MOS EISLEY CANTINA SYSTEM

Table of Contents

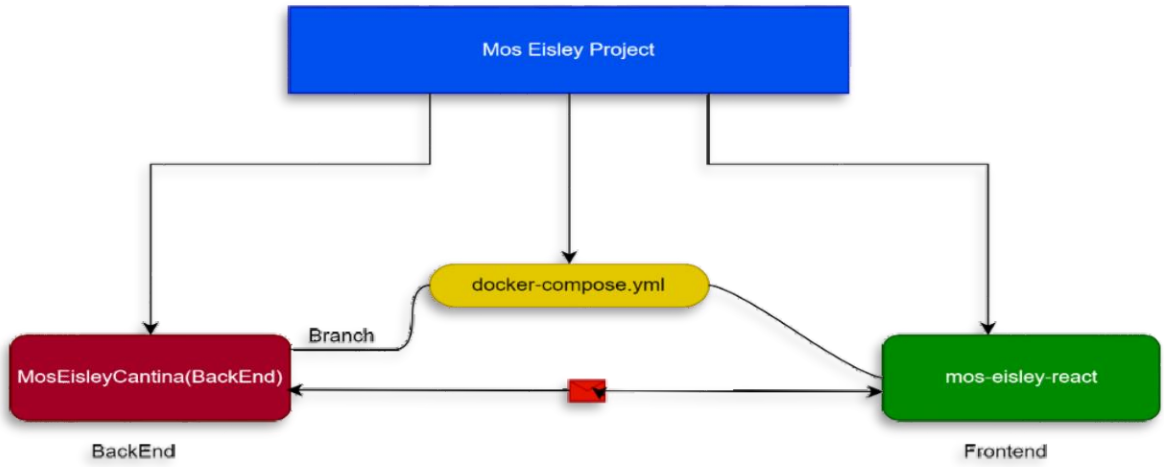
User Guide for Mos Eisley Cantina system

Document Revisions

Date	Version Number		Document Changes	
17/12/2024	1		Initial Draft	
18/12/2024	2		Database , Activity and ERD diagrams	
19/12/2024	3		Development	
20/12/2024	4		API Documentation	

1 *Introduction*

System Architecture Overview
Folder structures



1.1 Scope and Purpose

Objective

The **Mos Eisley Cantina API** is designed to deliver a scalable, maintainable, and secure REST API for Chalmun's cantina on Tatooine. This API enables seamless management of dishes, drinks, user authentication, and reviews. The project aims to enhance customer experience, expand service offerings, and boost sales through a robust technological solution.

- **Introduce the practical system and its purpose.**
  - **Project Background**

    Chalmun, the Wookiee proprietor of the Mos Eisley Cantina, wishes to modernize the cantina's operations. Patrons currently must dine onsite, limiting reach and scalability. By contracting a modern API-driven solution, Chalmun can offer a digital interface for customers to browse, order, and engage with the cantina's services remotely. This documentation outlines the technical implementation of the REST API and supporting systems.

- **Key features and benefits.**
  - **Trials (Specifications)**

Mos Eisley Cantina

The project is structured into four task levels to accommodate varying expertise:

Task 1 Basic CRUD Functionality (All Candidates)

- **Requirements**:
    - Implement CRUD operations for dishes and drinks.
    - Entities must include name, description, price, and image fields.
    - Enable customers to search, view, and rate dishes and drinks.

Task 2: Authentication and Permissions (Intermediate & Senior)

- **Requirements**:
    - User registration and login functionality.
    - Role-based access control (Admin, User).
    - Validate data entities and protect against brute force attacks.
- **Features**:
    - JWT-based authentication.
    - IP/device blocking for suspicious login attempts.

Task 3: Performance Optimization (Senior)

- **Requirements**:
    - Enhance API performance on legacy hardware.
    - Implement solutions like caching, pagination, and HTTP compression.
- **Advanced Options**:
    - Suggest additional performance strategies, such as load balancing and content delivery networks.

Task 4: Advanced Features (Above and Beyond)

- **Requirements**:
    - Dashboard for visualizing reviews.
    - Rate-limiting to prevent abuse.
    - OAuth2 SSO integration (e.g., Google).

❖ **Purpose of the user guide**.

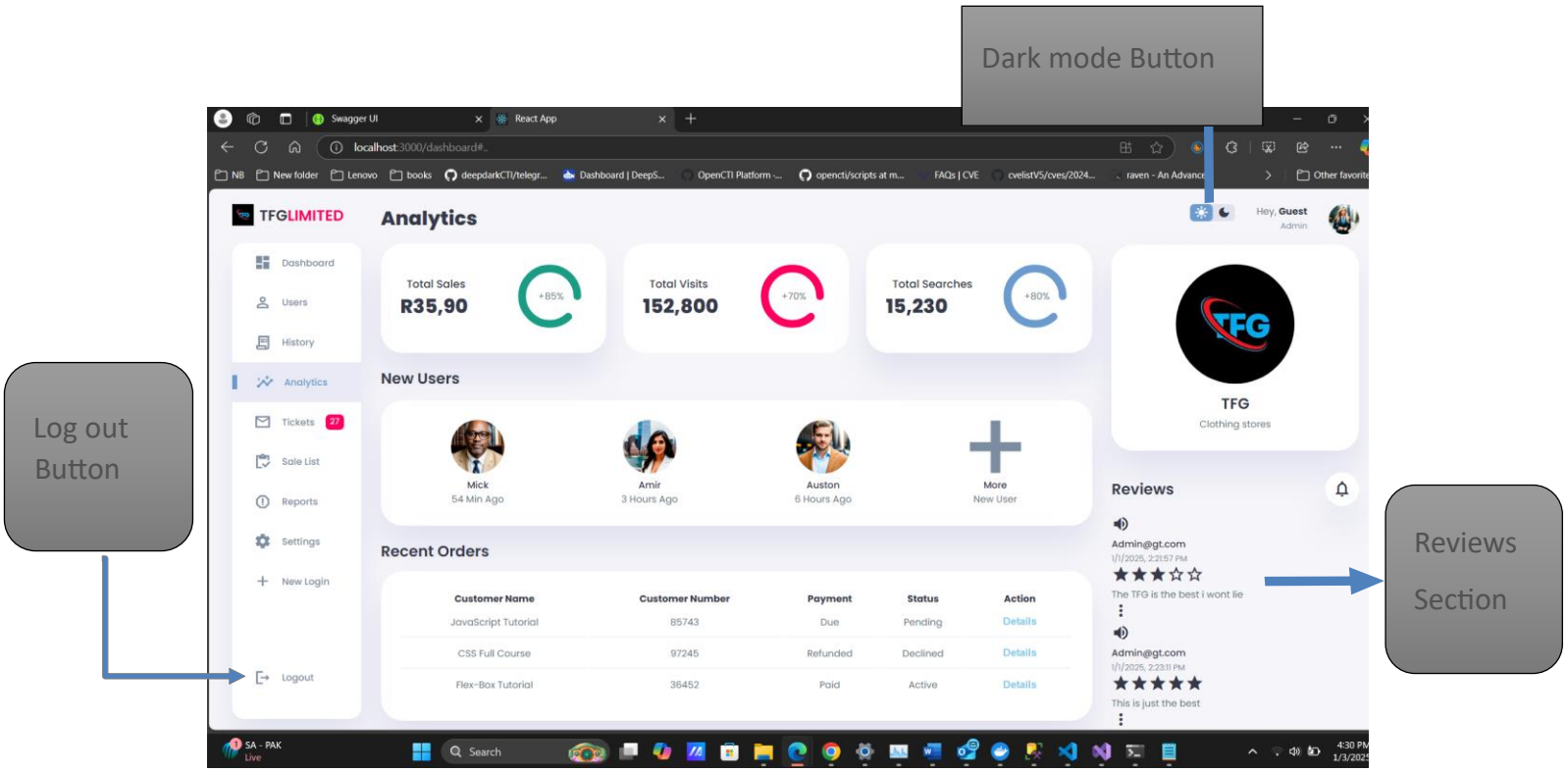
This user guide offers a detailed overview of the System, providing instructions for parents, guardians, and administrators to navigate the

platform effectively and make the most of its features. It aims to ensure that all users can access the system easily, resulting in a smooth registration experience.

1.2 Quick Start Guide

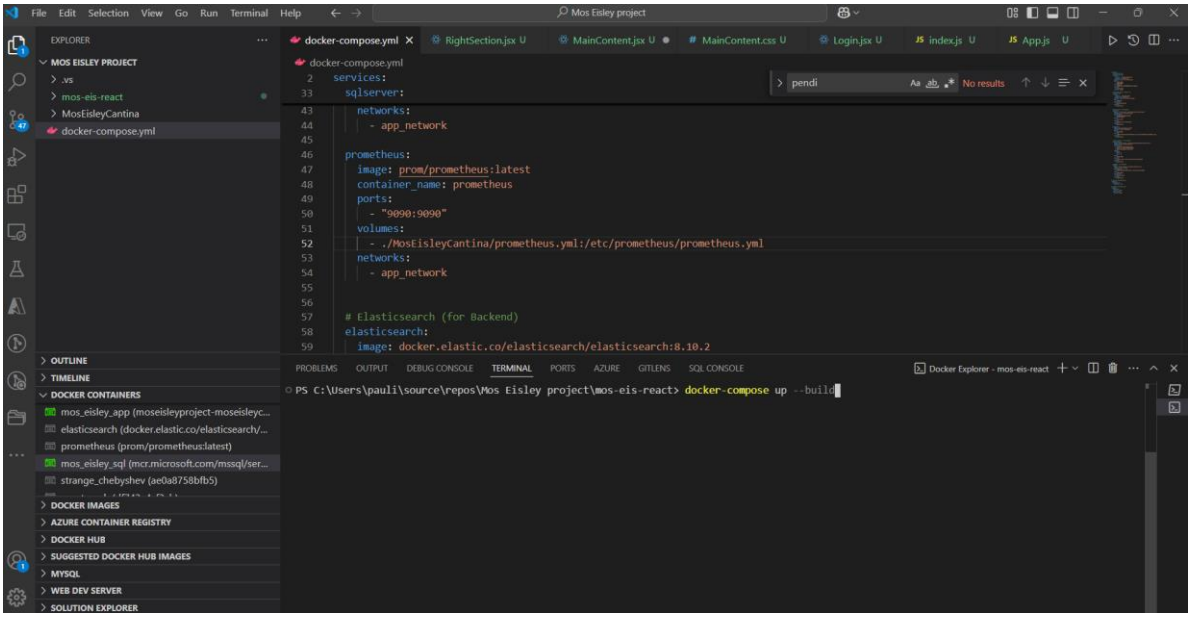
Overview of the functionality of the system and how the user will interact with the system.

- Clone the repository from GitHub.
- Install required dependencies using `.NET CLI` and `npm` for backend and frontend, respectively on windows or run the docker compose file using `startdocker-compose up --build`.
- Configure the appsettings.json file for database connection.
- Launch the backend API using `dotnet run` and the frontend using `npm start`.
- Access the Front end on http://localhost:3000/ and backend swagger at

