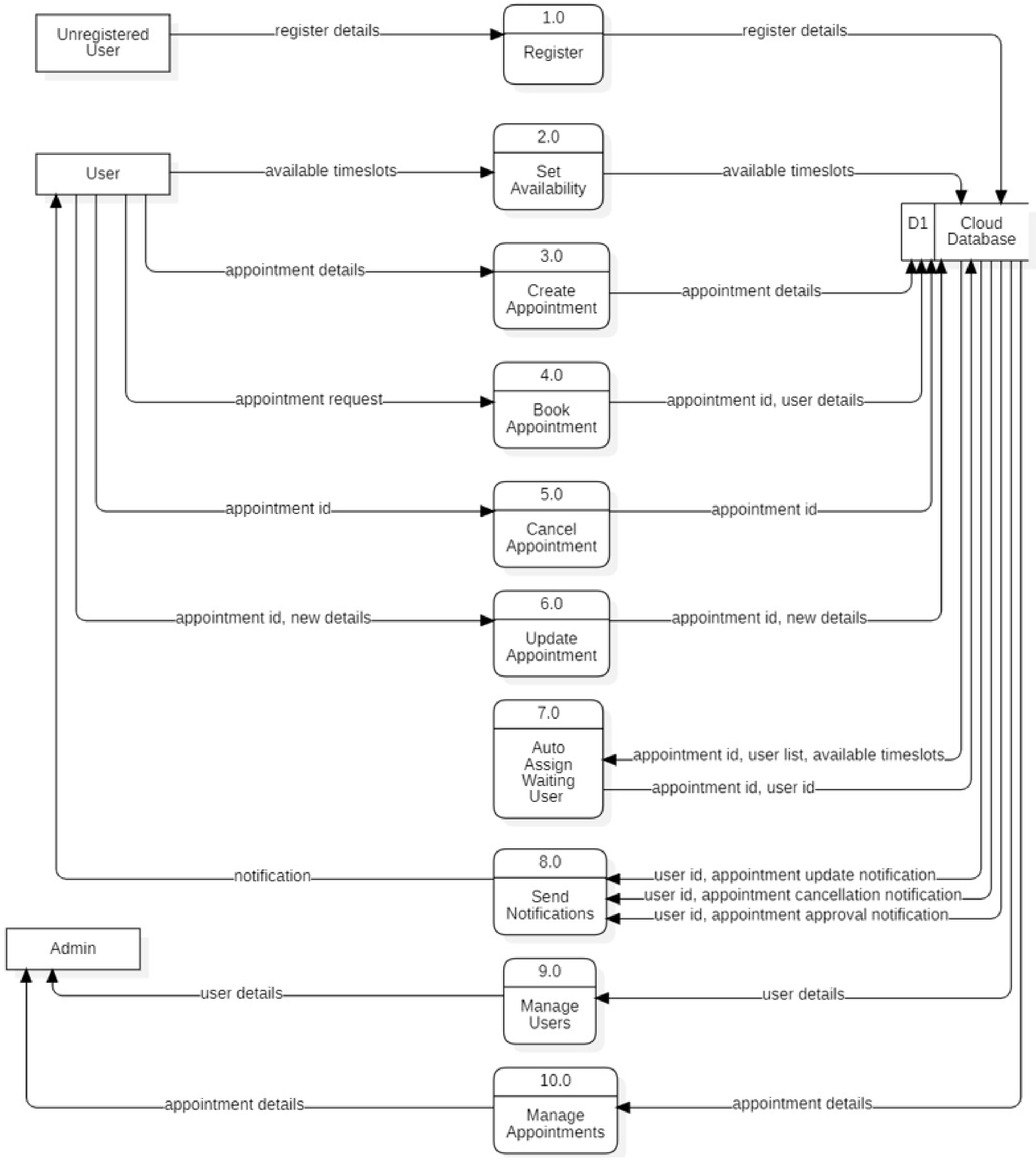


Figure 5: Level-1 Data-Flow Diagram of DynaZOR

3.2 Use case Diagram:

The use case diagram illustrates the main interactions between users and the DynaZOR system. Its purpose is to show the system's core functionalities, such as creating, updating, rescheduling, and cancelling appointments, and how different user roles interact with these features.

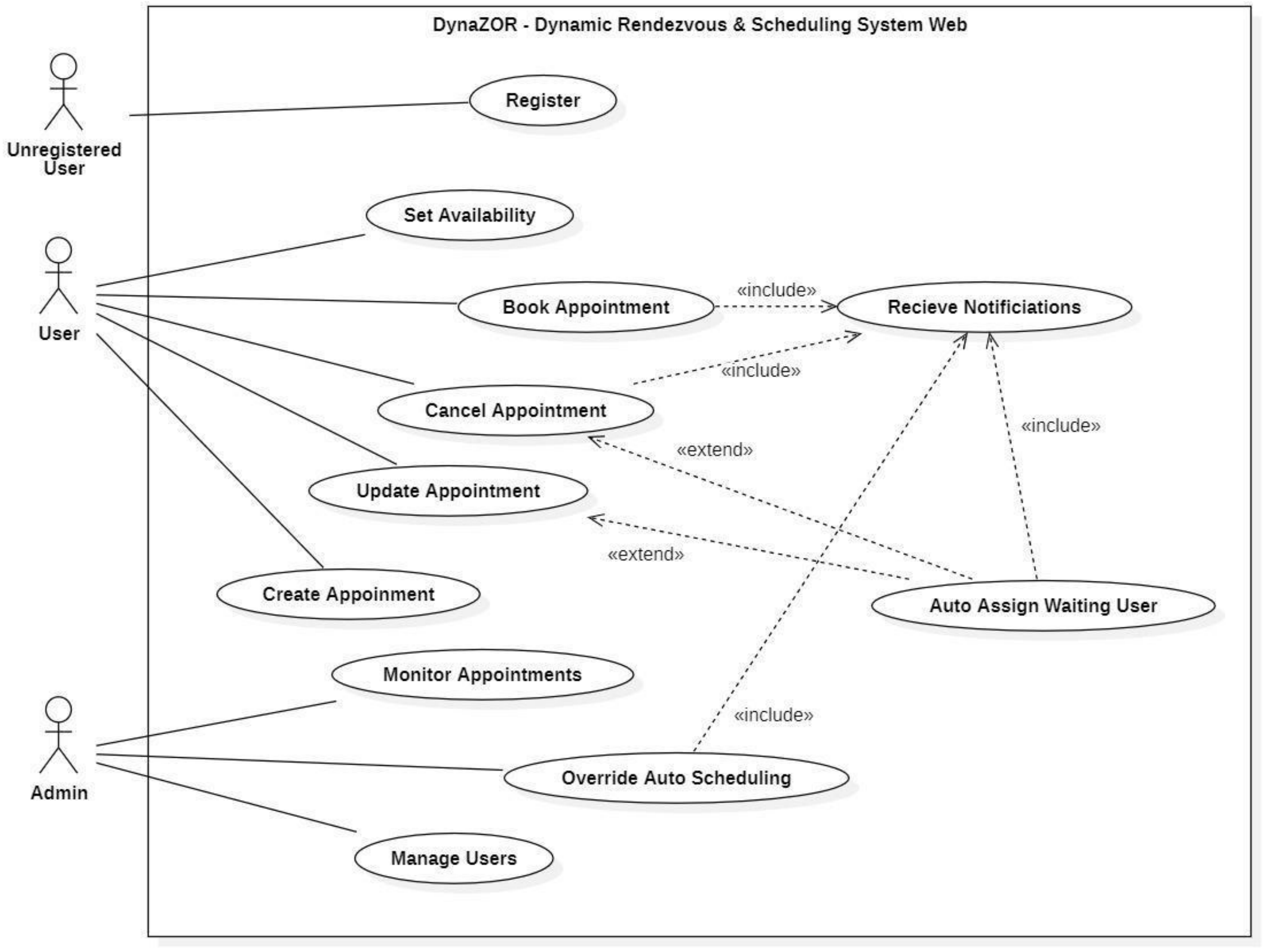


Figure 6: Use case Diagram of DynaZOR

4. Data Types

The system primarily handles **JSON data** for user and appointment information.

Examples:

- appointment.json: {user_id, provider_id, date, time, location}
- users.json: {name, email, preferred_hours}

5. Computation

The backend performs key computations related to scheduling optimization and conflict detection. When a new appointment is created or an existing one is rescheduled, the system calculates available time slots based on all users' preferences and existing bookings. The backend handles these computations before updating the cloud storage.

6. Expected Contribution

- **Yıldız Alara Köymen:** Frontend development with React, Flask API design.
- **Burak Mirac Dumlu:** Backend AWS database integration.
- **Emir Canlı:** Backend SNS integration, cloud setup.

