eSpaza

Sprint 1 Technical Documentation

By Team Snorlax

1. Introduction

The e-spaza project is designed to equip spaza shop owners with a platform to efficiently manage their stock and provide shoppers with the ability to order from multiple shops and optimize their shopping journey. The front-end of this web-based application was developed using React, Bootstrap, and TypeScript, ensuring a responsive, user-friendly interface that meets the needs of both shop owners and shoppers.

2. Frontend

2.1 Technology stack

1. **React**: We chose React as the core framework for the frontend due to its robust component-based architecture, which allows for reusable UI components and efficient state management. This choice facilitated the rapid development of a dynamic and interactive user interface.
2. **Bootstrap**: Bootstrap was integrated to handle the styling and layout of the application. Its pre-built components and responsive grid system allowed us to quickly create a consistent and responsive design across different devices, ensuring a seamless user experience. We extended Bootstrap with the use of CSS where we could not achieve the preferred result.
3. **TypeScript**: TypeScript was utilized to add type safety to the JavaScript code, which enhanced code maintainability and reduced the likelihood of runtime errors

2.2 Architecture

1. **Component Structure:** The front-end architecture is organized around a set of reusable React components. These components are divided into presentational components, responsible for rendering UI elements, and container components, which handle data fetching and state management.
    a. **Presentational Components:** These include buttons, forms, and list items that are styled with Bootstrap and handle user interactions.
    b. **Container Components:** These components manage the application state, often using React's useState hooks, and interact with the backend to fetch data.
    c. **Component Reusability:** The Header and other components were designed to be reusable across the application. This enhances code maintainability and scalability. Bootstrap provides a set of prebuild components that were used, for example the cards where items were shown. The Bootstrap components were also utilized to create more complex components such as a header or side navigation bar.
2. **Routing:** The application employs React Router, managed through the RoutesManager component, to handle navigation between different views, such as the shopping page, order management page, and admin dashboard. Each route corresponds to a specific component, with route-specific data fetching.
3. **State Management:** The state of the application is managed primarily through React’s Context API, which allows for global state management without the need for third-party libraries.

4. **Responsive Design:** Bootstrap's responsive grid system and utility classes were extensively used to ensure that the application is fully responsive. This approach ensures that users have a consistent experience.
5. **Code Organization:** The frontend codebase is organized into key directories to enhance maintainability and scalability. The ‘components/’ directory contains reusable React components. The ‘pages/’ directory houses main views, assembling components to create complete UI pages. The ‘styles/’ folder contains global and component-specific styles, while ‘assets/’ stores images and other static resources. This structure promotes consistency and ease of navigation across the project.

2.3 Third-Party Services and Libraries

1. **React-Bootstrap:**
    a. **Description:** React-Bootstrap provides Bootstrap components tailored for React, enabling us to leverage Bootstrap’s styling and layout capabilities within a React component-based architecture.
    b. **Key Features:**
        i. **Pre-Built Components:** We utilized a range of React-Bootstrap components, such as buttons, modals, and navigation bars, which are responsive and customizable, ensuring a cohesive look across devices.
        ii. **Responsive Grid System:** The React-Bootstrap grid system was crucial in creating layouts that adapt smoothly to different screen sizes.
2. **Vercel Frontend hosting:**
    a. Description:
        i. Vercel is a cloud platform designed to simplify the deployment and scaling of modern front-end applications. It is known for its seamless integration with frameworks like Next.js, providing an optimized environment for static and dynamic site generation.
        ii. Vercel is ideal for projects requiring quick iteration, seamless deployment, and global distribution. It is well-suited for web applications, static sites, and serverless API endpoints.
    b. Key Features:
        i. Continuous Deployment: Automatically builds and deploys your application every time you push code to your repository.
        ii. Global CDN: Ensures fast load times and low latency by serving content from a global network of servers.
        iii. Serverless Functions: Supports backend logic through serverless functions, allowing for dynamic content generation and API handling.
        iv. Preview Deployments: Every pull request automatically gets its own preview URL, making it easy to test changes before they go live.
    c. Implementation Details:
        i. **Frontend Integration:** The frontend application, built with a modern framework (e.g., React), will be deployed on Vercel. The platform’s features like automatic HTTPS, global CDN, and environment variables will be utilized.