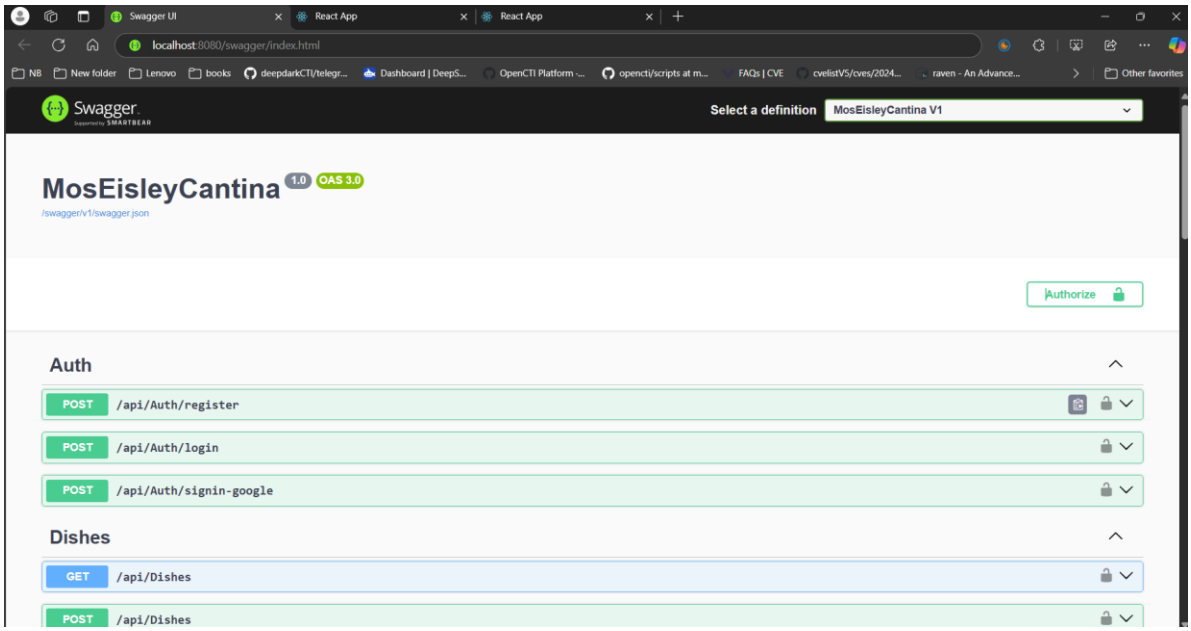


2.1 Steps for Processing the system(Sign in to the system)

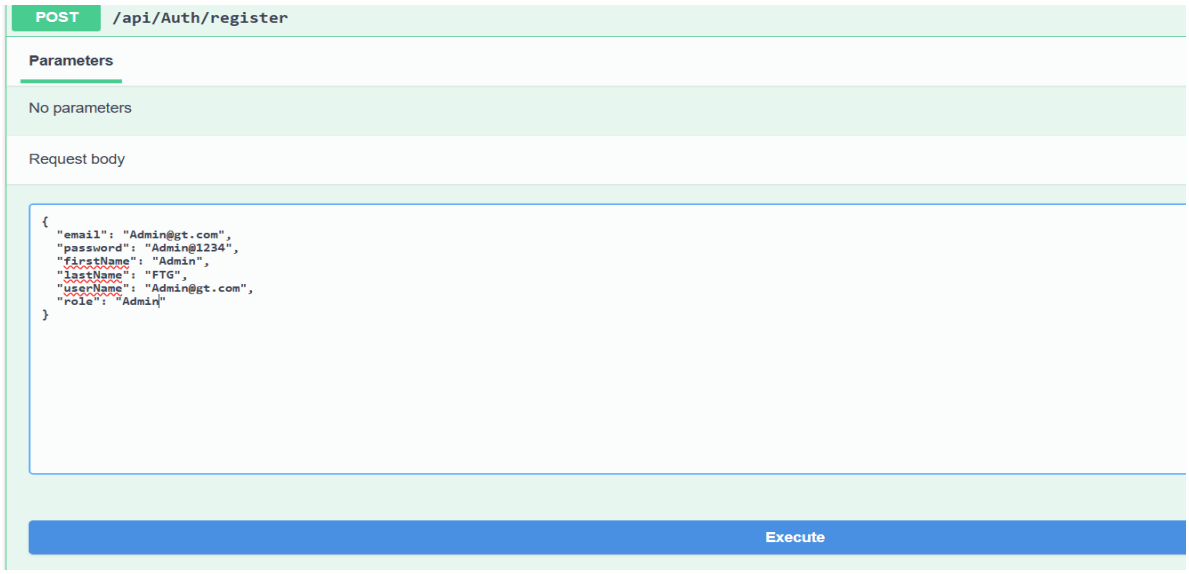
1. Launch the application on vscode and run your docker compose file



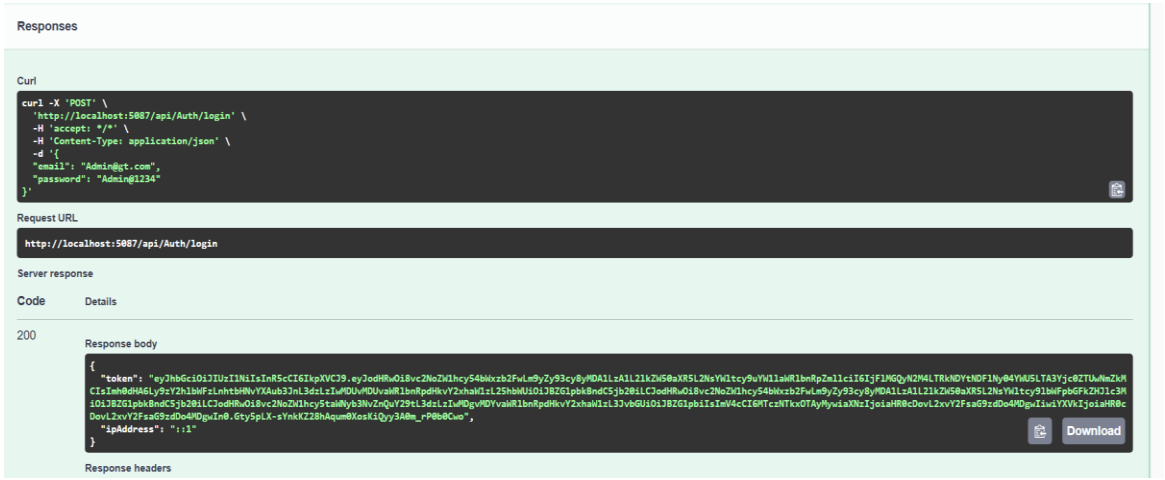
1. Access the Swagger interface at http://localhost:8080/swagger/index.html.



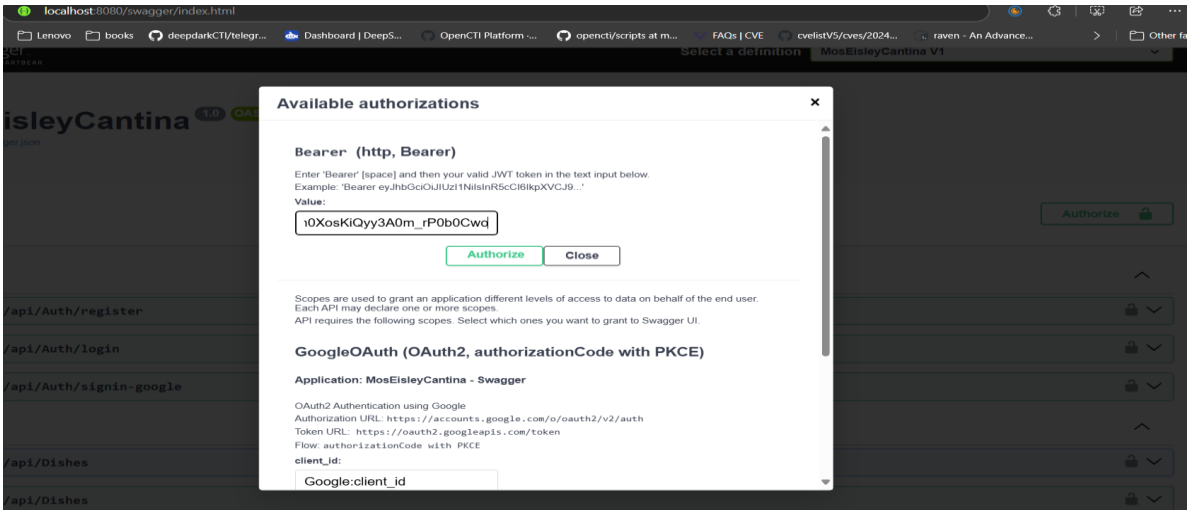
2. Register an admin and a user account using the `/Auth/register` endpoint.



3. Authenticate and obtain a JWT token using `/Auth/login`.

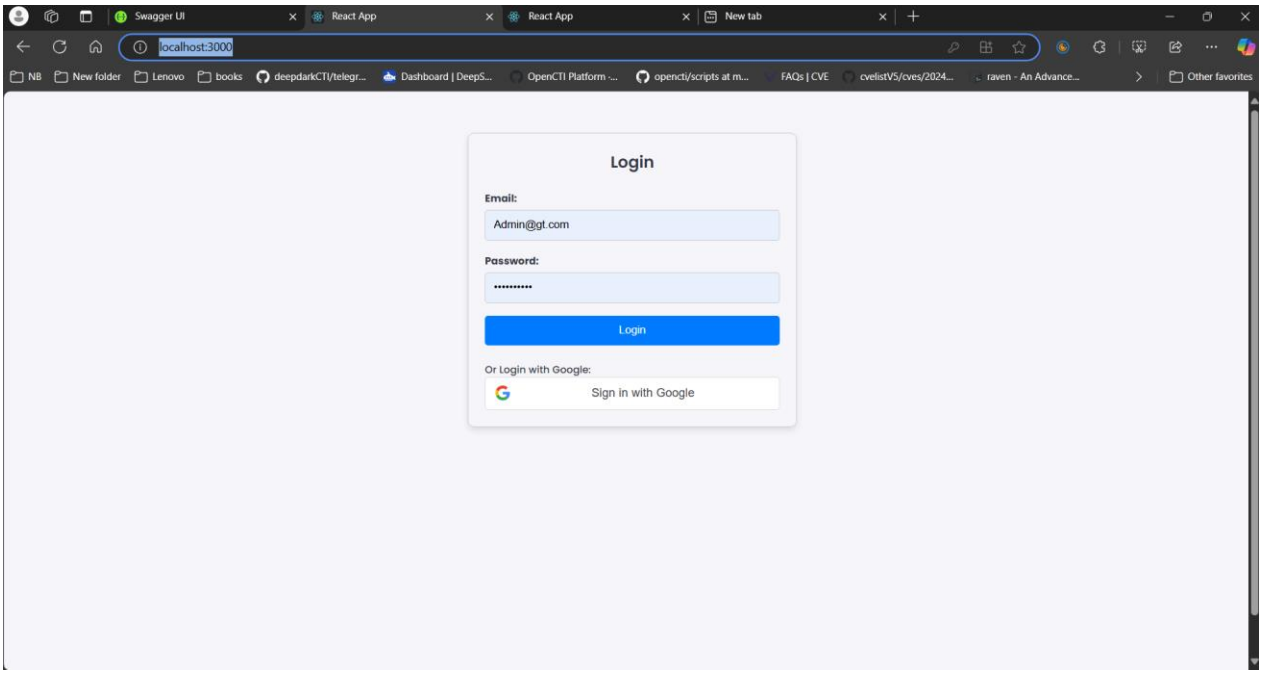


4. Use the token to authorize API requests.



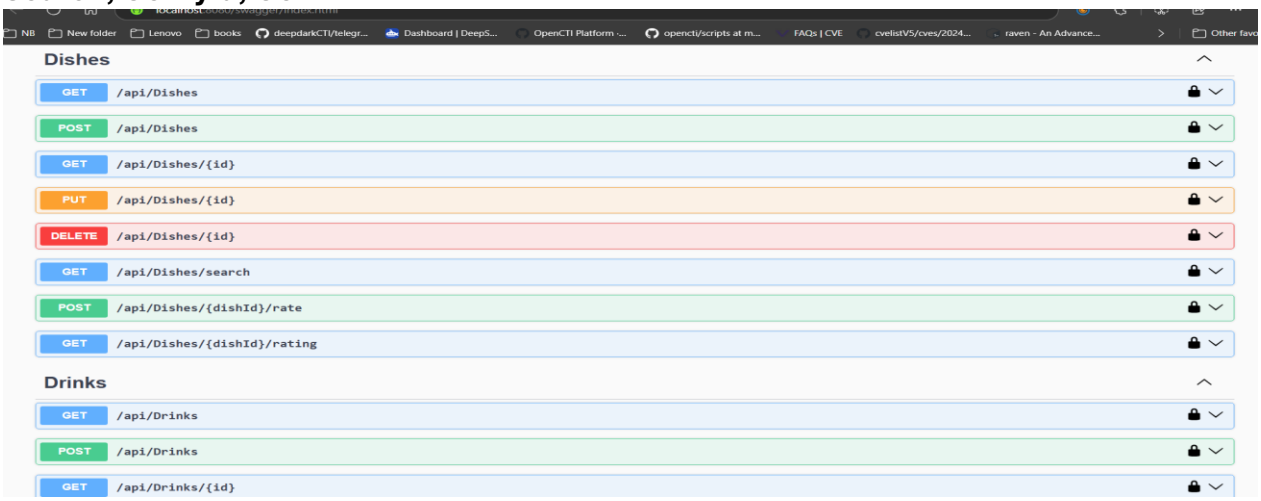
5. Perform CRUD operations on dishes and drinks via their respective endpoints.

6. Go to your browser and enter the swagger url http://localhost:8080/swagger/index.html for the back-end and React App (http://localhost:3000/) for the front-end.