7. References

Amazon Simple Notification Service Examples. (n.d.). Retrieved from AWS SDK for JavaScript: https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/sns-examples.html

Boto3 documentation. (n.d.). Retrieved from amazon aws: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

Flask Documentation. (n.d.). Retrieved from Flask: https://flask.palletsprojects.com/en/stable/

React. (n.d.). Retrieved from React: https://react.dev/

Yu, I. (2023, October 31). *[Part 2] REST API components & How to read them*. Retrieved from SkipLevel: https://www.skiplevel.co/blog/part-2-rest-api-components-how-to-read-them

CNG 495
CLOUD COMPUTING

FALL 2025

CAPSTONE PROJECT

PROGRESS REPORT

Yıldız Alara Köymen - 2453389

Burak Mirac Dumlu - 2584944

Emir Canlı - 2637544

Table of Contents

1. Project Review & Changes

This section describes any changes made during the development process and gives an overall evaluation about the current status of the project

1.1. Project overview

While we are in the early stages of the project, we are satisfied with the progress that has been achieved so far. Many aspects of the project we've assessed during the project proposal turned out to be a great fit. The project's potential and possible obstacles have been clarified by the preliminary research, requirement analysis, and development work. Moving forward, we aim to build on this foundation by completing the next stages of development, conducting testing, and integrating additional components in line with the project goals.

1.2. Changes Made

Initially we planned to use S3 to store information of users, their schedules and other data that we would use for the project. Since the communication between frontend and backend happens via json files sent to each other, and S3 being a file based storage service, we chose S3 at start. However, we later realised that it would be hard to work on files, since file reading and writing could potentially bring lots of errors. Also, we thought it wouldn't be as efficient as a database on higher traffic. Therefore, instead of using the file based cloud storage service Amazon S3, we used a proper cloud database: Amazon RDS (Relational Database Service) for the backend data connections. RDS supports many database engines but for our project we chose Microsoft SQL.

2. Project Timeline

2.1. Planned Timeline

The project did start on 7th November and the Progress Report is to be done by 1st December. The milestones achieved according to the timeplane planned weekly.

- Week 1(3 November - 9 November) : Proposal Review & Clarification
- Week 2(10 November - 16 November) : Research & Literature Review
- Week 3(17 November - 23 November) : Implementation
- Week 4(24 November - 30 November) : Early Testing

Although this weekly planning helped prioritizing the main phase we focused on, we did not only make implementations on implementation week or only testing on test week, but rather we focused on it.

2.2. Milestones Achieved

In the following part, the milestones achieved are explained week by week briefly to be later explained in detail. Thanks to work load distribution we are able to research, implement and test backend,frontend and connections simultaneously. Thus the milestone achievements is simultane teamwork of collaborated team members.

2.2.1. Week 1(3 November - 9 November) : Proposal Review & Clarification

This week, project context and proposal has been reviewed by the supervisor and unclarified parts have been cleared. Simple changes according to the suggestions have been made on both the proposal report and overall project.

2.2.2. Week 2(10 November - 16 November) : Research & Literature Review

In this week, the similar software/programs were reviewed to get an idea and a path to how to implement&deploy our project. We have done research and brainstormed on our early decision about the tools and/or services we wish to use and decided if we wish to change or continue with the same tool. Additionally, the achievements on the implementation of structure wished decided in this week.

2.2.3. Week 3(17 November - 23 November) : Implementation

In implementation week we started to code our project. As mentioned earlier, the simultaneous development on different branches required early work to be isolated but as soon as parts started to come together, the development carried out to collaborative work.

2.2.4. Week 4(24 November - 30 November) : Early Testing

Testing week was to see the results of our early accomplishments. The major objective was not to see a wanted output but rather if the structure is stable.

3. Completed Work to Date

Although this project is a product of teamwork and collective contribution, each team member mainly worked on dedicated parts of the project that is specified in “Chapter 6: Expected Contribution” part of project proposal.

3.1. Frontend

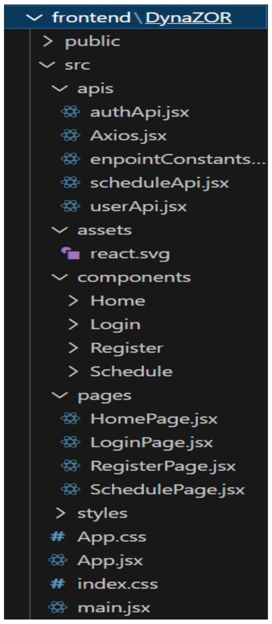
The frontend of the project was developed using React with a fully modular and component oriented approach. The project was created with Vite, which made development fast and smooth, especially during constant UI updates. Using React turned out to be a solid choice because the schedule feature needed a structure where every part of the UI behaves independently. With React components, each schedule cell was built as a separate component, which makes the table easier to manage, update, and extend later.

For handling API communication, Axios was used for the project. A small custom wrapper was created (getAxiosInstance) so that it can be reused while creating each API function. Since the backend was still under development, mock data was used temporarily for the schedule API. Then later, deleted to test the API connection between the frontend and the backend.

Three main pages are currently implemented: Login, Register, and Schedule. Each has its own folder and JSX structure under the components and pages directories. This separation makes the code cleaner and avoids mixing UI logic with page routing. The modular structure of the frontend is prepared so that when the project progresses to other functionalities, they can easily be extended.

Styling was originally planned to be done using plain CSS in a dedicated styles/ folder. However, as the UI grew in complexity-especially with the schedule layout-the project transitioned to using Tailwind CSS. Tailwind significantly reduced repetitive styling, improved consistency across the UI, and sped up layout development. This approach has proven more flexible and maintainable for the long term.

Overall, separating the frontend and backend work among the team has forced the team to communicate and ensure that everyone is on the same page. The frontend developer had to understand the backend to be able to create the intended vision. The modular file separation of the frontend and the use of React create a solid base for future expansion. Reusability of the components was always a focus to make future development easier. This can be clearly seen in the folder structure shown in figure 1 below.



```
frontend\DynaZOR
  > public
  v src
    v apis
        authApi.jsx
        Axios.jsx
        enpointConstants...
        scheduleApi.jsx
        userApi.jsx
    v assets
        react.svg
    v components
      > Home
      > Login
      > Register
      > Schedule
    v pages
        HomePage.jsx
        LoginPage.jsx
        RegisterPage.jsx
        SchedulePage.jsx
    > styles
      App.css
      App.jsx
      index.css
      main.jsx
```

Figure 1: Frontend folder structure

3.1.1 File separation:

The apis folder holds 5 different files:

3.1.1.1. **authApi:** Contains the API functions for login. For now, it only includes a login function that sends the credentials (email and password) to the backend.

3.1.1.2. **Axios:** Initializes the Axios instance needed for all API functions.

3.1.1.3. **endpointConstants:** Contains the constants that the API functions use for the backend endpoints. Currently, it includes: User/list, User/add, Schedule/list, and Schedule/add. This approach makes expansion easier, as new endpoints can simply be added here.

3.1.1.4. **scheduleApi:** Contains the add and list API functions for schedules.

3.1.1.5. **userApi:** Contains the add and list API functions for users.

The components folder holds the components necessary for the home page, login page, register page, and schedule page. The top level parent component for each of these is then rendered inside its corresponding page, which can be seen in the pages folder. Lastly, the styles folder contains the CSS files used for these pages.

3.1.2. DynaZOR Login Page:

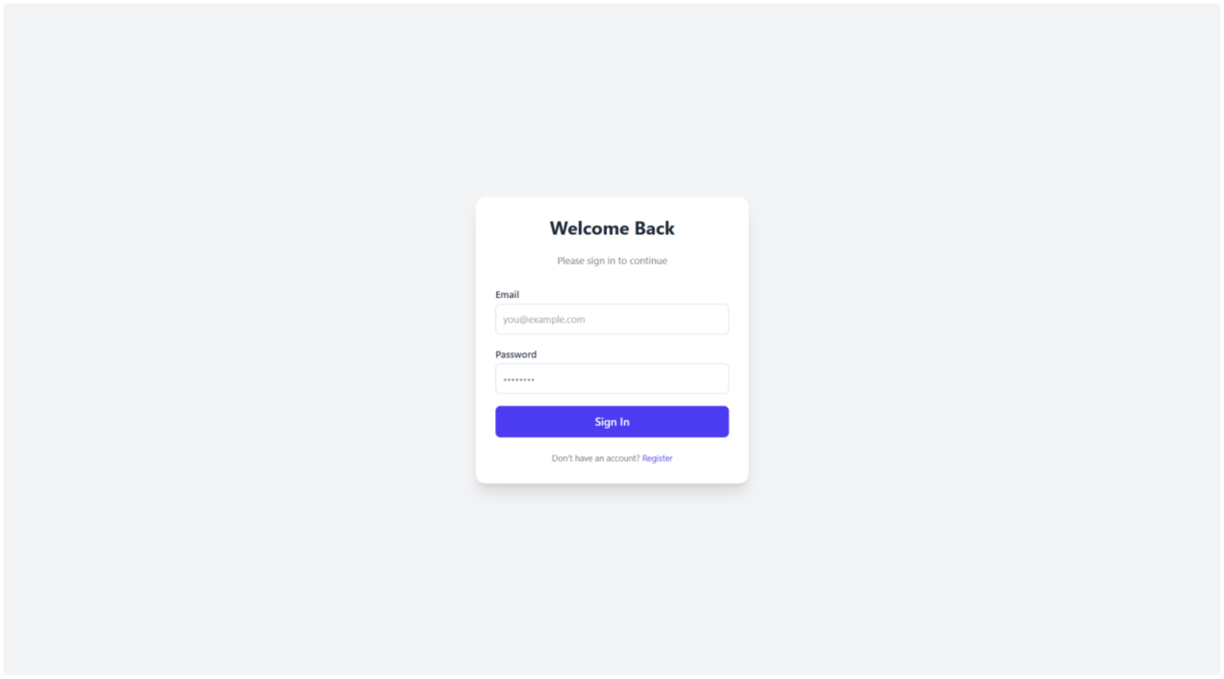


Figure 2: Login page

The login page was developed first because it forms the entry point to the system and many of its elements could later be reused when building the register page. If the user doesn't have an account, they can click the "Register" text underneath and be routed to the register page. The user is required to provide an email and a password, and the frontend performs basic validation to ensure that the fields are not left empty and that the email format is reasonable. Once the user submits the form, the values are placed into a simple JavaScript object, converted into JSON, and sent to the backend using the login API function.

