**StoreAPI Manual**

**Index**

1. Introduction
2. Server-side Architecture
   1. Server-side Structure
   2. Database Layout
3. Client-side Architecture
4. Launch Process
   1. Cloud Deployment
   2. Application Encapsulation
5. Application Verification
   1. Server-side Verification
   2. Client-side Verification
   3. Load Testing
6. Security Measures and User Control
7. Intercommunication Features
8. Artificial Intelligence
9. Building and running the application
10. Encounterd challenges

**1. Introduction**

StoreAPI is a complete software solution designed to effectively oversee a group of stores along with their staff, roles, and schedules. The software provides an all-encompassing way to arrange various employee shifts within each store.

**2. Server-side Architecture**

Server-side

The server-side structure of the software takes care of all server-end operations. It's developed using C# with .NET 7 and provides a REST API, usable through both Swagger UI and Postman. The implemented features include CRUD operations, filtration, and statistical reporting.

Database

The server-side API communicates with a Microsoft SQL Server 2022 database. The database consists of a substantial dataset, with one million records each for roles, employees, and stores, and ten million records for shifts. A Python script generates a SQL file that inserts large amounts of data into the database using the Faker library. Database migrations handle adjustments to the database schema.

**3. Client-side Architecture**

The client-side is a Single Page Application (SPA) built using the React framework in conjunction with Typescript, Vite, Node.js, and Material UI for the user interface components. This element is hosted on Netlify and can be visited at <https://sdistoreapi.netlify.app/>.

The client-side design adapts smoothly to any display size, doesn't include scrollbars, and ensures the correct sizing and arrangement of components. Users can browse paginated data and adjust the page size via their profile settings.

**4. Launch Process**

Cloud Deployment

The entire software is deployed on a Google Cloud Ubuntu-based virtual machine, reachable at <https://storeapi.ddns.net/api>. The virtual machine houses Microsoft SQL Server, .NET 7 SDK, and nginx (set up with a reverse proxy to protect the application server from direct internet exposure).

A Kubernetes Cluster is installed on the cloud for dynamic resource scaling. The scaling process can be observed using JMeter with the Ultimate Thread Group plugin, displaying how the system responds to accommodate JMeter requests.

Application Encapsulation

The software has been containerized on a separate Ubuntu-based virtual machine. There are two distinct configurations provided: one for development and one for production use. Based on the user's choice, selecting 'Debug' starts the development version of the software, while 'Release' triggers the production version.

**5. Application Verification**

Server-side Verification

The server-side is confirmed through unit tests designed with the NUnit framework and the Moq library for mocking data.

Client-side Verification

The client-side undergoes End to End (E2E) testing with Cypress for intercepting server-side requests and mocking data.

Load Testing

Software performance is measured via stress testing with spikes using Apache JMeter. The resulting diagrams show how user counts affect CPU utilization and response times. The test starts with a few users: approximately 2 constant ones and roughly 10 for the spikes, and gradually increases until 100% CPU utilization to evaluate how the system copes with sudden load increases.

**6. Security Measures and User Control**

The system employs a logging mechanism using JWT tokens. Passwords are securely hashed using the SHA-256 algorithm. The system outlines different roles for users: anonymous users (no login) who can only view content, standard logged-in users who can add and edit their entries, moderator users who can add entries and modify all records, and admin users who can add entries, modify all records, change user roles (including other admins), access the admin panel for bulk data operations, and alter the page preferences for all users.

**7. Intercommunication Features**

The software includes a chat page implemented with web sockets, enabling all visitors to view others' messages. Messages stay in the system, allowing for a message history. Visitors can also choose a nickname when entering the chat.

**8. Artificial Intelligence**

The software incorporates a Machine Learning model that predicts an employee’s salary based on their role and tenure. This model, developed using the Fast Tree algorithm, was trained on a custom dataset with one hundred thousand rows. The machine learning model is integrated into the software using the ML.NET framework.

**9. Building and running the application**

To launch the app, follow these instructions:

1. Installation: Make sure Visual Studio 2022, .NET 7, and Entity Framework are installed on your system.

2. Clone the Repository: Use the git clone command on the <https://github.com/UBB-SDI-23/lab-5x-davidcristian> link to obtain a copy of the repository.

3. Switch to the **development** branch: Go to the cloned repository and switch to the development branch using the command git checkout development.

4. Open the project’s solution file in Visual Studio.

5. Restore the NuGET package dependencies by using the Build > Clean solution and the Build > Rebuild solution menu options.

6. Add the database connection details inside of the appsettings.json file. The applcation comes with a built-in JWT secret for development purposes, which you should change.

7. Start the application. The Swagger UI will open, signaling that the application started successfully.

8. Open the front-end project folder in your preferred editor. For the purposes of this example we will be using Visual Studio Code.

9. Modify the URLs for the back-end in the constants.ts file, if needed.

10. Run the **npm run dev** command to start the front-end.

Additionally, the entire back-end application, including the database server, can be started using the **docker-compose up --build** command inside of the Docker directory. This skips every step and results in easy deployment.

**9. Encountered challenges**

There were some hurdles encountered while developing the application. For example, the initial Machine Learning model was developed with the help of Jupyter Notebook, using Python 3.10. The model was making use of Pytorch, which unfortunately cannot be imported inside C# even after being saved in the compatible .cnnx format. Multiple attemps at using different libraries, even sklearn, resulted in the same issue. Unfortunately, we had to settle and use Microsoft’s ML.NET solution to build the model, which is less than ideal.

Of course, we could have used the initial model by executing a python script though a shell command, but this has several issues. First, it introduces additional abstraction by adding yet another layer we had to worry about. This includes sanitizing input, and making sure all components are up to date to prevent exploits. Second, not only would this have been an extremely bad practice, but it would have been difficult to include this in the containerized version of the application due to constraints. Using ML.NET was the best available option.

Additionally, creating the Kubernetes cluster to handle automatic horizontal pod scaling proved being rather challenging. Extensive research had to be done, and multiple failed attempts had to be sat through. The result was worth it due to it being a true enterprise solution that saves costs and headaches for not only the engineers, but also the end users.

Lastly, multiple cloud solutions had to be used during the development life cycle of the project. Initially, the project was deployed as a Web App, including a Cloud Database, on Microsoft Azure. This proved to be less than ideal due to the extreme costs of the service. Once the credit expired, we settled on using Virtual Machines inside Google Cloud. A much better option.