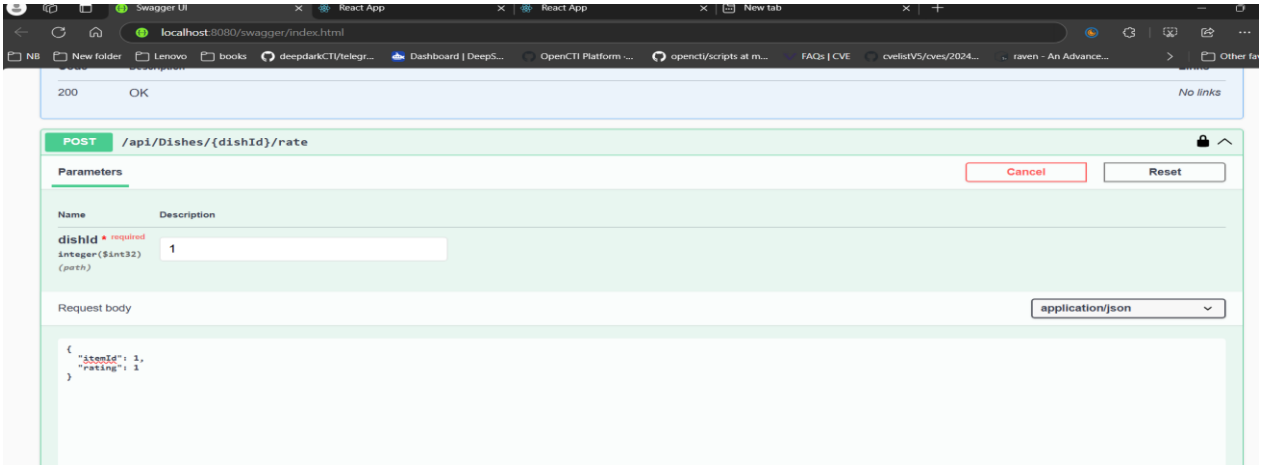


7. Test all your API's, Note the System will seed dishes and drinks data the other data will be manually entered for testing depending on the Roles.

Task 1 (All Candidates) Admin can test this API's : GetAll, GetById, Create, Update, Delete, RateDish, GetDishRating. And the User can test: RateDish, Search, GetById, GetAll.

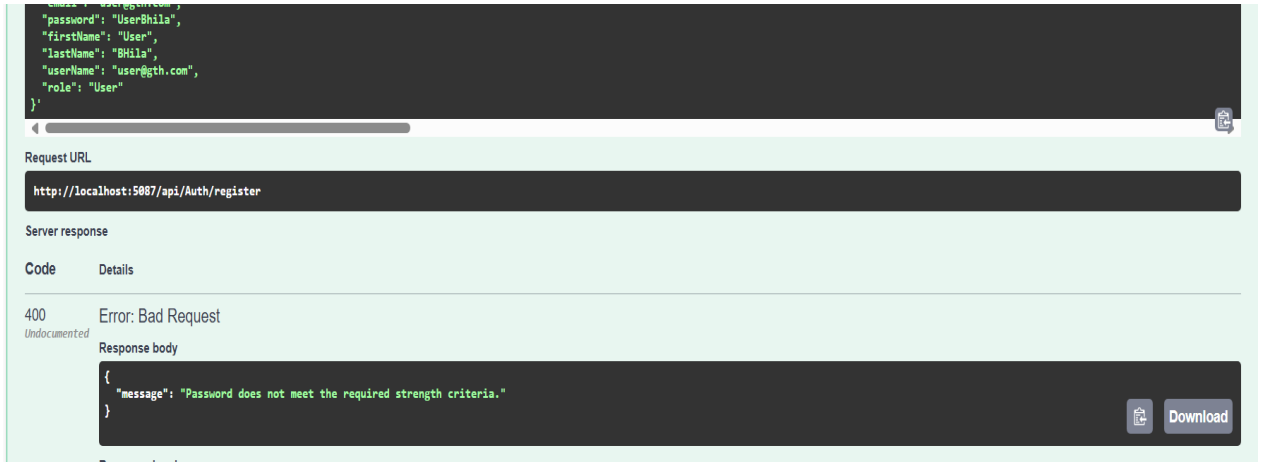


On the rating API the ItemId must be the same as the Dish or DrinkID



Task 2 (Intermediate & Senior)

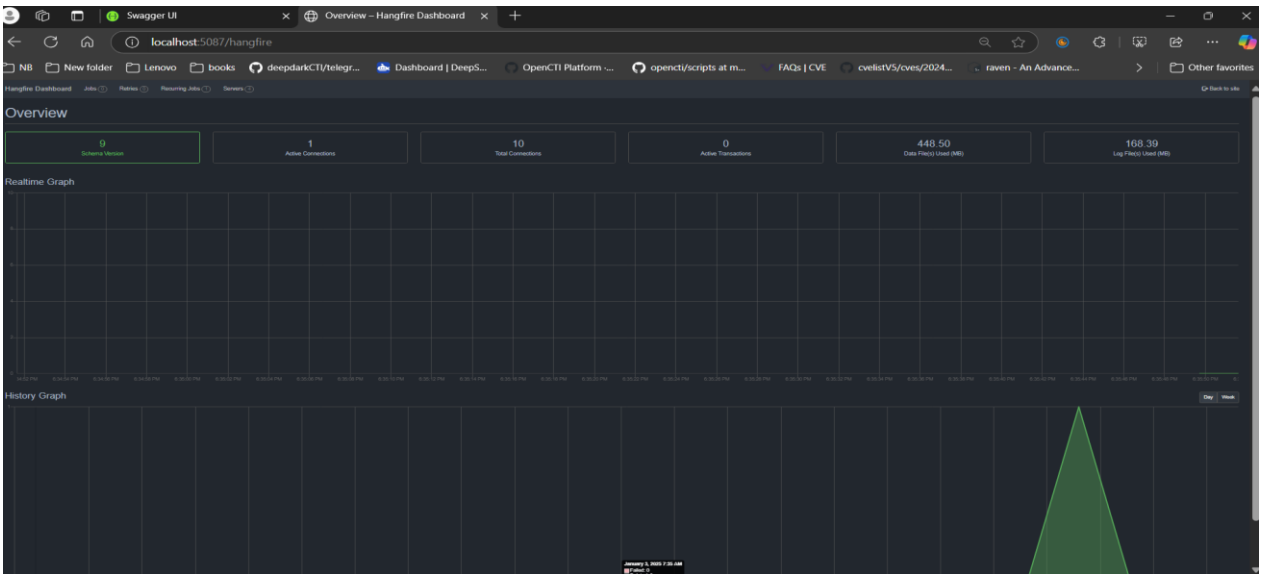
An Evil Sith Lord is using the dark side of the force to brute force passwords for known email addresses. Add defences to prevent a data breach. ***Note on registering the system will capture the users IP Address and or Device Id , if the user tries to login with a different device not registered in the system the system will block the IPAddress***



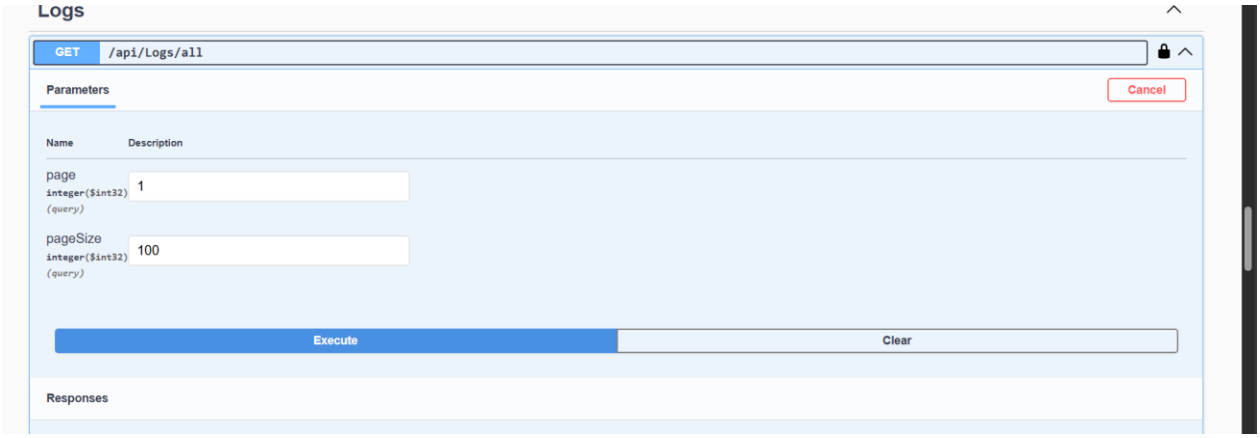
Task 3 (Senior)

The API is running on an old remote Server in the outer rim of planets that is starting to struggle with the load of traffic hitting it. Implement a solution to improve the performance of the API on the same hardware.

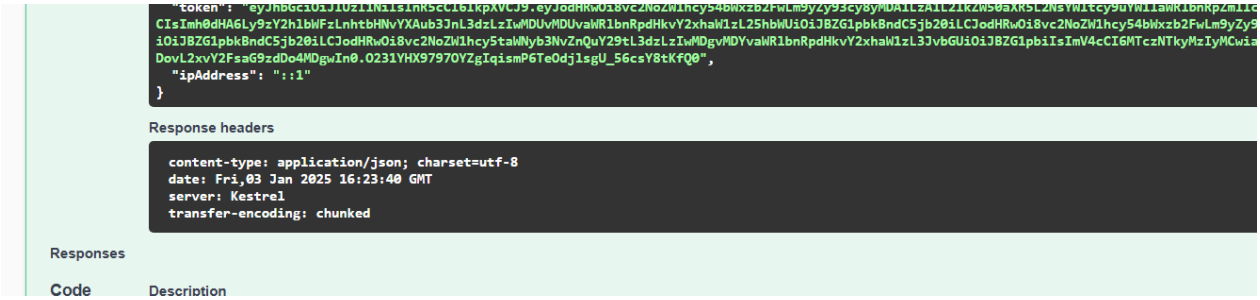
Hangfire: is an open source framework for background job processing in .NET applications. It allows you to easily schedule and run background tasks such as long-running processes, background jobs, and delayed tasks without blocking the main application thread. It integrates with ASP.NET Core and provides a simple API for scheduling and managing jobs.



Reduce Data Load with Pagination or Filtering and Memory Cache (In-Memory Caching): pagination or filtering is used to limit the amount of data being retrieved and returned.



HTTP compression for your API responses: The Transfer-Encoding: chunked header in HTTP responses allows a server to send data in chunks, rather than all at once. This is particularly useful when dealing with large amounts of data or streaming data, as it allows the server to start sending parts of the response before the entire response is ready.



Afterwards, also consider suggesting other options to increase performance beyond using the existing hardware.

Load Balancing:

Content Delivery Networks (CDNs):

Lazy Loading:

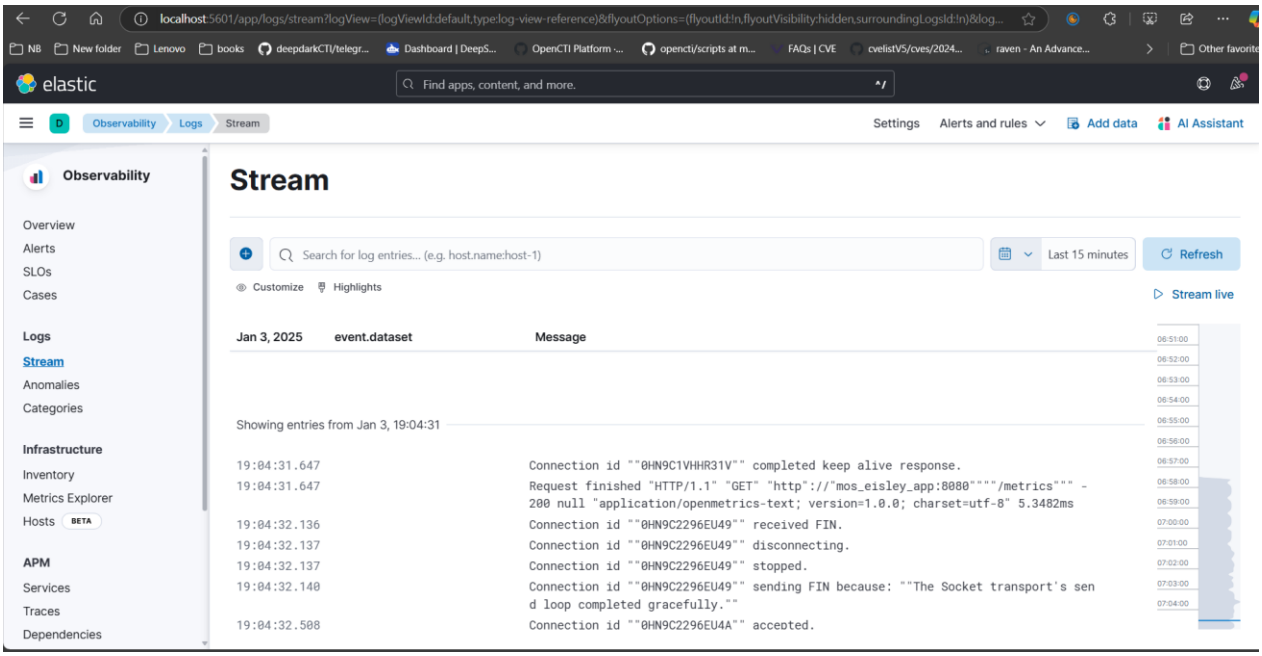
Vertical Scaling (Scale Up):

Application Insights (Azure):

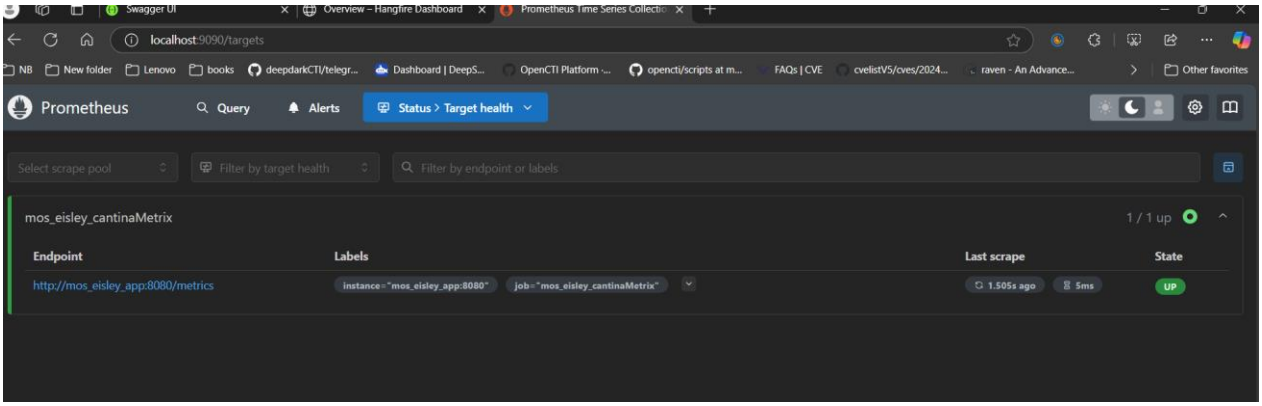
Logging and Profiling:

Monitoring and **rate limiting**.