This project uses a simplified authentication flow. Instead of generating persistent session tokens, the backend simply verifies the provided credentials and returns a success or failure response. Based on this response, the frontend determines whether the login attempt is valid. In the next stage of development, when a user successfully logs in, the application will temporarily store minimal information such as the username to personalize the UI during the session. This feature has not yet been implemented but is planned for the upcoming development phase.

The login page also incorporates several UI and UX improvements to make the interaction smoother. Tailwind CSS was used to implement a clean, responsive interface with consistent spacing, modern input styles, and visual feedback such as loading states and error messages. These enhancements are user friendly and are added to improve usability. Even though the authentication system is intentionally lightweight due to the scope of the project, the structure of the login page mirrors real world design principles and provides a clean base for future expansion if needed.

3.1.3. DynaZOR register page:

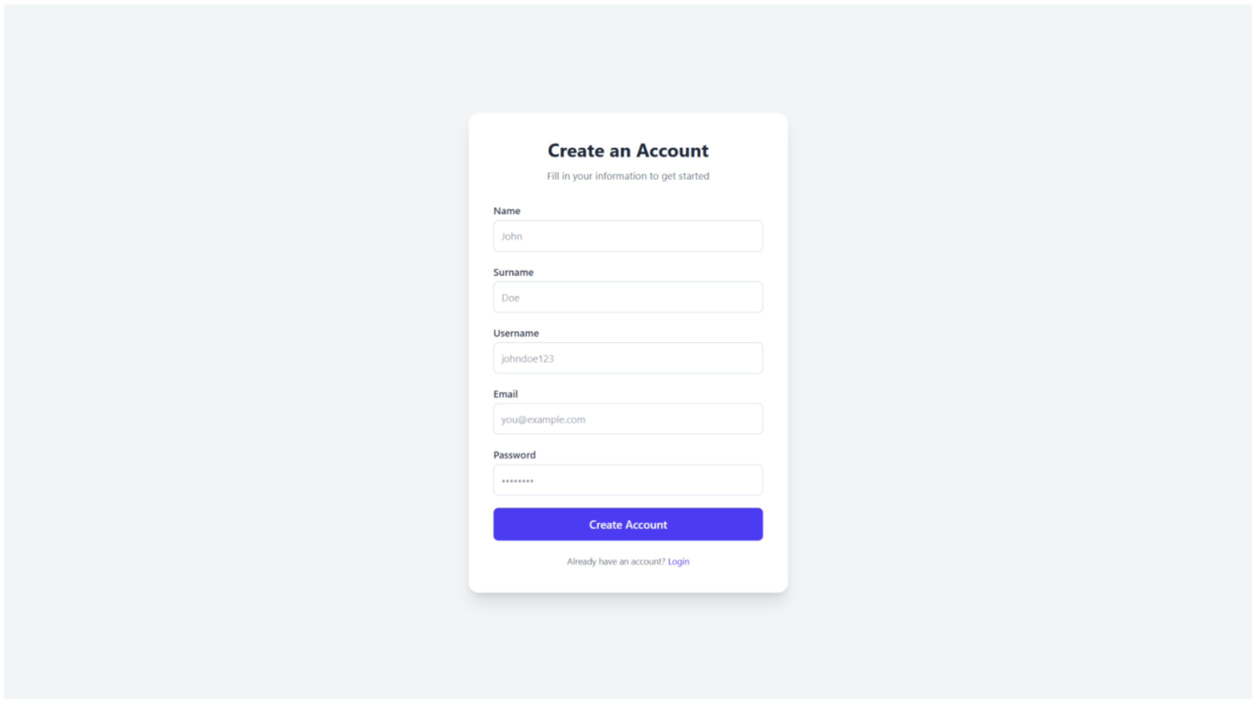


Figure 3: Register page

The register page was developed after the login page and builds upon many of the components and layout patterns already established in the authentication flow. Unlike the login page that only has two fields, the register page requires additional inputs such as name, surname and username to collect the information needed to create a new user profile. Basic validation is also performed on the frontend to ensure that all required fields are filled in, and that the data entered follows expected formats.

When the user submits the form, the collected values are placed into a JSON object and sent to the backend through the registration API function. The API currently performs essential checks to confirm whether the user was successfully created.

The visual layout of the register page follows the same design language as the login page to maintain a consistent look across the authentication flow. Tailwind CSS was again used to create a clean and responsive interface. Similar to the login page, this stage of the project prioritizes establishing a solid functional and stylistic foundation, which will allow additional improvements such as real time validation feedback or dynamic UI updates that will be integrated smoothly in future development phases.

3.1.4. DynaZOR schedule:

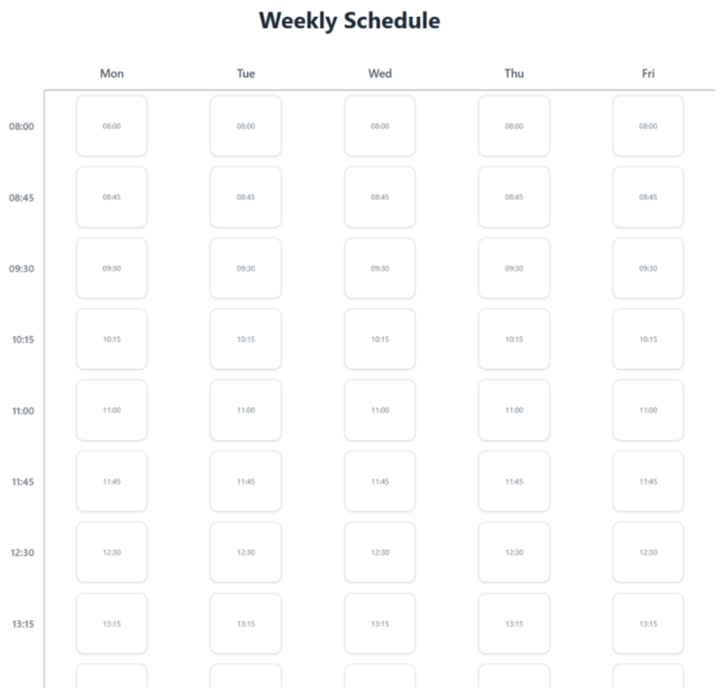


Figure 4: Schedule page

The schedule page is the core functional component of the DynaZOR system, as it provides the interface where users can interact with their weekly timetable. The schedule layout is constructed using a fully modular approach, where each time slot is represented as a separate React component. This component based structure allows each cell to behave independently which makes the overall table easier to update, toggle, and expand as the project evolves.

The page is built using a modular component structure, where each time slot is represented as an independent React component. This approach creates a baseline that can later support richer interactions such as editing, assigning courses, or marking availability. While the basic click toggle behavior for marking a cell as “Busy” has been implemented, the full set of planned features such as storing user-specific schedules, displaying course names dynamically, or enabling different user roles will be added in the upcoming stages of development.

Styling for the schedule page is still in progress. Tailwind CSS has been applied to create an initial grid structure, but the visual design has been intentionally kept simple while the core logic and API connection were prioritized. Improvements such as resizing cells, refining alignment, color coding entries, and enhancing the overall layout will be completed once the functional requirements are fully implemented. By focusing first on data flow and component behavior, the project ensures that future UI enhancements can be added smoothly without restructuring the underlying code.

3.2. Backend

The backend development period started with assessment of data that should be kept on the database. The main needs are decided by the data-flow diagram and use case diagram. With this starting point, the implementation has been started and the other needs and changes have been made along the way.

Before starting the implementation to rather ease the job, the relational database has been created to ease the implementation.