2.4 CI/CD Pipeline

As we explore the implementation of a CI/CD pipeline for our React and Node.js web application, we are currently in the research phase, particularly focusing on the integration and utilization of GitLab's CI/CD capabilities. Our preliminary plan involves the creation of a .gitlab-ci.yml file at the root of our project repository. This file is crucial as it defines the stages of our pipeline, which typically include:

1. **Install Stage**: Here, we specify commands to install necessary dependencies. This setup ensures that our application has all the required packages to run tests and builds successfully.
2. **Test Stage**: At this stage, our focus will be on running unit tests to validate the functionality and integrity of our code.
    a. We are considering using Jest. Jest provides a robust testing framework that integrates well with React, allowing us to simulate and test various user interactions and component behaviors.
    b. We plan to configure our package.json to include a test script that Jest can execute. This script will encompass tests that render components, check build compilation, and perform assertions to ensure components behave as expected under simulated user actions.
3. **Build Stage**: This stage will be responsible for compiling the React application for production deployment. We will define jobs within the .gitlab-ci.yml file that handles the compilation process, ensuring that our application is built according to the defined specifications.

3. API

3.1 Overview

Our web application leverages a **GraphQL API** built with **Apollo Server** to facilitate communication between the front-end and back-end. The API serves as the central interface between the client-side React application and the PostgreSQL database, providing flexible, efficient, and type-safe data interactions.

3.2 Endpoints and Usage

1. **GraphQL Endpoint:**
    a. **URL**: https://ourapi.com/graphql
        i. All requests (queries, mutations, and subscriptions) are sent to this single GraphQL endpoint.
    b. **Methods Supported**: POST (GraphQL queries and mutations are sent as POST requests).
2. **Queries:**
    a. **Purpose**: Used to fetch data from the server.
    b. **Usage**: Used by the front-end React app to populate user-related data on the UI.
3. **Mutations**
    a. **Purpose**: Used to create, update, or delete data on the server.
    b. **Usage**: Used when a new user signs up through the front-end, triggering the creation of a user record in PostgreSQL.
4. **Subscriptions**
    a. **Purpose**: For real-time updates (e.g., notifications).
    b. **Response**: The server pushes real-time updates to the client whenever a new user is created.
    c. **Usage**: Used for features like live notifications or updates without requiring manual refreshes.

3.3 Third-Party Services and Libraries

1. **Barcode Recognition with QuaggaJS:**
    a. **Description:** QuaggaJS is an open-source JavaScript library for barcode scanning, providing efficient and accurate barcode recognition for web applications. Our project uses QuaggaJS to enable seamless barcode scanning capabilities.
    b. **Key Features**:

i. **Static Image Scanning:** Supports barcode detection from static images, offering versatility for various use cases.
ii. **Multi-Format Support:** Decodes multiple barcode formats, including Code 128, EAN, and UPC, ensuring broad compatibility.

c. **Implementation Details:**
i. **Camera Access:** Utilizes the device's camera via the WebRTC API to capture images or video streams.
ii. **Image Processing:** QuaggaJS processes captured images, applying recognition algorithms to detect and decode barcodes.
iii. **Integration:** Integrated into our React application with TypeScript, ensuring type safety and development efficiency.
iv. **Data Recording:** Upon successful barcode detection, QuaggaJS returns the decoded barcode data, which is then saved to the database for inventory management and item tracking.