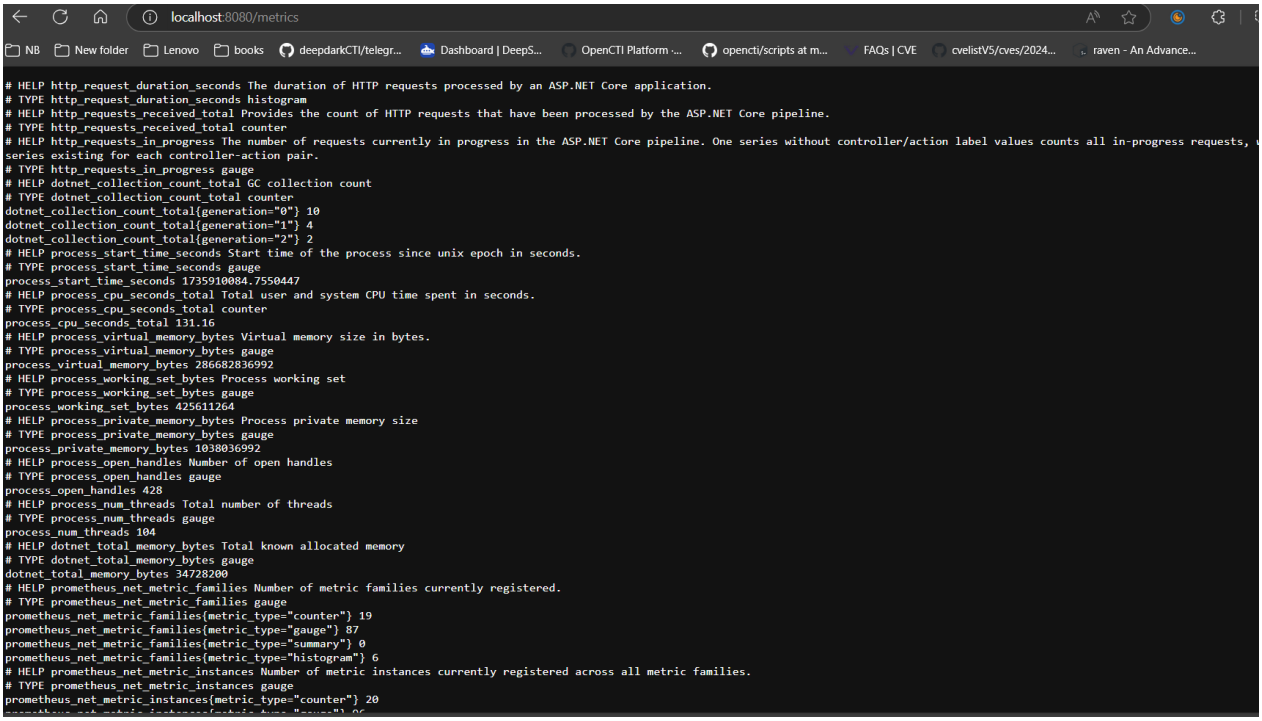
Integrated Serilog with Elasticsearch and kibana: Serilog sends logs to Elasticsearch, and Kibana provides a rich web-based dashboard to visualize and analyze the log data and for maximum security against threaths in the system



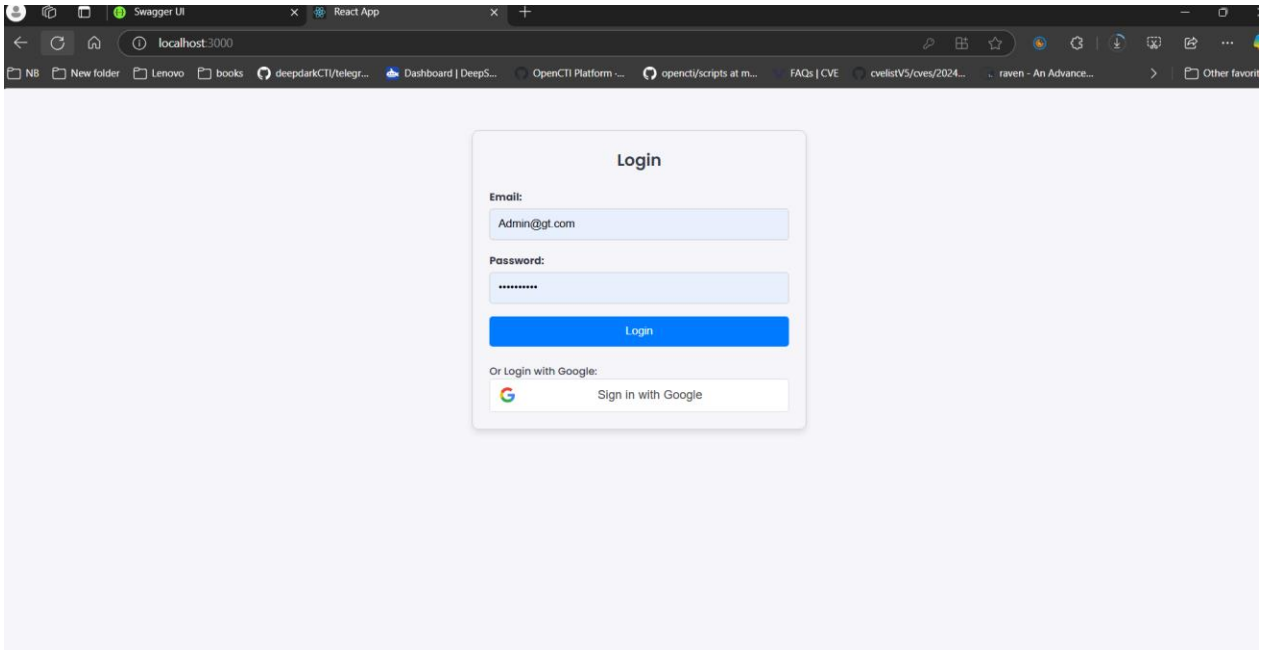
Prometheus: is great for monitoring application performance metrics, system health, and infrastructure metrics



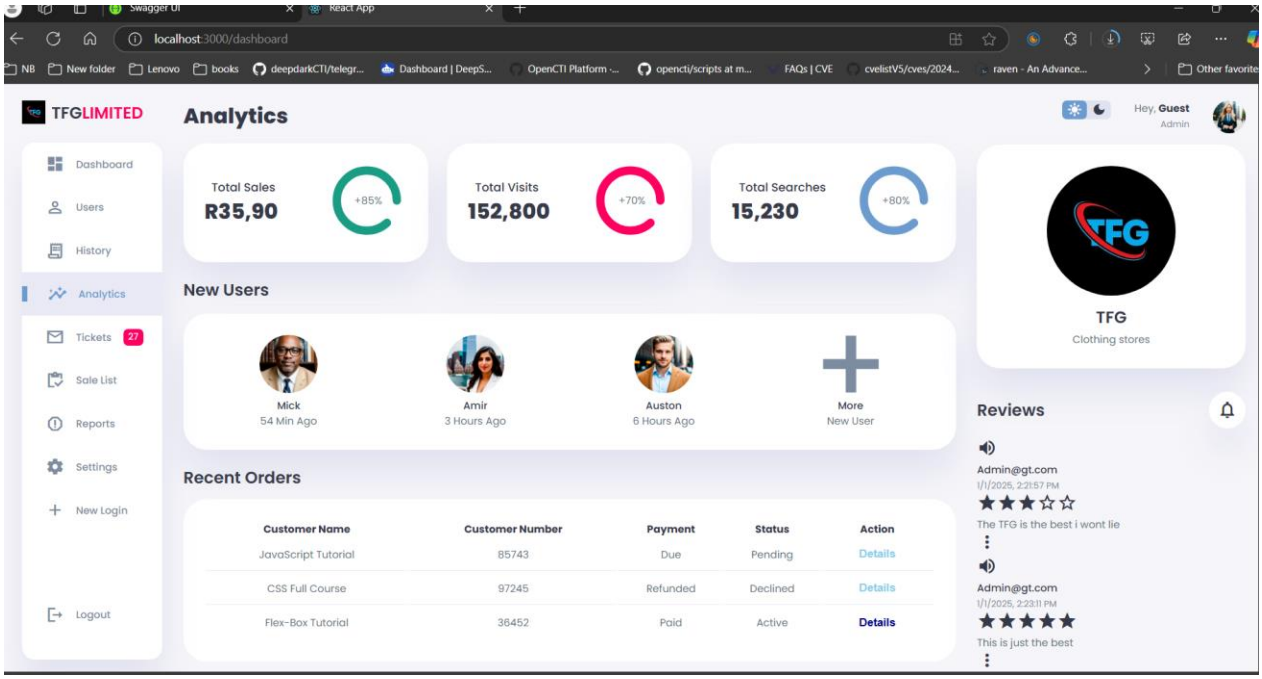
Mos Eisley Cantina



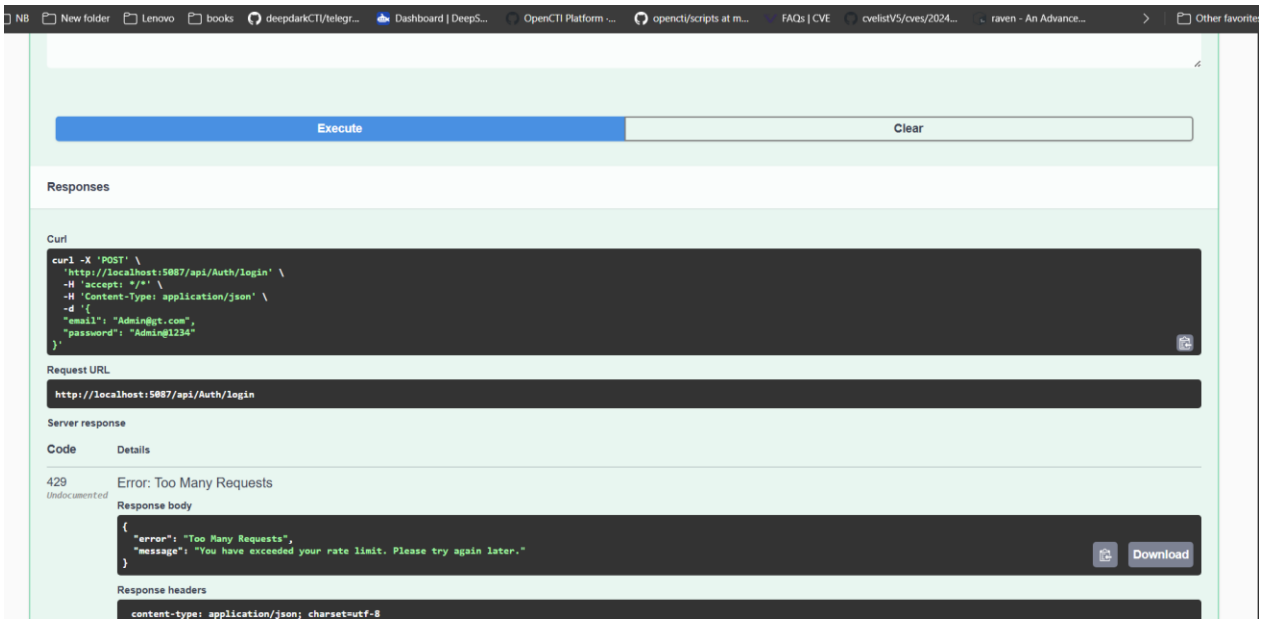
The Sith Lord has created many apprentices accounts and has left many bad reviews. Create a dashboard to view the reviews.