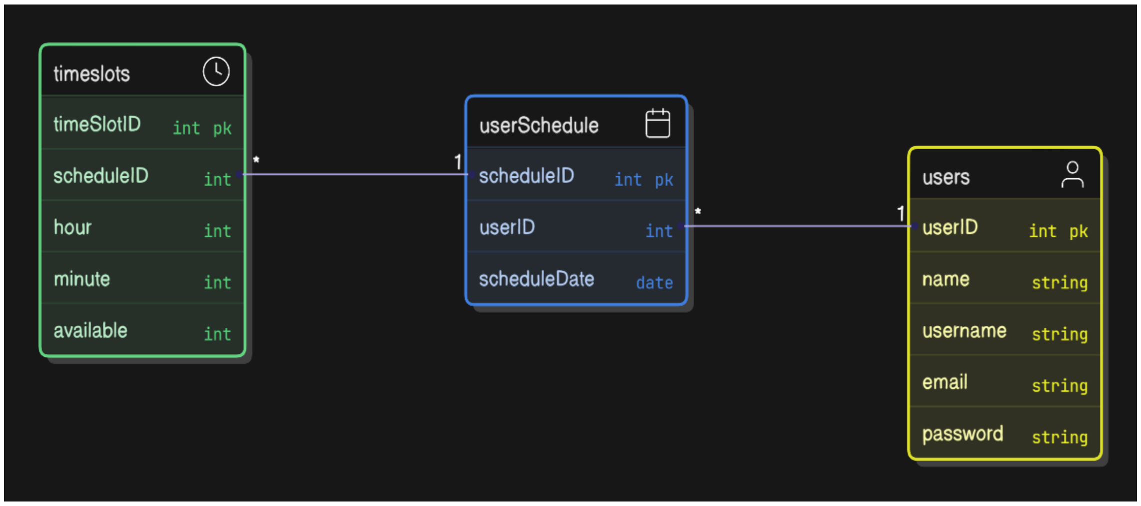
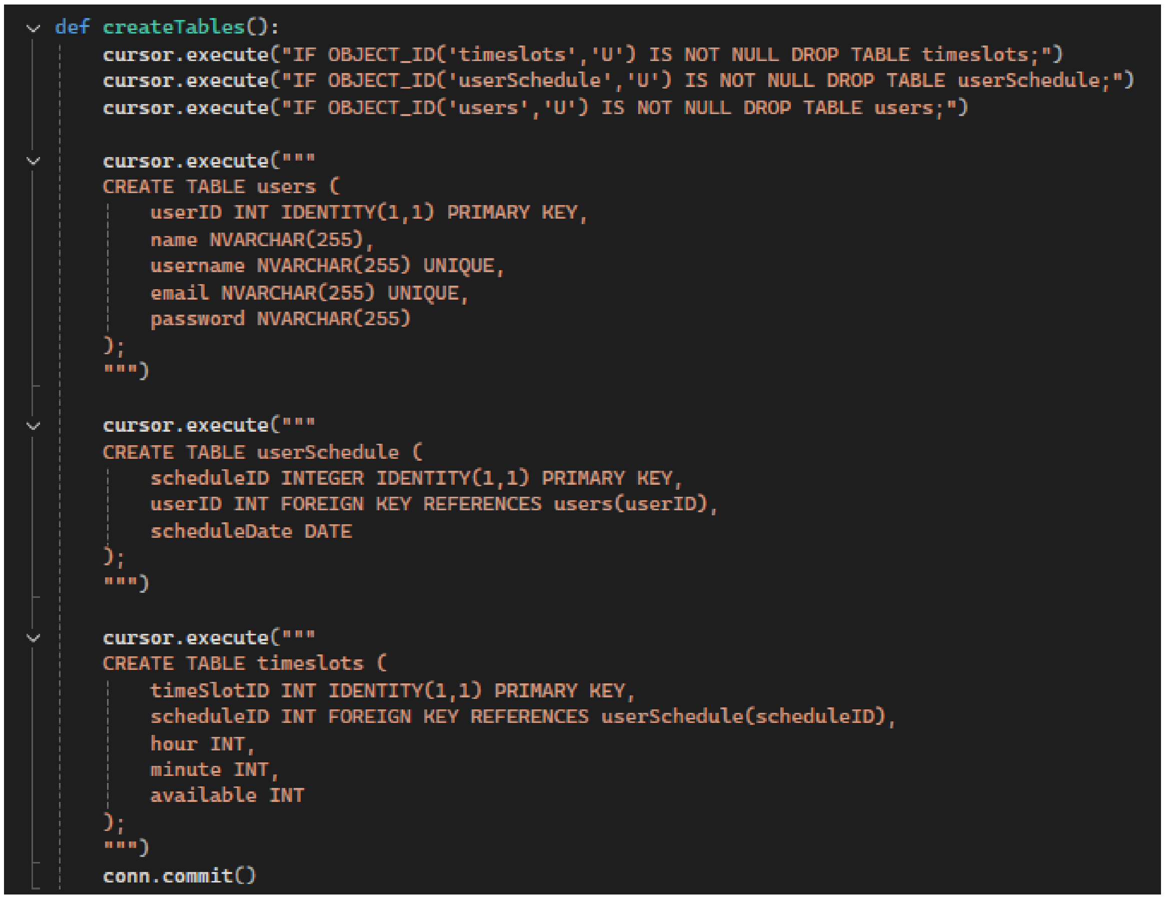


Figure 5: Relational database

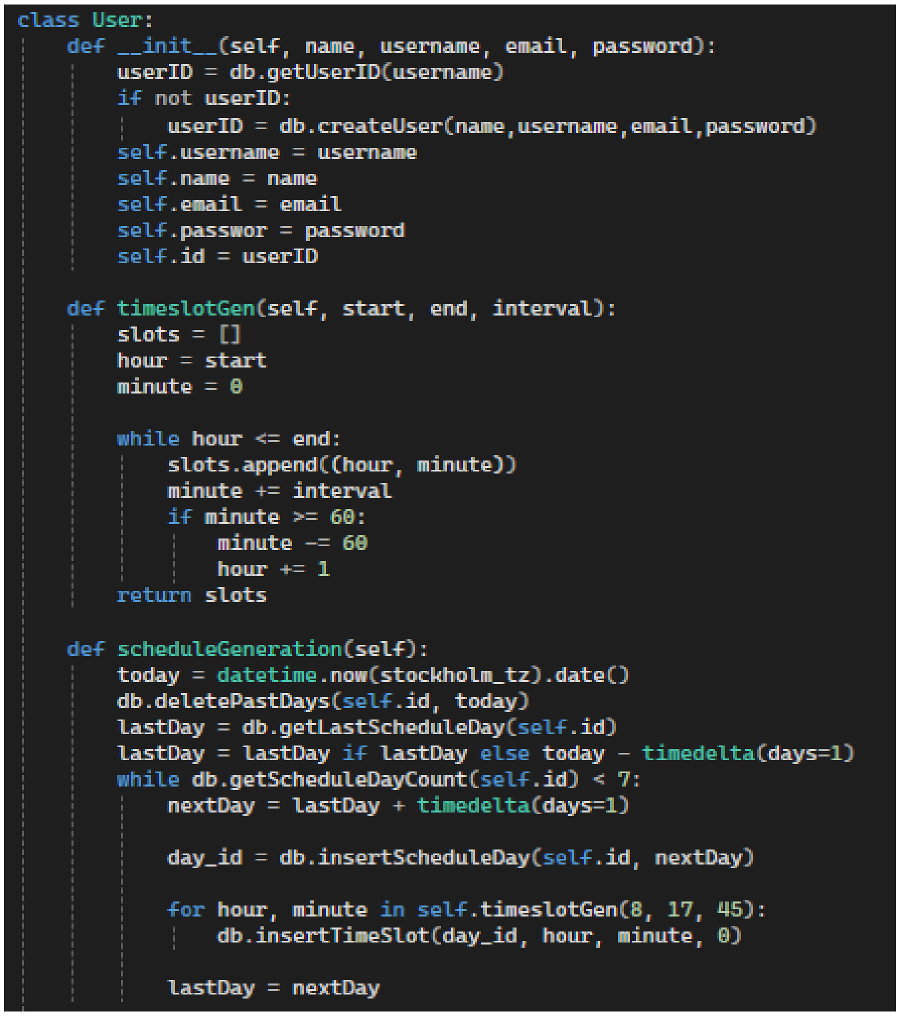
From this, the local database creation made:



```
def createTables():
    cursor.execute("IF OBJECT_ID('timeslots','U') IS NOT NULL DROP TABLE timeslots;")
    cursor.execute("IF OBJECT_ID('userSchedule','U') IS NOT NULL DROP TABLE userSchedule;")
    cursor.execute("IF OBJECT_ID('users','U') IS NOT NULL DROP TABLE users;")

    cursor.execute("""
    CREATE TABLE users (
        userID INT IDENTITY(1,1) PRIMARY KEY,
        name NVARCHAR(255),
        username NVARCHAR(255) UNIQUE,
        email NVARCHAR(255) UNIQUE,
        password NVARCHAR(255)
    );
    """)

    cursor.execute("""
    CREATE TABLE userSchedule (
        scheduleID INTEGER IDENTITY(1,1) PRIMARY KEY,
        userID INT FOREIGN KEY REFERENCES users(userID),
        scheduleDate DATE
    );
    """)

    cursor.execute("""
    CREATE TABLE timeslots (
        timeSlotID INT IDENTITY(1,1) PRIMARY KEY,
        scheduleID INT FOREIGN KEY REFERENCES userSchedule(scheduleID),
        hour INT,
        minute INT,
        available INT
    );
    """)
    conn.commit()
```