2. **Overview of Auth0:**
a. **Description:** Auth0 is a cloud-based authentication and authorization service that simplifies user management.
b. **Key Features:**
i. It offers secure login methods, including social logins, multi-factor authentication, and single sign-on. Auth0 issues JSON Web Tokens (JWT) for authenticated users, which are used to secure API requests.
c. **Implementation Details:**
i. **Frontend Integration:** The React app will use the Auth0 React SDK to handle user authentication. The main app component will be wrapped with Auth0's provider, allowing us to manage login, logout, and user sessions easily.
ii. **Backend Integration:** The backend will verify JWT tokens from Auth0 to secure GraphQL endpoints. Middleware will be added to ensure only authenticated users can access sensitive data.
iii. **Database Security:** Access to the PostgreSQL database will be restricted through the backend, ensuring only verified users can interact with it.

3. **HERE Maps API**
a. Description: HERE Maps API is a robust location-based service platform offering advanced mapping, geolocation, and routing functionalities. It provides the tools necessary to deliver precise geographic data, which can enhance various user experiences, especially in location-dependent services.
b. **Key Features:**
i. Mapping and Geocoding: Provides detailed maps and the ability to geocode addresses (convert addresses into coordinates) and reverse geocode (coordinates to addresses).
ii. Routing and Navigation: Offers optimal routing with real-time traffic data, turn-by-turn navigation, and multi-modal transport options.
iii. Weather Information: Integrates weather data into maps, offering insights into current and forecasted weather conditions, which can be critical for logistics and planning.
iv. Customization and Global Coverage: Enables customized map styles, layers, and interactive features with extensive global coverage.
c. Implementation Details:

i. **Frontend Integration:** The HERE Maps JavaScript API will be embedded into the client shops hosted on our platform, allowing shop owners to display their locations, **provide customer navigation, and manage delivery routes.**
ii. **Weather Functionality:** The API will include weather overlays on maps, showing current and predicted weather conditions. This feature will be used to inform customers of potential delays in delivery due to adverse weather conditions.

d. **Possible use Cases:**
i. Shop owners can use this to enhance customer experience by providing accurate location data, route planning, and real-time updates on weather-related delays.
ii. Customers will benefit from knowing if weather conditions might impact on their deliveries, helping manage expectations and improving overall service transparency.

4. Backend

4.1 Technology Stack

1. **GraphQL**: Used for querying and mutating data. GraphQL allows the front-end to request exactly the data it needs, reducing over-fetching and enabling highly optimized and flexible data flows.
2. **Apollo Server**: Apollo Server powers our GraphQL API. It handles incoming GraphQL requests from the client, resolves them through defined schema and resolvers, and interacts with the PostgreSQL database.
3. **TypeORM**: We utilize TypeORM as an Object-Relational Mapping (ORM) library to interact with the PostgreSQL database, ensuring smooth integration with TypeScript and providing a straightforward way to map between database tables and application models.

4.2 Database Design

The database schema consists of several tables that represent the core entities of the e-Spaza app. The tables are designed to store information about users, shops, items, orders, inventory, categories, notifications, and the relationships between these entities.

4.2.1 Tables and Relationships:

1. **Roles**: This table stores the distinct roles that users can have in the system.
    a. It has a primary key role_id and a unique role_name column.
2. **Users**: This table stores user information.
    a. It has a primary key user_id, a foreign key role_id referencing the Roles table, a name column, a unique email column, and a password_hash column for securely storing hashed passwords.
    b. The foreign key establishes a many-to-one relationship between Users and Roles.
3. **Shops:** This table stores information about shops.
    a. It has a primary key shop_id, a name column, an optional address column, and a foreign key owner_id referencing the Users table.
    b. The foreign key establishes a many-to-one relationship between Shops and Users, indicating that a user can own multiple shops.
4. **Items:** This table stores information about items available in the shops.
    a. It has a primary key item_id, a name column, an optional description column, a price column, a foreign key category_id referencing the Categories table, and a foreign key shop_id referencing the Shops table.