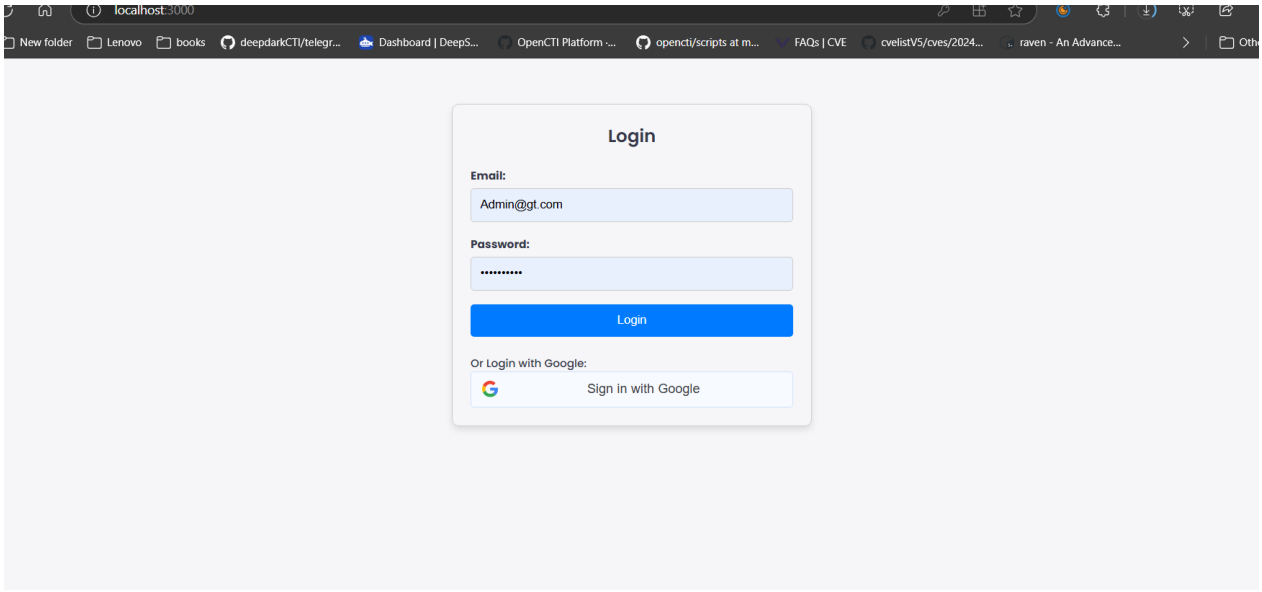
At the far end right corner every review will be seen here. Please Note that only the login , logout darkmode and Reviews is working and integrated with backend.

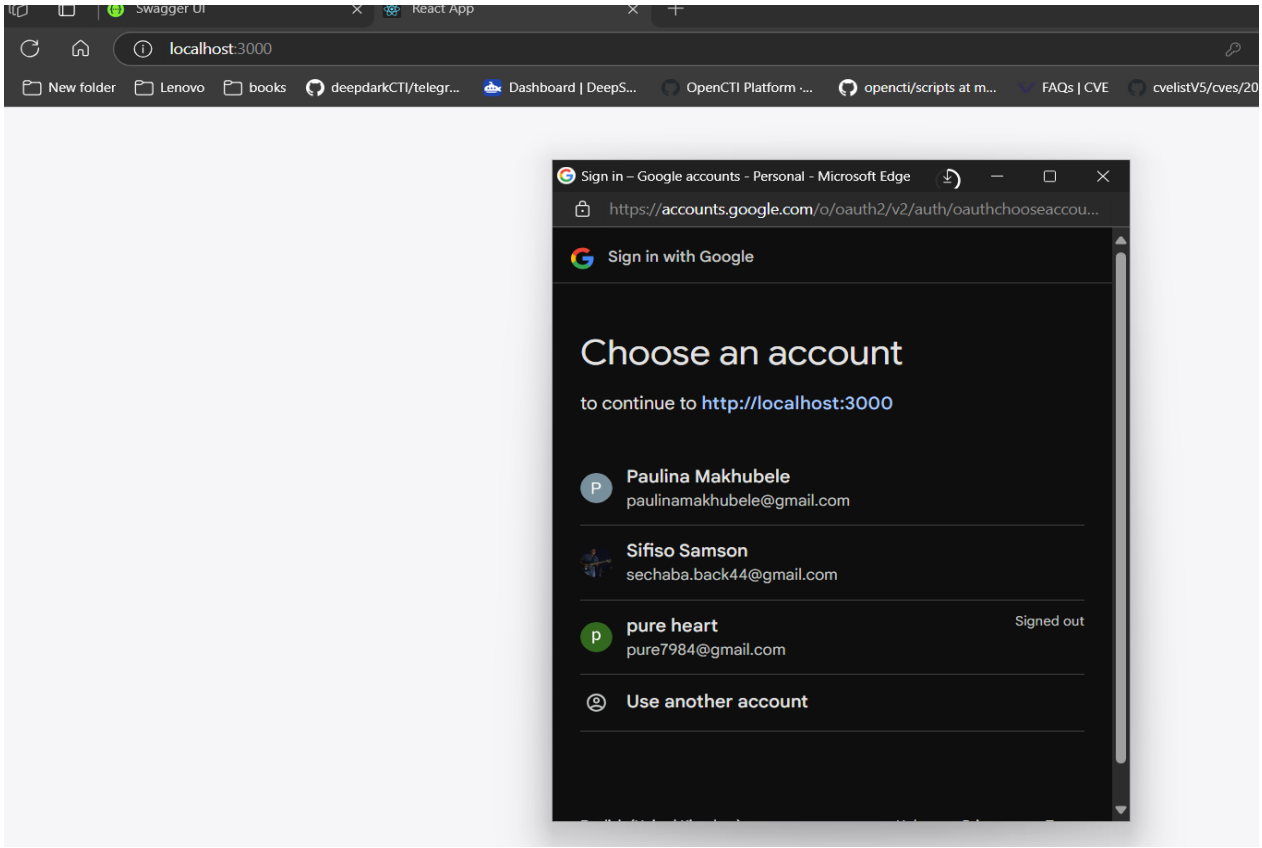


To prevent abuse, add **rate-limiting** per logged in customer.



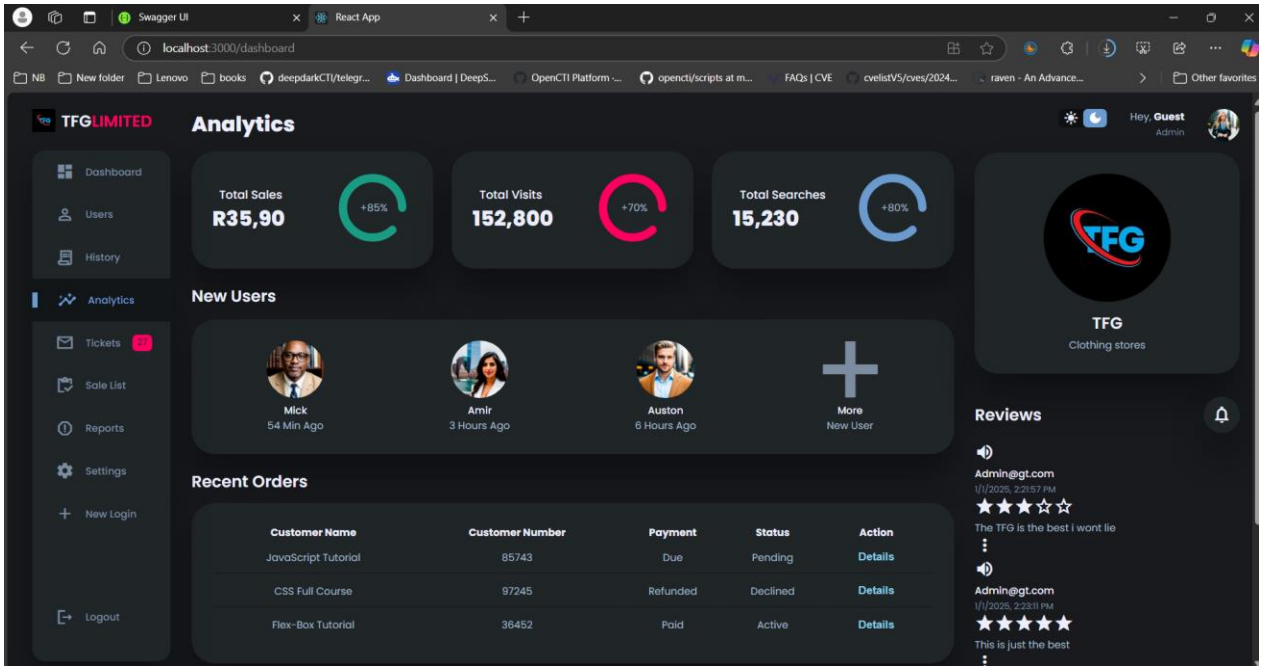
Allow users to login using **OAuth2 based SSO (Google, etc):** click on the signin with google button and follow the process.





In testing the reviews dashboard , login as an Admin or use your google email that you registered with.

8. Then you have a logout button for exiting the system **an overview of the Steps of Process**



Installation Guide

Installation | Deployment Guide for Mos Eisley Cantina app

1 Introduction

1.1 Purpose

The Installation document is to assist a software designer / developer when they create installation instructions for a practical system that is to be installed. And to provide guidelines on the general rules governing the contents.

2 Installation Manual

❖ Pre-requisites ❖ Install procedure ❖ Backup and Recovery of the Practical System and Database

2.1 Pre-requisites

In this section a **list of install pre-requisites are mentioned** that must be fulfilled before the install can begin.

Pre-requisites are of the order of:

- .NET Core SDK 8.0
- Node.js (v16 or higher)
- Docker
- Microsoft SQL Server

- **Operating System**: Linux Server 64-bit x64 based processor or above.
  ➢ **Software**: Web server, database server and React.js.
- **Hardware**: Adequate RAM 16GB or above and storage 100GB or above CPU 2 cores @ 1.90GHz, based on the expected user load, along with a reliable internet connection for remote access.
- **Access and Permissions**: Administrator rights for server access, plus necessary firewall and network configurations to allow external access to the server.

2.2 Pre-installation Tasks

- Install required software and tools.
- Clone the repository.
- Configure database connection strings.

2.3 Installation Procedure [or the Deployment]

2.3.1

Once all pre-requisite checks and controls are positively finished, the **running process** can start.

Step 1:

Database

- Run the docker-compose.yml to set up SQL Server.
- Apply migrations using `dotnet ef database update`.
- Verify database tables and relationships.

Step 2: Opening the app in the browser

- Access the Swagger interface at http://localhost:8080/swagger/index.html
- Access the UI interface (frontend) at http://localhost:3000/
- Access the UI interface (frontend) at http://localhost:9200/
- Access the UI interface (frontend) at

To set up the database for the System locally, follow these steps:

Install Database Server Software

Download and install the database management system (DBMS) specified for the project, such as SQL Server(SSMS)
Follow the on-screen instructions to complete the installation, including setting up the root user and password.
Configure Database Server

After installation, run your docker compose to access the database locally or running migrations
Set essential parameters such as max_connections, port, and data directory.
Restart the database server to apply the changes.

Create the Database

Run a few test queries to ensure the tables are created and accessible by the application user. Confirm that there are no errors during import and that all tables, indexes, and relationships are correctly set up.

Set Up Database Backups

Schedule regular automatic backups of the database. Use a cron job or a task scheduler to run backups at regular intervals.

Store backups in a secure, accessible location to ensure data recovery in case of any failure.

By following these steps, the database for the System will be correctly installed, configured, and ready for use by the client.

2.3.2 Mos Eisley Cantina System Backup and Recovery

2.4 Backup and Recovery of the System and Database

list of the steps that needs to be taken for the client – how the backup and recovery of the practical system and database will work.

Backup: Schedule regular backups of both the application files and the database, storing backups securely in an off-site location.

Recovery: In case of system failure, use the latest backup files to restore the system. Begin with the database restoration, followed by the application files. Confirm successful restoration by conducting a functionality test.

2.5 Contact Information

Senior Software/Systems developer	
Name	Sifiso
Surname	Shishaba
E-mail address	samsonshishaba@outlook.com

Business Plan

SOFTWARE DEVELOPMENT PROPOSAL | Business Plan

PREPARED FOR: TFG Limitted

PREPARED BY: Sifiso Shishaba

DATE: 18 December 2024

Sifiso Shishaba,

1 System Overview

Aims

To implement a REST API for a cantina.