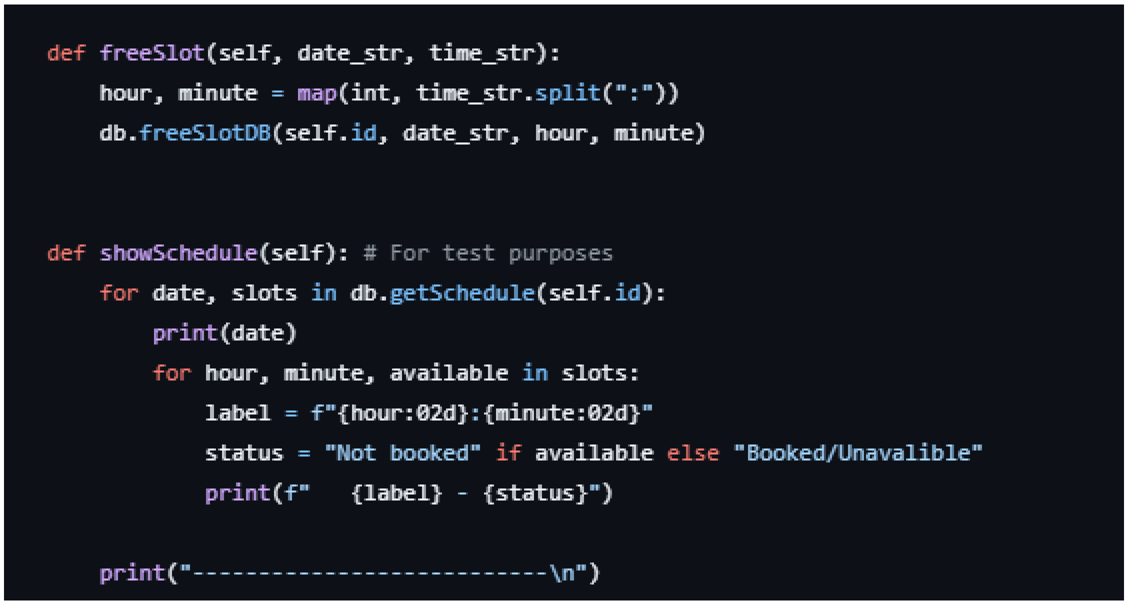
Figure 6: Database creation code

Because the project mainly will run on the user, and the operations considered to be user operations, object oriented approach considered. And functions needed to be operated under the user are created.



```python
class User:
    def __init__(self, name, username, email, password):
        userID = db.getUserID(username)
        if not userID:
            userID = db.createUser(name,username,email,password)
        self.username = username
        self.name = name
        self.email = email
        self.passwor = password
        self.id = userID

    def timeslotGen(self, start, end, interval):
        slots = []
        hour = start
        minute = 0

        while hour <= end:
            slots.append((hour, minute))
            minute += interval
            if minute >= 60:
                minute -= 60
                hour += 1
        return slots

    def scheduleGeneration(self):
        today = datetime.now(stockholm_tz).date()
        db.deletePastDays(self.id, today)
        lastDay = db.getLastScheduleDay(self.id)
        lastDay = lastDay if lastDay else today - timedelta(days=1)
        while db.getScheduleDayCount(self.id) < 7:
            nextDay = lastDay + timedelta(days=1)

            day_id = db.insertScheduleDay(self.id, nextDay)

            for hour, minute in self.timeslotGen(8, 17, 45):
                db.insertTimeSlot(day_id, hour, minute, 0)

            lastDay = nextDay
```

Figure 7: User class



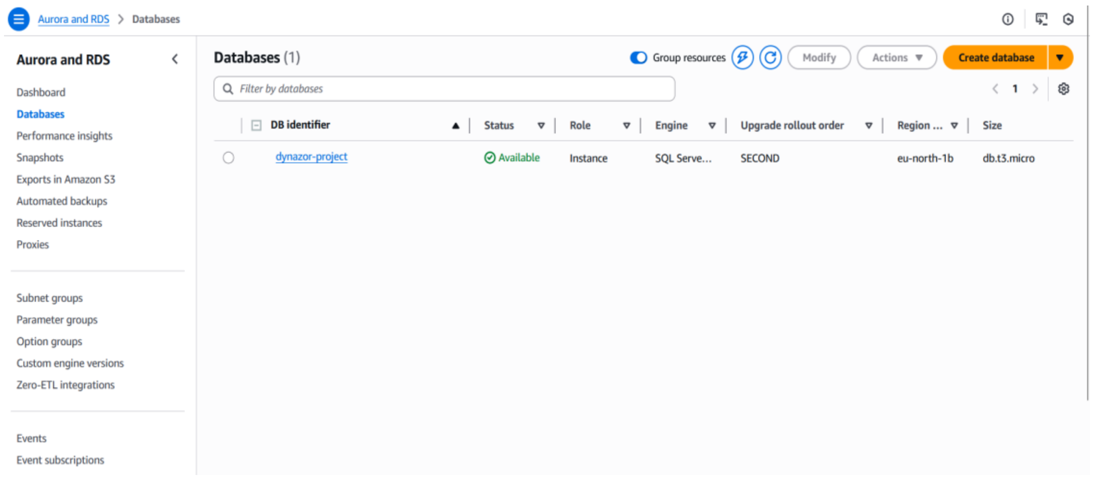
```python
def freeSlot(self, date_str, time_str):
    hour, minute = map(int, time_str.split(":"))
    db.freeSlotDB(self.id, date_str, hour, minute)


def showSchedule(self): # For test purposes
    for date, slots in db.getSchedule(self.id):
        print(date)
        for hour, minute, available in slots:
            label = f"{hour:02d}:{minute:02d}"
            status = "Not booked" if available else "Booked/Unavalible"
            print(f"   {label} - {status}")

    print("--------------------------\n")
```

Figure 8: User class continued

Scheduling and appointment algorithms remained as next milestones to better see the results and effects on the front ends.

Database functionality tested via creating a local database first, using tester functions to see if the creation is successful. Functions are created specifically usable by only inserting required parameters thus adapting it to frontend is relatively easy.

After testing it on a local database, we created the database on the Amazon RDS cloud server with the tables we used on our local database.

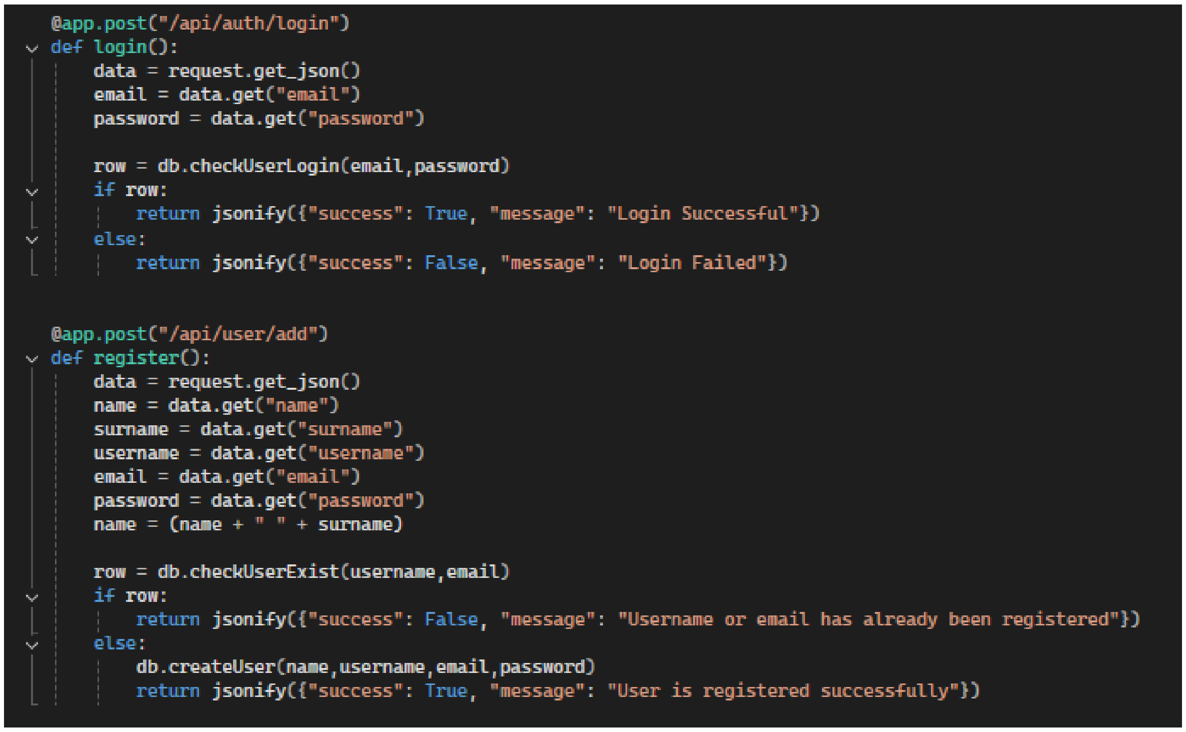


Databases (1)

DB identifier	Status	Role	Engine	Upgrade rollout order	Region ...	Size
dynazor-project	Available	Instance	SQL Serve...	SECOND	eu-north-1b	db.t3.micro