    b. The foreign keys establish many-to-one relationships between Items and Categories, and between Items and Shops.
5. **Orders:** This table stores information about user orders.
    a. It has a primary key order_id, a foreign key user_id referencing the Users table, a total_price column, an order_date column with a default value of the current timestamp, and a status column with a default value of 'pending'.
    b. The foreign key establishes a many-to-one relationship between Orders and Users.
6. **OrderItems:** This table stores information about the items in each order.
    a. It has a primary key order_item_id, a foreign key order_id referencing the Orders table, a foreign key item_id referencing the Items table, a quantity column with a check constraint ensuring a positive value, and a foreign key shop_id referencing the Shops table.
    b. The foreign keys establish many-to-one relationships between OrderItems and Orders, Items, and Shops.
7. **Inventory:** This table stores information about the inventory of items in each shop.
    a. It has a primary key inventory_id, a foreign key shop_id referencing the Shops table, a foreign key item_id referencing the Items table, a quantity column with a check constraint ensuring a non-negative value, and a unique constraint on the combination of shop_id and item_id.
    b. The foreign keys and unique constraint ensure that each shop can have only one entry for each item in the inventory.
8. **Categories:** This table stores information about item categories.
    a. It has a primary key category_id and a unique name column.
9. **Notifications:** This table stores notifications for users.
    a. It has a primary key notification_id, a foreign key user_id referencing the Users table, a message column, and a created_at column with a default value of the current timestamp.
    b. The foreign key establishes a many-to-one relationship between Notifications and Users.
10. **UsersShops:** This table represents the many-to-many relationship between users and shops, indicating which users have access to which shops and their roles within those shops.
    a. It has a composite primary key consisting of user_id and shop_id, with foreign keys referencing the Users and Shops tables, respectively.
    b. It also has a foreign key role_id referencing the Roles table to specify the user's role within the shop.
11. **OrderShops:** This table represents the many-to-many relationship between orders and shops, indicating which orders belong to which shops.
    a. It has a composite primary key consisting of order_id and shop_id, foreign keys referencing the Orders and Shops tables, respectively.

4.3 Third-Party Services

1. **Hosting on Render.com:** Our PostgreSQL database is being hosted on Render.com, a cloud platform that also provides easy deployment for our database.
    a. **Interface:** It offers a simple and intuitive interface for deploying applications built with the various technologies our backend uses, including Node.js, Express, and PostgreSQL.
    b. **PostgreSQL:** Render provides managed PostgreSQL databases, making it easy to set up and connect the e-Spaza app backend to a reliable and scalable database.
    c. **Scalability**: Render offers automatic scaling based on the application's resource usage. As the traffic and demand increase, Render can automatically scale our app to handle the load, ensuring optimal performance and availability.

5. Team Communication and Branch Structure

5.1 Communication

1. **WhatsApp**: For quick and constant communication with each other, delivering updates quickly.
2. **Discord:**
    a. **Media:** Our client has created a discussion group that we use to send them any media relating to the project.
    b. **Internal Meetings:** Discord is used again for impromptu team meetings and discussions, ensuring everyone is aligned on tasks and progress.
    c. **Client Stand-ups:** We meet with the client every Thursday via Discord for stand-up meetings to provide updates and gather feedback.
3. **Tutorial Sessions:** We hold sessions at the beginning of every tutorial to discuss challenges and plan solutions, helping us stay on track.
4. **ClickUp:** Used to help us allocate tasks, add any bugs found to the to-do list, and helps us view the full scope of the project at hand.

5.2 Branch Structure

We follow a feature-based branch structure in our version control system to keep development organized and maintain code quality. Each feature or bug fix is developed in its own branch.

Once a feature is complete, a merge request is submitted for review. Team members review the code for quality and consistency, providing feedback if necessary. After approval, the branch is merged into the main branch, which serves as the stable version of the codebase. This process helps ensure that only well-tested and reviewed code is integrated, maintaining stability, and reducing the risk of bugs.