Scope

Chalmun, a Wooki who owns the **Mos Eisley Cantina on Tatooine**, is wanting to boost sales. The Cantina is very popular but patrons can only order and consume their delicious offerings on site. Chalmun has contracted you from the guild of technologists to build the website for his restaurant. Your remuneration will be in republic credits. You record will be screened via republic channels, be sure that everything is in order.

Trials (Specifications)

There are 4 trials based on your level of experience. We expect candidates applying for a Junior engineer positions to complete at least the first trial, Intermediate engineers must also complete the second trail, and finally, seniors must also complete the 3rd trial. Lastly there are some ideas in the 4th trial for engineers who want to go above and beyond, likely to be one with the force.

Task 1 (All Candidates)

Deliver a REST API that meets the following requirements:

- An API consumer must be able to:
    - Create, View, List, Update, and Delete dishes and drinks.
    - Dishes & drinks must have a name, description, price, and image.
- Customers must be able to take the following actions:
    - Search, View, and Rate dishes & drinks

Junior engineers do not need to worry about users or authentication.

Task 2 (Intermediate & Senior)

- Add user, permission, and authentication support.
- Users must be able to register and login.
- All functionality of the API must require a logged in user (except Registration)
- At a minimum, the system should support password based authentication.
- Users must have a name and email address and password.
- Add validation to the data entities in the API.

- An Evil Sith Lord is using the dark side of the force to brute force passwords for known email addresses. Add defences to prevent a data breach.

Task 3 (Senior)

- The API is running on an old remote Server in the outer rim of planets that is starting to struggle with the load of traffic hitting it. Implement a solution to improve the performance of the API on the same hardware.
- Afterwards, also consider suggesting other options to increase performance beyond using the existing hardware.

Task 4 (Above and Beyond)

- The Sith Lord has created many apprentices accounts and has left many bad reviews. Create a dashboard to view the reviews.
- To prevent abuse, add rate-limiting per logged in customer.
- Allow users to login using OAuth2 based SSO (Google, etc)

Constraints

- Additional points go to those who use Golang.
- C#, Python (preferably fastapi), or JS/Typescript other languages, accept we will.
- Implement a REST API utilizing JSON for request and response bodies where applicable.

Intended Operation

The system will operate as a web-based application that can be accessed from any device with internet connectivity. Users will be able to log in to swagger. Administrators will have access and test swagger API's and access to the dashboard and be able to view the reviews.

Design Process

The design process followed an iterative approach, involving:

1. Requirement Analysis: Gathering detailed requirements from stakeholders.
2. System Design: Creating design documents and prototypes.
3. Development: Coding and integrating system components.
4. Testing: Conducting thorough testing to ensure functionality and performance.
5. Deployment: Installing the system on the server and making it accessible to users.
6. Maintenance: Providing ongoing support and updates.

Functionality

Key functionalities of the system include:

- **User Registration and Authentication:** Secure login for Admin and Users.
- **Dish Management:** Create, View, List, Update, and Delete dishes and dishes.
- **Drinks Management:** Create, View, List, Update, and Delete dishes and drinks.
- **Logs:** Generation of various logs for administrative purposes.
- **Reviews:** Reviews are displayed on Dashboard under Analysis.

Technology Choices

- **Hardware:** The system will be hosted on a reliable server with sufficient processing power and storage capacity.
- **Software:** The system will be developed using modern web technologies such as front-end Reactjs, HTML, CSS, and JavaScript, with a backend framework .Net core Web API. The database will be managed using SSMS(SQLServer).

Software Distribution

The software will be distributed as a web application accessible via a web browser, requiring no additional installation on user devices.

User Capacity

The system is designed to handle multiple concurrent users, with scalability options to accommodate future growth in user numbers.

Intellectual Property Ownership

The intellectual property of the software, including the source code and design documents, will be owned by TFG. TFG will have full rights to modify, distribute, and use the software as needed.

Testing

Comprehensive testing will be conducted, including:

- **Unit Testing:** Testing individual components for functionality.
- **Integration Testing:** Ensuring that all components work together seamlessly.
- **User Acceptance Testing (UAT):** Involving end-users to validate that the system meets their needs.

Support and Warranties

- **Support:** Ongoing technical support will be provided to address any issues and ensure smooth operation.
- **Warranties:** A warranty period will be offered during which any defects or issues will be resolved at no additional cost.

This overview provides a clear description of the Mos Eisley Cantina, outlining its aims, scope, intended operation, design process, functionality, and support

2 Technology requirements

2.1 Hardware

Desktop and Laptop Compatibility:

The system will run on 32-bit and 64-bit Windows desktop systems (Windows 8 and later), Mac OS (macOS 10.10 Yosemite and later), and Linux distributions.

Recommended hardware: Intel i3 processor (or equivalent) with 8GB RAM and 100MB of free disk space.

Mobile Compatibility:

The system is optimized for mobile browsers and is compatible with iOS and Android smartphones. iPhones (iOS 11 and later) and Android smartphones (Android 8.0 Oreo and later) are supported.

Browser Requirements:

The system is compatible with the latest versions of major web browsers, including Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge.

JavaScript and cookies must be enabled in the browser for optimal functionality.

Server Requirements:

The system will be hosted on a cloud server (such as AWS, Google Cloud, or Microsoft Azure) to allow for secure data handling, remote access, and scalability. Minimum recommended server specifications:

2 CPU cores, 6GB RAM and 100GB storage space

Secure SSL/TLS support for data encryption

This compatibility ensures that the Mos Eisley Cantina System is accessible across various devices and operating systems, meeting the needs of all users..

2.2 Software

Programming language:

- .Net core Web API and React.js **Graphical user interfaces:**
- HTML,CSS and JavaScript for the GUI development and API's **Report generation and presentation:**
- Kibana, Elasticsearch, Serilog, Hangfire, PrometheusMetrics or any other report generator that you can present in an online Environment

APIs and Integration:

RESTful API: For communication between the front-end and back-end, enabling **smooth** data transfer and a modular structure.

2.3 Obstacles

Potential Risks and Solutions

Integration Challenges:

- **Risk:** Integrating the new System with Elasticsearch may lead to compatibility issues.
- **Solution:** Conduct a thorough analysis of current systems to ensure compatibility by using standard APIs and middleware. Regular testing throughout the development process will help identify and resolve integration issues early.

Data Security and Privacy:

- **Risk:** Protecting the security and privacy of users data is critical, as any breach could have serious consequences.
- **Solution:** Implement robust security measures, including encryption, secure authentication, and regular security audits. Adhere to data protection regulations and best practices to safeguard sensitive information.

Scalability:

- **Risk:** The system must be capable of handling a growing number of users and large volumes of data without performance degradation.
- **Solution:** Design the system with scalability in mind by utilizing cloud services and scalable architecture. Conduct load testing to ensure the system can manage peak usage effectively.

User Adoption:

- **Risk:** Users - including users and staff - may resist adopting the new system, preferring traditional paper-based methods.
- **Solution:** Provide comprehensive training and support for users. Develop an intuitive and user-friendly interface to facilitate the transition. Gather user feedback and make iterative improvements based on their suggestions.

Technical Issues:

- **Risk:** Unforeseen technical issues during development, such as bugs or performance bottlenecks, could delay the project.
- **Solution:** Employ agile development practices to allow for flexibility and efficient issue resolution. Regular code reviews, automated testing, and continuous integration will help maintain code quality and identify problems early.

Maintenance and Support:

- **Risk:** After deployment, the system will require ongoing maintenance and support to address issues and ensure smooth operation.
- **Solution:** Establish a dedicated support team and a clear maintenance plan. Provide regular updates and patches to keep the system secure and functioning optimally. By identifying these potential obstacles and implementing proactive solutions, we can effectively manage risks and ensure the successful development and deployment of the System.

3 Deployment

3.1 Installation

To deploy the Mos Eisley Cantina System, we will follow a structured and detailed process to ensure a smooth and efficient installation:

Preparation

1. **Server Setup**: Ensure that the server environment has all necessary hardware and software prerequisites installed.
2. **Network Configuration:** Verify network settings and ensure stable internet connectivity.

Deployment Method

Installation

- The system will be packaged and made available from a secure server.
- The client will receive a link along with detailed instructions for system operation.
- Alternatively, we can push the installation package directly to the client's server if remote access is provided.

Upload Application Files to Azure App Service:

- Deploy the application using one of the following methods:
    - **Azure Portal**: Use the App Service Deployment Center to upload files directly.
    - **Azure CLI**: Deploy the application with the right command
    - **GitHub Actions**: Set up a CI/CD pipeline to automate deployment.

1. **Database Setup on Azure SQL Database**:
    - **Provision a Database**:
        - Navigate to the Azure Portal and create a new **Azure SQL Database**.
        - Choose the desired performance tier.
    - **Import Schema and Data**:
        - Use Azure Data Studio or SQL Server Management Studio to import the initial database schema and seed data:

sql
Copy code
:connect <AzureSQLServerName>
:use <DatabaseName>
:run <SchemaFile.sql>

    - **Enable Firewall Rules**:
        - Allow access to the database from your application by configuring the firewall settings in the Azure SQL resource.
2. **Application Configuration on Azure**:
    - **Update Configuration Settings**:
        - Navigate to the **Configuration** section of your Azure App Service.
        - Add the necessary settings, such as connection strings, environment variables, and secrets:
            - Example:
                - **ConnectionStrings:Database**: Server=tcp:<AzureSQLServer>.database.windows.net;Database=<DatabaseName>;User ID=<Username>;Password=<Password>;
    - **Secure Sensitive Data**:
        - Store sensitive settings like database credentials in **Azure Key Vault** and link them to the App Service.
3. **Web Server Configuration -Automatic with Azure App Service**:

- Azure App Service automatically configures the underlying server to serve your application.
- If you need custom configurations, use the **App Service Extensions** or deploy a custom **Docker image**.

Testing on Azure

1. **Initial Testing**:
   - Access the application using the URL provided by Azure App Service.
   - Verify that all endpoints are accessible and functioning correctly.
   - Check database connectivity and ensure CRUD operations are working.
2. **Provide Access Credentials**:
   - Create administrative and user accounts in the system.
   - Share the URL and access credentials with stakeholders for testing.
   - Ensure all user roles and permissions are functioning as intended.
3. **Monitoring**:
   - Use **Azure Application Insights** to monitor application performance and diagnose issues.
   - Set up alerts for downtime or unusual traffic patterns.
4. **Backup**:
   - Enable automatic backups for the App Service and Azure SQL Database to ensure data integrity.

Updates and Maintenance

1. **Regular Updates:**
   - Periodic updates will be provided to enhance functionality, fix bugs, and improve security. The client will be notified about available updates, which can be downloaded and installed following the provided instructions.
2. **Automated Updates:**
   - For critical updates, we can implement an automated update mechanism to push updates directly to the system.

Support:

- Ongoing support will be available to assist with any issues or questions related to updates and maintenance.

By following these steps, we ensure that the Mos Eisley Cantina System is deployed efficiently and effectively, providing a seamless experience for the client and end-users. Regular updates and dedicated support will help maintain the system's reliability and performance.

3.2 Training

To ensure a smooth transition to the Mos Eisley Cantina System, we will provide comprehensive training and ongoing support for all users.

Initial Training:

1. **Training Sessions:**
    - We will conduct on-site or virtual training sessions for administrators, staff, and other key users. These sessions will cover all aspects of the system, including user registration, payment processing, and report generation.
2. **User Manuals:**
    - Detailed user manuals and quick reference guides will be provided to assist users in navigating the system and performing common tasks.
3. **Video Tutorials:**
    - We will create and distribute video tutorials demonstrating key functionalities and processes within the system.

Ongoing Support:

1. **Helpdesk Support:**
    - A dedicated helpdesk support service will be available via phone, email, or chat to address any questions or issues users may encounter.
2. **Online Resources:**
    - We will maintain an online knowledge base with FAQs, troubleshooting tips, and step-by-step guides to help users resolve common issues independently.

Post-Sign-Off Training and Support:

1. **Refresher Training:**
    - Periodic refresher training sessions will be scheduled to ensure users remain proficient with the system and are aware of any new features or updates.
2. **Advanced Training:**
    - Additional training sessions will be provided for advanced users or administrators who require a deeper understanding of the system's capabilities.
3. **Feedback Mechanism:**
    - We will implement a feedback mechanism to gather user input and continuously improve our training and support services.

By offering comprehensive training and ongoing support, we aim to ensure that all users feel confident and capable of effectively using the System, thereby maximizing the system's benefits.

3.3 Testing

To ensure that the Mos Eisley Cantina System meets all requirements and functions correctly, we will conduct thorough testing in collaboration with the client. The testing process will include the following steps:

Initial Testing:

- **Developer Testing**: Our development team will perform unit testing and integration testing to verify that individual components and their interactions work as expected.
- **Automated Testing:** We will use automated testing tools to conduct regression tests, ensuring that new changes do not introduce any bugs.