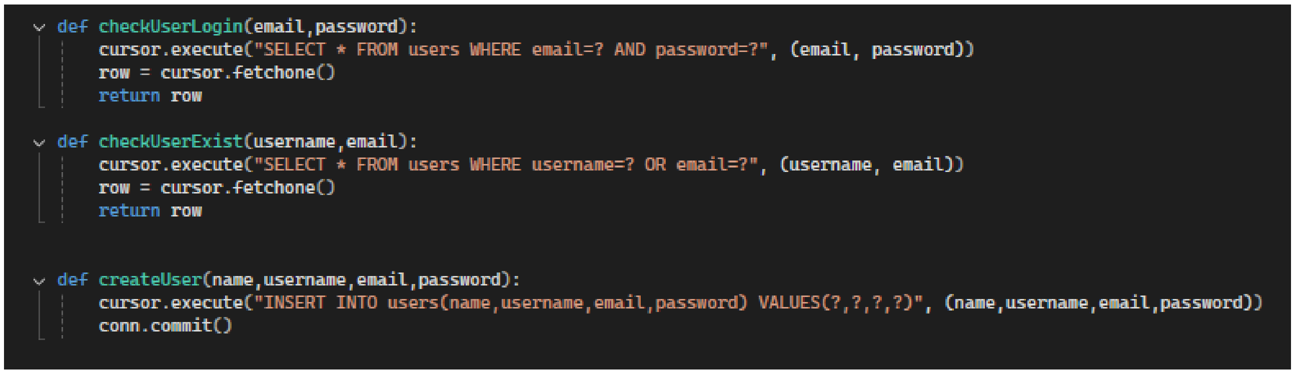
Figure 9: Amazon RDS portal

Then we implemented both login and register functions, Flask validating the operations (checking if the user exists, creating a user in the database etc.) and sending the relevant JSON to the frontend.



```
@app.post("/api/auth/login")
def login():
    data = request.get_json()
    email = data.get("email")
    password = data.get("password")

    row = db.checkUserLogin(email,password)
    if row:
        return jsonify({"success": True, "message": "Login Successful"})
    else:
        return jsonify({"success": False, "message": "Login Failed"})


@app.post("/api/user/add")
def register():
    data = request.get_json()
    name = data.get("name")
    surname = data.get("surname")
    username = data.get("username")
    email = data.get("email")
    password = data.get("password")
    name = (name + " " + surname)

    row = db.checkUserExist(username,email)
    if row:
        return jsonify({"success": False, "message": "Username or email has already been registered"})
    else:
        db.createUser(name,username,email,password)
        return jsonify({"success": True, "message": "User is registered successfully"})
```

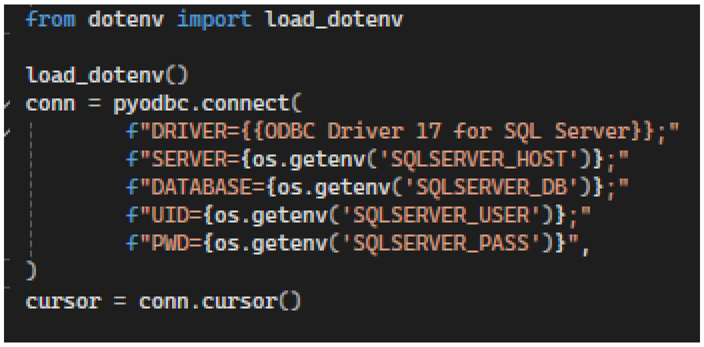
Figure 10: Backend codes for communication with frontend



```
def checkUserLogin(email,password):
    cursor.execute("SELECT * FROM users WHERE email=? AND password=?", (email, password))
    row = cursor.fetchone()
    return row

def checkUserExist(username,email):
    cursor.execute("SELECT * FROM users WHERE username=? OR email=?", (username, email))
    row = cursor.fetchone()
    return row


def createUser(name,username,email,password):
    cursor.execute("INSERT INTO users(name,username,email,password) VALUES(?,?,?,?)", (name,username,email,password))
    conn.commit()
```

Figure 11: Database operation codes

For the database operations, since we now moved onto the cloud database, we had to connect to the cloud database server that we have created.



```
from dotenv import load_dotenv

load_dotenv()
conn = pyodbc.connect(
        f"DRIVER={{ODBC Driver 17 for SQL Server}};"
        f"SERVER={os.getenv('SQLSERVER_HOST')};"
        f"DATABASE={os.getenv('SQLSERVER_DB')};"
        f"UID={os.getenv('SQLSERVER_USER')};"
        f"PWD={os.getenv('SQLSERVER_PASS')}",
)
cursor = conn.cursor()
```

Figure 12: Cloud server connection

The dotenv module here hides the connection strings for connecting to the cloud database server and so makes sure that the cloud database server is secure. All the connection strings (server ip, database name, uid and password) are stored in a .env file, which only we have access to on our local computers since we didn't want to push the connection strings to GitHub.

In conclusion, **overall milestones achieved** were implementing core functionality of the user - as it is the main role here- and creating the user's timetable by user's own specifications. Also added functionality which allows users to free up a slot. **Later milestones remain as** : automatic scheduling algorithm, statistical algorithm, admin role for database & some other user algorithms dependent on the needs of frontend.

4. Tools & Infrastructure

4.1. Software Development Tools

4.1.1. Frontend Tools

The frontend of the project was developed using Vite as the build tool due to prior familiarity, its rapid hot reload performance, and efficient project setup. Tailwind CSS was incorporated to streamline the styling process. Axios was used to handle communication with the backend through a set of modular API functions. Visual Studio Code served as the primary development environment and was selected because of prior experience with the editor and its overall usability. Together, these tools provided a comfortable and productive workflow for implementing the frontend of the project.

4.1.2. Backend Tools

The backend of the project was developed using Python, more specifically Flask. Flask was used because it is well suited for building RESTful APIs. It has an efficient and simple routing logic. Through Flask, user registrations, logins, database queries and communications are done.

4.2. Programming Languages & Frameworks

4.2.1. Frontend Languages & Frameworks

JavaScript serves as the primary programming language,with the framework of React. Typescript was also an option but was decided against as not all members had prior familiarity. React was selected as the main frontend framework.

4.2.2. Backend Languages & Frameworks

Python is used as the primary programming language for the backend with the Flask micro framework since it’s very efficient, easy to use and mostly used in web developing.

4.3. Cloud Services

Amazon RDS (Relational Database Service) is used as the primary cloud database tool in this project. RDS is a relational database service provided by AWS that helps with backups, scaling and maintenance. It is one of the most important tools for this project, since it will store all the data in its cloud database that is created. It is very easy to use too, after registering and doing some automated steps, the cloud database was created. All there has to be done was configuring a network protocol (for protection) so that everyone on the team could access it.

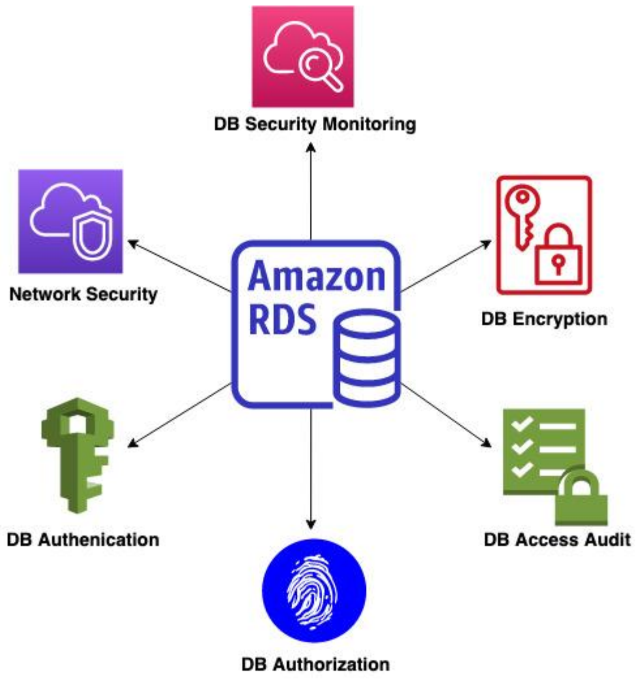
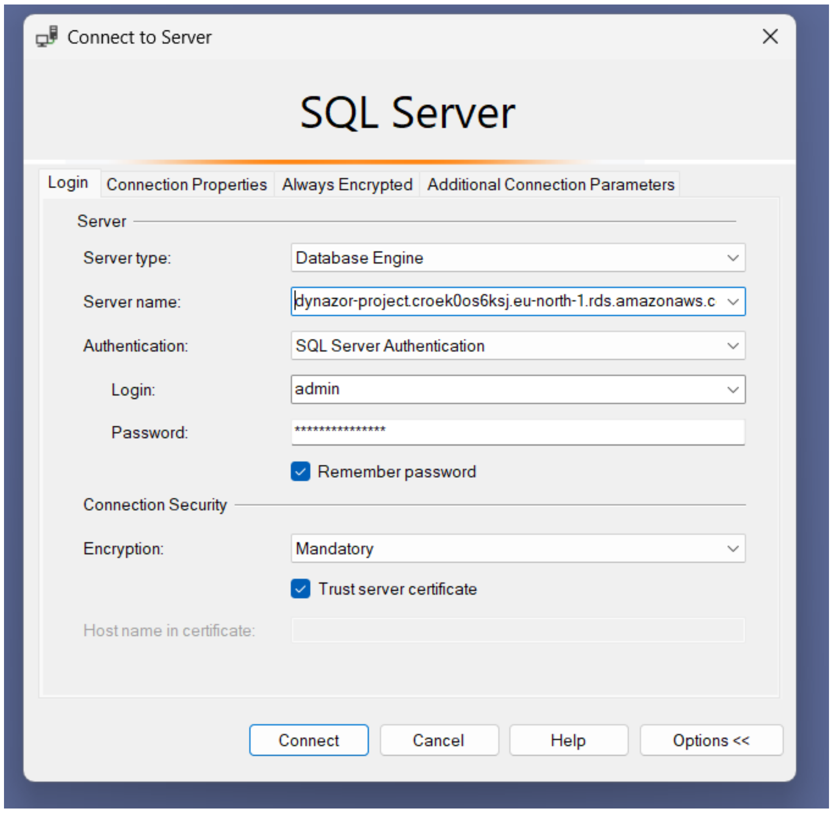


Figure 13: Amazon RDS

4.4. Database Infrastructure

Since we progressed only so far, we only have "users", "timeslots" and "userSchedule" tables in the database. Since we are using Microsoft SQL Server for the database, we use SSMS (SQL Server Management Studio), a program to manage SQL Server databases, to log in to the cloud database server and check our database.


Connect to Server

SQL Server

Server

Server type: Database Engine

Server name: dynazor-project.croek0os6ksj.eu-north-1.rds.amazonaws.c

Authentication: SQL Server Authentication

Login: admin

Password: ***************

☑ Remember password

Connection Security

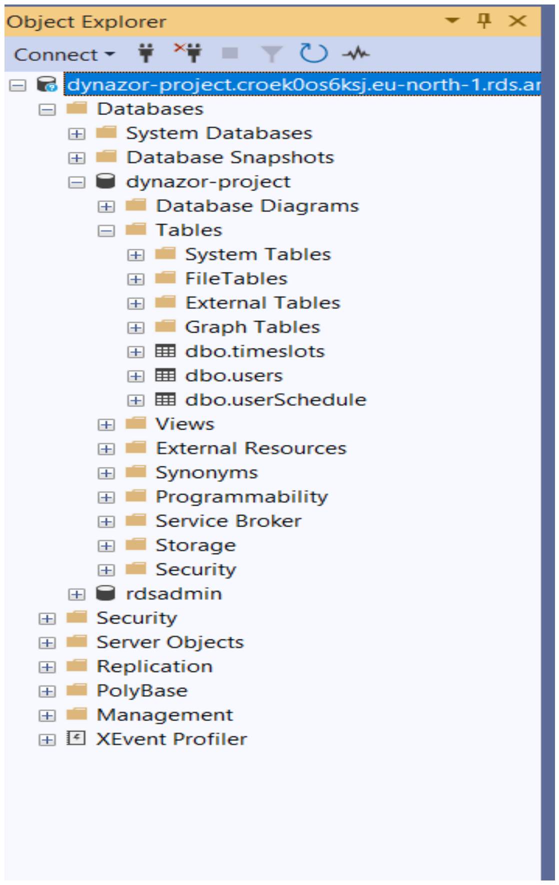
Encryption: Mandatory

☑ Trust server certificate

Host name in certificate:

Figure 14: Login page for SSMS

We can see our database “dynazor-project” in our cloud database server.



Object Explorer

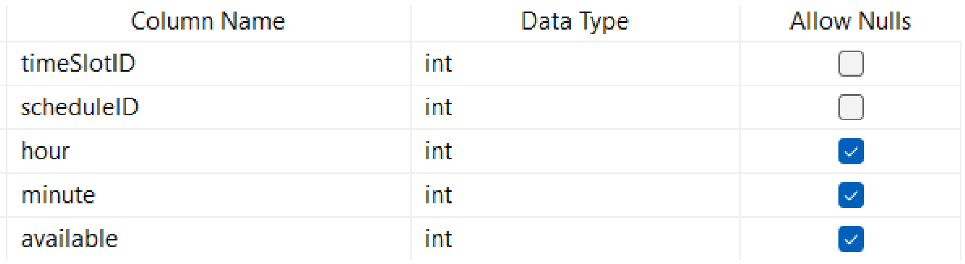
Connect

- dynazor-project.croek0os6ksj.eu-north-1.rds.ar
  - Databases
    - System Databases
    - Database Snapshots
    - dynazor-project
      - Database Diagrams
      - Tables
        - System Tables
        - FileTables
        - External Tables
        - Graph Tables
        - dbo.timeslots
        - dbo.users
        - dbo.userSchedule
      - Views
      - External Resources
      - Synonyms
      - Programmability
      - Service Broker
      - Storage
      - Security
    - rdsadmin
  - Security
  - Server Objects
  - Replication
  - PolyBase
  - Management
  - XEvent Profiler

Figure 15: Cloud database "dynazor-project" in the server

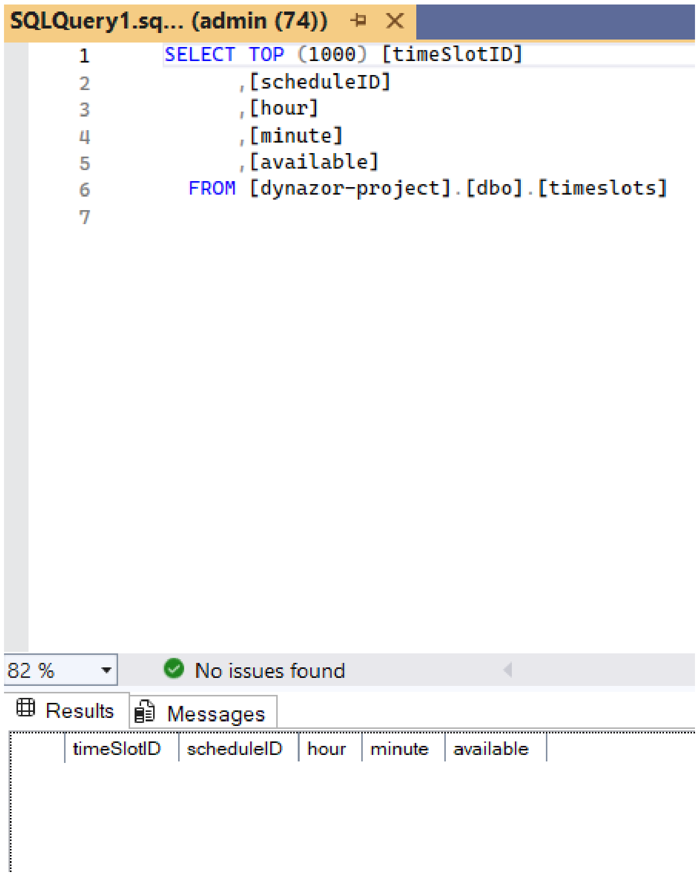
We can also see the tables that is created on the database:

"timeslots" table (empty for now):



Column Name	Data Type	Allow Nulls
timeSlotID	int	☐
scheduleID	int	☐
hour	int	✓
minute	int	✓
available	int	✓

Figure 16: "timeslots" table

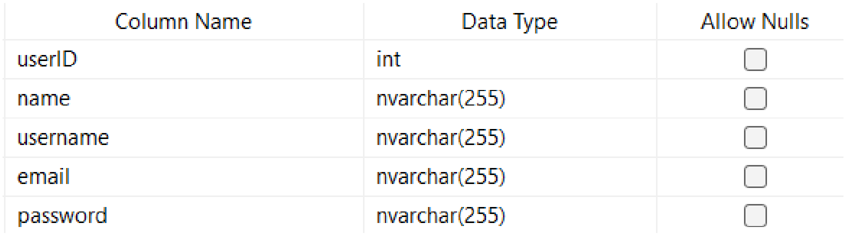


```sql
SELECT TOP (1000) [timeSlotID]
      ,[scheduleID]
      ,[hour]
      ,[minute]
      ,[available]
  FROM [dynazor-project].[dbo].[timeslots]
```