Client Involvement:

- **Test Plan:** We will provide the client with a detailed test plan outlining the testing phases, objectives, and timelines.
- **Test Cases:** A comprehensive set of test cases covering all functionalities of the system will be shared with the client. These test cases will guide the client in systematically testing the system. User Acceptance Testing (UAT):
- **Client Testing:** The client will conduct User Acceptance Testing to validate that the system meets their requirements and expectations. This process involves testing the system in a realworld scenario using actual data.
- **Feedback Collection:** We will gather feedback from the client and end-users during UAT to identify any issues or areas for improvement.

Issue Resolution:

- **Bug Tracking:** A bug tracking system will be employed to document and prioritize any issues found during testing.
- **Fixes and Updates:** Reported issues will be addressed promptly, and updates will be provided to the client for re-testing.

Final Approval:

- **Sign-Off:** Once all issues are resolved and the client is satisfied with the system's performance, we will obtain formal sign-off from the client.
- **Documentation:** We will provide comprehensive documentation of the testing process, including test results and any changes made based on feedback.

Post-Deployment Testing:

- **Monitoring:** After deployment, we will closely monitor the system to ensure it operates smoothly in the live environment.
- **Support:** We will offer immediate support to address any unforeseen issues that may arise post-deployment.

By involving the client in the testing process and ensuring thorough validation at each stage, we aim to deliver a reliable and high-quality Mos Eisley Cantina System that meets all requirements and expectations.

3.4 Backup and recovery

Regular Backups:

- **Database Backups:** Will be Schedule daily to capture all recent changes and transactions. These backups will be securely stored on an off-site server.

Secure Storage:

- Storing backups in a secure, encrypted format to protect against unauthorized access. Backups will be kept in multiple locations, including off-site storage, to safeguard against data loss due to physical damage or disasters.

Backup Retention Policy:

- Implementation of a retention policy to maintain multiple versions of backups over a specified period. This will allow recovery from different points in time, providing flexibility in case of data corruption or other issues.

Recovery Strategy:

- **Recovery Plan:** Develop a detailed recovery plan that outlines the steps to restore the system and data from backups. This plan will include procedures for various recovery scenarios, such as full system recovery, partial data recovery, and disaster recovery.
- **Testing Recovery Procedures:** Regularly test the recovery procedures to ensure they function as expected. This includes conducting mock recovery drills to validate the effectiveness and efficiency of the recovery plan.

By implementing these backup and recovery measures, we aim to protect Mos Eisley Cantina System and its data, ensuring that the system remains reliable and resilient in the face of potential issues.

3.5 Documentation

The following documents will accompany the Mos Eisley Cantina System to ensure that users have all the necessary information for installation, usage, and troubleshooting:

1. **Installation Guide:**
- Detailed instructions for setting up the system on the server.
- Steps for configuring the web server and database.
- Available in both printed format and as a downloadable PDF from the TFG intranet.
2. **User Manual:**
- A comprehensive guide covering all functionalities of the system.
- Instructions for user registration, API testing, and report generation.
- Available online in the system's help section and as a downloadable PDF.
3. **Quick Reference Guide:**
- A concise guide with step-by-step instructions for common tasks.
- Designed for quick access and ease of use.
- Available online and as a printed booklet.
4. **Administrator Guide:**
- Detailed instructions for system administrators on user management, settings configuration, and system maintenance.
- Available as a downloadable PDF from the TFG intranet.
5. **Backup and Recovery Guide:**

- Instructions for performing regular backups and restoring the system from backups.
- Available online and as a downloadable PDF.

6. **Troubleshooting Guide:**
- Information on common issues and their solutions.
- Available online in the system's help section.

7. **Training Materials:**
  - Video tutorials and recordings of training sessions.

8. **API Documentation:**
- Detailed documentation of any APIs used or provided by the system(Swagger).
- Available online for developers and technical staff.

All documentation will be available in both printed and digital formats to ensure accessibility for all users. The digital versions will be hosted on the TFG intranet and in the system's help section, providing easy access for users whenever needed.

4 Reporting and Reviews

The Mos Eisley Cantina System will include a comprehensive reporting and reviews system to support Management Information Systems (MIS) needs. The following reports will be available:

Details of all users registered for the system's services.
Includes names, username, contact information, and role.
Available in both summary and detailed formats.

Site visits Report:

Analysis of site usage usage, including the number of users per IP address to trace and set targets for subscribers.
Documentation of any incidents or issues reported during systems operation.
Includes details of the incident, involved parties, and actions taken.

Maintenance Report:

Schedule and records of system maintenance activities.
Includes dates, types of maintenance performed, and any issues found.

User Activity Report:

Logs of user activities within the system.
Includes login times, actions performed, and any changes made.
Useful for auditing and ensuring system security.

Custom Reports:

Ability to generate custom reports based on specific criteria.
Users can select data fields and filters to create tailored reports for their needs.
All reports will be accessible through the system's reporting module and can be exported in various formats such as PDF, Excel, and CSV for further analysis and record-keeping. This robust reporting

system will provide valuable insights and support decision-making processes for the TFG administration.

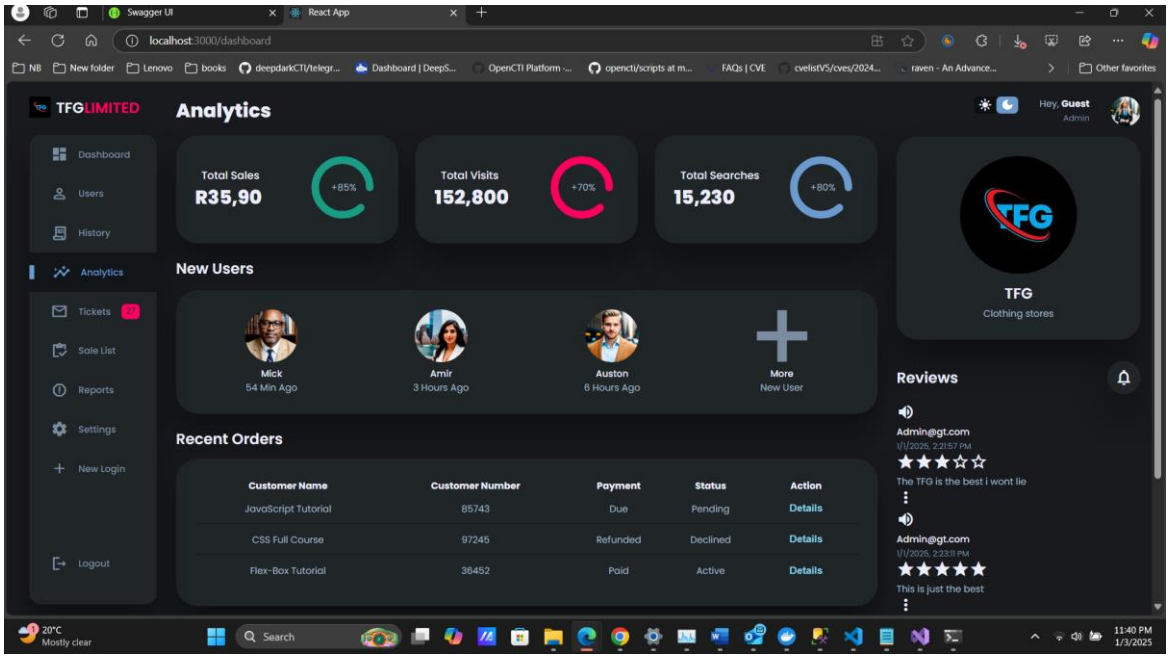
5 Evidence

5.1.1 1. Mos Eisley Cantina Dashboard

Purpose: This screen serves as the central hub for administrators to view and manage all Mos Eisley Cantina activities. It provides a summary of the Reviews, site visits, total sales, total searches, and other key statistics and analysis.

Look and Feel: The dashboard is designed with clear sections, using vibrant yet professional colors to highlight key metrics. The interface is intuitive, allowing users to easily navigate through different functionalities.

Screenshot:



Purpose: This form allows the admin to view the reviews on the dashboard analytics

6. Initial draft

6.1 Section A

I. Programming Languages

- Back end .Net core Web API
- front end Reactjs
- Report generation and presentation: Kibana, Elasticsearch, Serilog, Hangfire, PrometheusMetrics or any other

II. Database

- Database: Microsoft SQL Server

6.2 Section B

I. Use Case

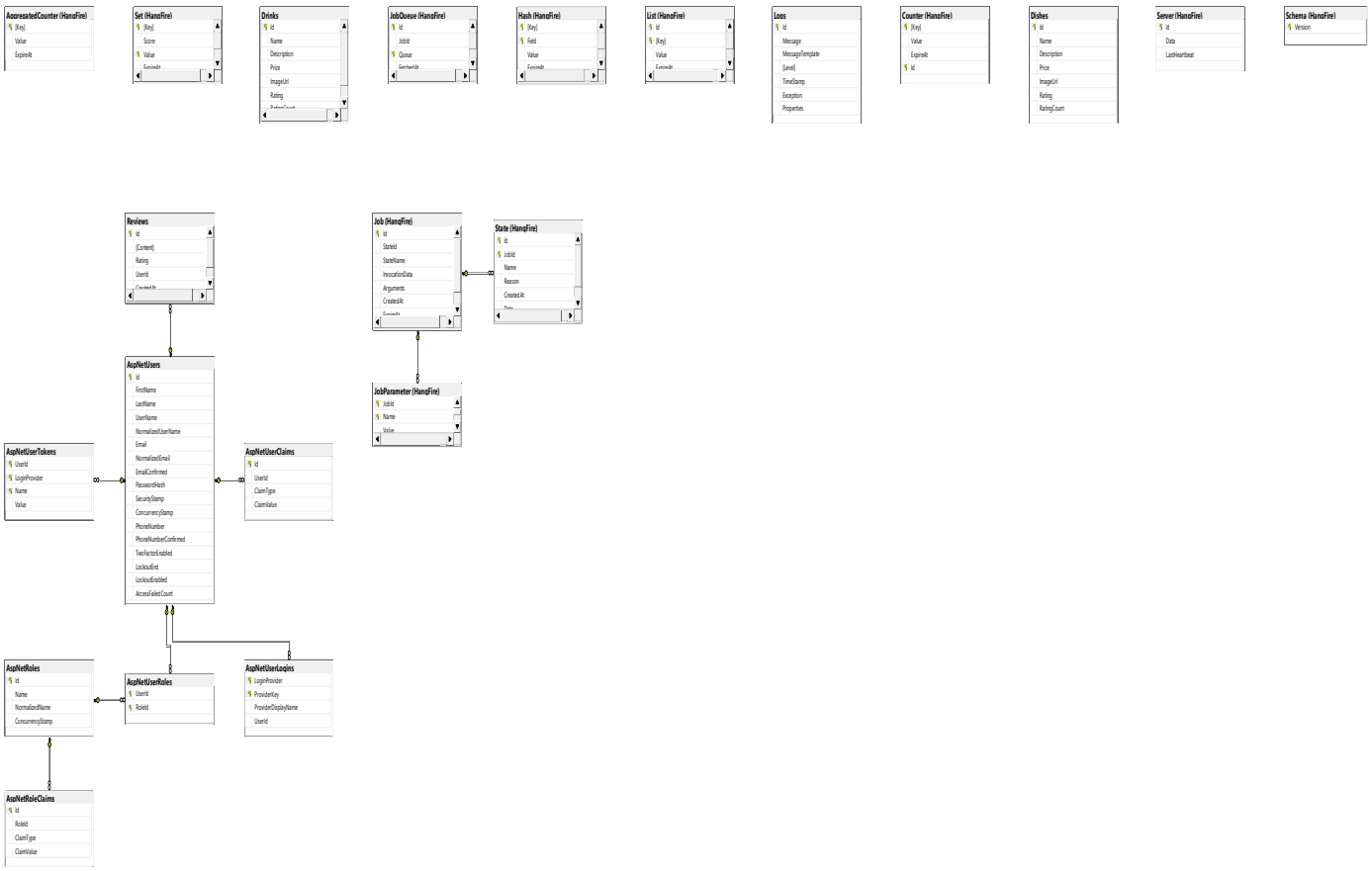
Use Case Name	Actor	Description
Manage Dishes	Admin	CRUD operations for dishes.
Manage Drinks	Admin	CRUD operations for drinks.
Manage Reviews	Admin	View or delete customer reviews.
Search Dishes or Drinks	Customer	Search for dishes or drinks.
Rate Dishes or Drinks	Customer	Provide ratings for items.
View Item Details	Customer	View details of a specific item.
View system Logs	Admin	View and analyze system logs(serilog)
Monitor and maintain system performance and threats	Admin	View system analytics on Elasticsearch(kibana), PrometheusMetrics and hangfire
Login	Admin/Customer	Authenticate and access the system.