timeSlotID	scheduleID	hour	minute	available

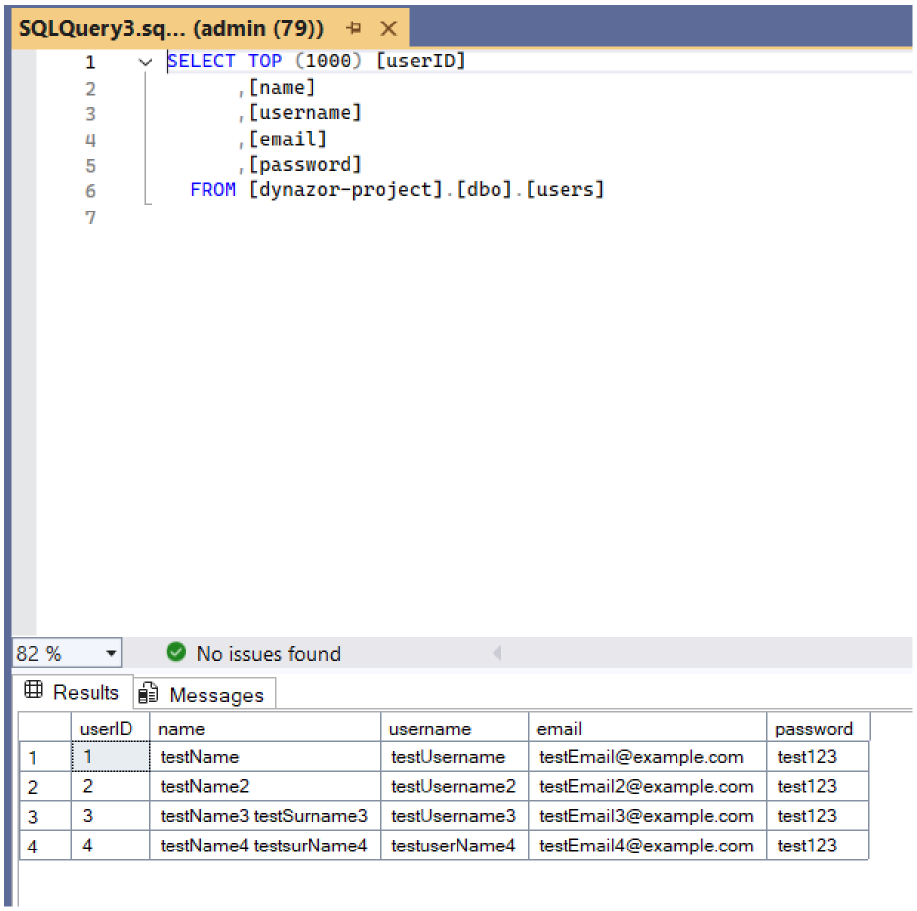
Figure 17: "timeslots" table in the database

“users” table (it has some test values for register and login in it):



Column Name	Data Type	Allow Nulls
userID	int	☐
name	nvarchar(255)	☐
username	nvarchar(255)	☐
email	nvarchar(255)	☐
password	nvarchar(255)	☐

Figure 18: "users" table


SQLQuery3.sq... (admin (79))

```
SELECT TOP (1000) [userID]
      ,[name]
      ,[username]
      ,[email]
      ,[password]
  FROM [dynazor-project].[dbo].[users]
```

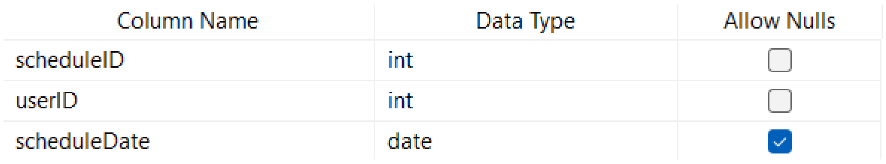
82 %   No issues found

Results   Messages

	userID	name	username	email	password
1	1	testName	testUsername	testEmail@example.com	test123
2	2	testName2	testUsername2	testEmail2@example.com	test123
3	3	testName3 testSurname3	testUsername3	testEmail3@example.com	test123
4	4	testName4 testsurName4	testuserName4	testEmail4@example.com	test123

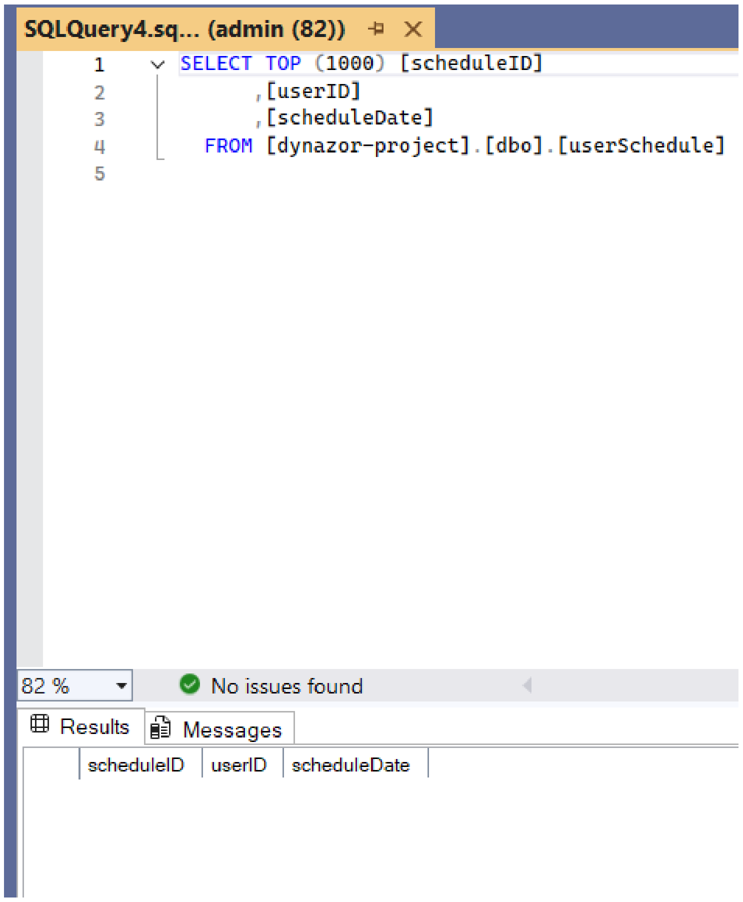
Figure 19: "users" table in the database

“userSchedule” table (empty for now):



Column Name	Data Type	Allow Nulls
scheduleID	int	☐
userID	int	☐
scheduleDate	date	✓

Figure 20: "userSchedule" table



```
SELECT TOP (1000) [scheduleID]
      ,[userID]
      ,[scheduleDate]
  FROM [dynazor-project].[dbo].[userSchedule]
```

```
scheduleID  userID  scheduleDate
```

Figure 21: "userSchedule" table in the database

4.5. Methodological tools & Version Control

4.5.1. Git and github

Version control was managed using Git, which played a central role in structuring the development process. Git branches were used to isolate new features, experiments, and fixes, preventing unfinished work from affecting the stable version of the project. The main branch was kept clean, and as the development process evolved, changes were merged. Git commits were made somewhat frequently to document progress and maintain a clear history of changes. GitHub was used to keep track of these changes. It also allowed the team to review commit history, and manage branches more effectively. This version control workflow supported collaboration, and provided an organized record of the project's development stages.

5. Next Steps & Conclusion

5.1. Milestones Remaining

Frontend:

The current priority is to refine the schedule page, both in terms of functionality and user experience. While the API connection for fetching and updating schedule and user data has already been integrated, the interactive features such as user specific schedules, saving schedule changes, and reflecting backend updates in real time still need to be developed.

Another major milestone is improving the overall styling and consistency of the interface. Tailwind CSS has been integrated, but much of the UI still relies on temporary or placeholder layouts. The plan is to refine the styling of all pages, especially the schedule page to finalize spacing and alignment..

Additionally, the authentication flow requires further work. The application will be updated to display the logged in user's name on the schedule page and maintain the user state for the duration of the session. This includes implementing temporary state storage of the username after login.

Navigation improvements are also part of the remaining tasks. Routing will be expanded so that users cannot access protected pages (like the schedule) without logging in first. Proper redirects will be added to achieve smooth transitions between pages.

Finally, general cleanup and optimization tasks remain, such as restructuring repetitive components, verifying API error handling, and polishing folder organization as new features are added. Once these milestones are completed, the frontend will be functionally complete and aligned with the goals of the project.

Backend:

As mentioned in Chapter 3.2.8, the remaining tasks for the backend are prioritizing scheduling algorithms and completing the needs of the user interactions. With this achievement being the major target, raw data that is being taken from the database will be processed in the backend before presenting the user as statistical data.

Admin role implementation is also needed in this case in order to ensure user privacy and overall application security in which the privileges will be revisited and determined.

Other user interactions such as booking/selecting free timeslots from other users table, their boundaries etc. later will be discussed and implemented after the interaction testing on the frontend.

As far as milestones are concerned, remaining achievements are expected to be completed relatively smoothly.

5.2 Conclusion

At this stage of the DynaZOR project, the core foundations of the system have been successfully established. On the frontend side, the main pages, Login, Register, and Schedule have been implemented using React with Vite and Tailwind CSS, and API connections have been set up to communicate with the backend. On the backend, the Flask based REST API, the cloud database on Amazon RDS, and the essential user operations have been implemented and tested against the current database schema. These achievements confirm that the general architecture proposed in the initial project plan is feasible and appropriate for the system’s goals.

Although the system is not yet feature complete, the current progress is a stable and extendable base for the next development steps. The remaining work mainly focuses on implementing the scheduling and statistical algorithms, refining user specific schedule handling, improving the visual design of the schedule interface, and finalizing role based features such as admin operations. With the core infrastructure, database, and communication layers already in place, the project is in a good position to move into the next phase, where more advanced functionality and usability enhancements can be developed.

6. GitHub Repository:

https://github.com/BurakTreaty/DynaZOR.git

7. References

Tailwind Labs. (n.d.). Installation: Using Vite. Tailwind CSS. https://tailwindcss.com/docs/installation/using-vite

Vite. (n.d.). Guide. Vite. https://vite.dev/guide/

Pallets Projects. (n.d.). Flask. Pallets Projects. https://flask.palletsprojects.com/en/stable/

Flask-RESTful. (n.d.). Flask-RESTful documentation. https://flask-restful.readthedocs.io/en/latest/