II. Class Diagram for Mos Eisley Cantina

Class Name	Attributes	Methods	Relationships
User	- Id: int - Name: string - Email: string - PasswordHash: string - Role: string (Admin or Customer)	- Login(email: string, password: string): bool - Register(user: User): void	1..* association with **Review**: A user can write multiple reviews.
Dish	- Id: int - Name: string -	- AddDish(dish: Dish): void - UpdateDish(dishId: int, updatedDish: Dish):	0..* association with **Review**:

	Description: string - Price: decimal - ImageUrl: string - Ratings: List<int>	void - DeleteDish(dishId: int): void - CalculateAverageRating(): double	A dish can have multiple reviews.
Drink	- Id: int - Name: string - Description: string - Price: decimal - ImageUrl: string - Ratings: List<int>	- AddDrink(drink: Drink): void - UpdateDrink(drinkId: int, updatedDrink: Drink): void - DeleteDrink(drinkId: int): void - CalculateAverageRating(): double	0..* association with **Review**: A drink can have multiple reviews.
Review	- Id: int - DishOrDrinkId: int (FK to Dish or Drink) - UserId: int (FK to User) - Comment: string - Rating: int	- AddReview(review: Review): void - DeleteReview(reviewId: int): void	Linked to **User**, **Dish**, and **Drink**: Represents a review by a user for a dish or drink.
SearchService	None	- SearchDishes(query: string): List<Dish> - SearchDrinks(query: string): List<Drink>	Utilizes **Dish** and **Drink** for searching functionality.
RatingService	None	- RateDish(dishId: int, rating: int): bool - RateDrink(drinkId: int, rating: int): bool	Depends on **Dish** and **Drink** for calculating average ratings.

III. ERD Diagram



6.3 Section C (Backup and Recovery for the Database and Programming code)

i. Backup and Recovery Software for the Database

- Microsoft SQL backup

ii. Backup and Recovery process for the Programming code

- BackupPC
- Git / GitHub

7 **API Documentation**

This API allows you to interact with and manage the Mos Eisley Cantina's operations, including managing orders, Reviews,Logs, Ratings and customers.

Base URL

Production: https://api.moseisleycantina.com

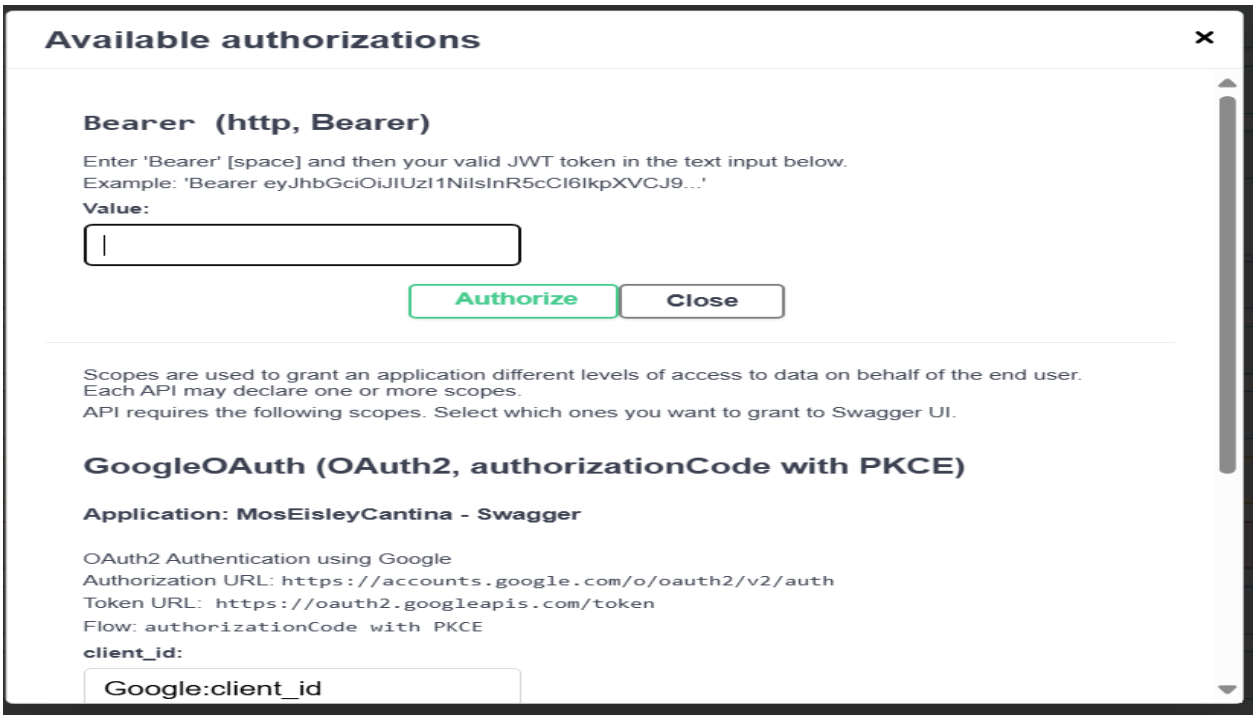
Development: https://dev.api.moseisleycantina.com

Authentication

The Mos Eisley Cantina API uses **JWT-based authentication**.

Authentication Headers:

- Authorization: Bearer <your_token>



To obtain a token, use the /Auth/login endpoint.

Endpoints

1. **Authentication**

POST api/Auth/login

Description: Logs in a user and returns a JWT token.

Request Body:

```
{
  "email": "user@example.com",
  "password": "stringst"
}
```

Response:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYW1tcy9uYW1laWRlbnRpZmllciI6IjFlMGQyN2M4LTRkNDYtNDF1Ny04Y
CIsImh0dHA6Ly9zY2hlbWFzLnhtbHNvYXAub3JnL3dzLzIwMDUvMDUvaWRlbnRpdHkvY2xhaW1zL25hbWUiOiJBZG1pbk8ndC5jb20iLCJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWl
iOiJBZG1pbk8ndC5jb20iLCJodHRwOi8vc2NoZW1hcy5taWNyb3NvZnQuY29tL3dzLzIwMDgvMDYvaWRlbnRpdHkvY2xhaW1zL3JvbGUiOiJBZG1pbiIsImV4cCI6MTczNjA5NzI2MiwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo4M
DovL2xvY2FsaG9zdDo4MDgwIn0.n5RRgUhDKb91UeIAvLI8xeO5rX8iRz7E0D8oyGmrS60",
  "ipAddress": "::ffff:172.19.0.1"
}
```

POST api/Auth/register

Description: Registers a new user.

Request Body:

```
{
  "email": "user@example.com",
  "password": "string",
  "firstName": "string",
  "lastName": "string",
  "userName": "string",
  "role": "string"
}
```

Response:

```
{
  "message": "User registered successfully."
}
```

---

1. **Dishes Please NOTE the drinks api follows the same procedure.**

GET api/Dishes (Admin/User)

Description: Retrieves a list of all dishes.

Response:

```
[
  {
    "name": "Tacccoo",
    "description": "A delicious taco with ground beef, lettuce, cheese, and salsa.",
    "price": 675.99,
    "imageUrl": "https://via.placeholder.com/150/FF0000/FFFFFF?text=Taco"
  },
  {
    "name": "Pizza",
    "description": "A cheesy pizza with pepperoni and a crispy crust.",
    "price": 112.99,
    "imageUrl": "https://via.placeholder.com/150/0000FF/FFFFFF?text=Pizza"
  },
```

POST api/Dishes (Admin)

Description: Creates a dish.

Request Body:

```
{
  "name": "string",
  "description": "string",
  "price": 10000,
  "imageUrl": "string"
}
```

Response:

```
{
  "name": "MCDonds pie",
  "description": "A cheesy pie with pepperoni and a crispy crust.",
  "price": 67.99,
  "imageUrl": "https://via.placeholder.com/150/0000FF/FFFFFF?text=Pizza"
}
```

GET /api/Dishes/{id} (Admin/User)

Description: Retrieves the dish by ID.

Request Body:

Name	Description
id * required integer($int32) *(path)*	1

Response:

```
{
  "name": "Tacccoo",
  "description": "A delicious taco with ground beef, lettuce, cheese, and salsa.",
  "price": 675.99,
  "imageUrl": "https://via.placeholder.com/150/FF0000/FFFFFF?text=Taco"
}
```

PUT /api/Dishes/{id} (Admin)

Description: Updates the dish.

Request body:

Name	Description
id * required integer($int32) *(path)*	id

Request body

Example Value | Schema


```
{
  "name": "string",
  "description": "string",
  "price": 10000,
  "imageUrl": "string"
}
```


Response:


```
{
  "message": "Dish was updated successfully."
}
```

DELETE /api/Dishes/{id} (Admin)

Description: Deteles or remove the dish record.

Request body:

Name	Description
id * required integer($int32) *(path)*	id

Response:


```
{
  "message": "Dish was deleted successfully."
}
```

GET /api/Dishes/search (User)

Description: Retrieves a list of available dishe by name or latter.

Request body:

Name	Description
query string *(query)*	Pizza

Response:


```
{
  "name": "Pasta",
  "description": "Pasta with marinara sauce and fresh basil.",
  "price": 119.99,
  "imageUrl": "https://via.placeholder.com/150/FFFF00/FFFFFF?text=Pasta"
}
```


POST /api/Dishes/{dishId}/rate

Description: Rate a dish.

Request Body: NOTE the dishId entered must be same as itemId

Name	Description
dishId * required integer($int32) *(path)*	6

Request body


```
{
  "itemId": 6,
  "rating": 4
}
```


Name	Description
dishId * required integer($int32) *(path)*	1

Response:


```
{
  "message": "Rating added successfully."
}
```


GET /api/Dishes/{dishId}/rating

Description: Retrieves dish rating by ID.

Request Body:

Name	Description
dishId * required integer($int32) *(path)*	1

Response:


```
{
  "id": 1,
  "name": "Tacccoo",
  "averageRating": 1,
  "ratingCount": 3
}
```


2. **Review**

GET /api/reviews

Description: Retrieves a list of all the reviews.

Response:


```
[
  {
    "id": 1,
    "content": "The TFG is the best i wont lie",
    "rating": 3,
    "userName": "Admin@gt.com",
    "createdAt": "2025-01-01T14:21:57.3648431"
  },
  {
    "id": 2,
    "content": "This is just the best",
    "rating": 5,
    "userName": "Admin@gt.com",
    "createdAt": "2025-01-01T14:23:11.1239356"
  },
```


POST /api/reviews

Description: Adds a new review.

Request Body:

Name	Description
content string *(query)*	This was just amazing
rating integer($int32) *(query)*	3

Response:

```
Review added successfully.
```

GET /api/reviews/{id}

Description: Retrieves reviews by ID.

Request Body:

Name	Description
id * required integer($int32) *(path)*	1

Response:


```
{
  "id": 1,
  "content": "The TFG is the best i wont lie",
  "rating": 3,
  "userName": "Admin@gt.com",
  "createdAt": "2025-01-01T14:21:57.3648431"
}
```


3. **Logs**

GET /api/Logs/all

Description: Retrieves Logs by pagination.

Request Body:

Name	Description
page integer($int32) (query)	*Default value* : 1 1
pageSize integer($int32) (query)	*Default value* : 100 100

Response:


```
{
  "totalLogs": 47785,
  "page": 1,
  "pageSize": 100,
  "logs": [
    {
      "id": 1,
      "message": "Entity Framework Core \"9.0.0\" initialized '\"AppDbContext\"' using provider '\"Microsoft.EntityFrameworkCore.SqlServer\":\"9.0.0\"' with options: \"Engine
\"",
      "messageTemplate": "Entity Framework Core {version} initialized '{contextType}' using provider '{provider}:{providerVersion}' with options: {options}",
      "level": "Debug",
      "timeStamp": "2025-01-03T21:16:56.197",
      "exception": null,
      "properties": "<properties><property key='version'>9.0.0</property><property key='contextType'>AppDbContext</property><property key='provider'>Microsoft.EntityFramework
operty><property key='providerVersion'>9.0.0</property><property key='options'>EngineType=SqlServer </property><property key='EventId'><structure type=''><property key='Id'>1
operty key='Name'>Microsoft.EntityFrameworkCore.Infrastructure.ContextInitialized</property></structure></property><property key='SourceContext'>Microsoft.EntityFrameworkCore.
roperty><property key='Application'>MyApp</property><property key='Environment'>Production</property><property key='Version'>1.0.0</property></properties>"
    },
    {
      "id": 2,
      "message": "Creating DbConnection.",
      "messageTemplate": "Creating DbConnection.",
      "level": "Debug",
      "timeStamp": "2025-01-03T21:16:56.317",
      "exception": null,
      "properties": "<properties><property key='EventId'><structure type=''><property key='Id'>20005</property><property key='Name'>Microsoft.EntityFrameworkCore.Database.Con
Creating</property></structure></property><property key='SourceContext'>Microsoft.EntityFrameworkCore.Database.Connection</property><property key='Application'>MyApp</propert
```


4. **Error Codes**
   - 400: Bad Request (e.g., invalid input data).
   - 401: Unauthorized (e.g., missing or invalid token).
   - 403: Forbidden (e.g., insufficient permissions).
   - 404: Not Found (e.g., resource not found).
   - 500: Internal Server Error.