TEAM <04> SECTION <01>

(<<Bearcat crew>>)

<< Campus Inspection System>>

**Prepared By**

|  |  |
| --- | --- |
| Adarsh Kumar | S566774 |
| Felipe Medina Sato  Hitesh Sharma Allakki  Keerthi Priya Yakkala  Jeevitha Thangallapalli | S531355  S576886  S575013  S576433 |
|  |  |

Table of Contents

[1. Introduction 1](#_Toc177999301)

[2.Requirement Analysis 1](#_Toc177999302)

[2.1 Software Requirements 1](#_Toc177999303)

[2.2 Hardware Requirements 1](#_Toc177999304)

[2.3 Functional Requirements 1](#_Toc177999305)

[2.4 Non-Functional Requirements 1](#_Toc177999306)

[3. Programing languages and tools 1](#_Toc177999307)

[3.1 Frontend technologies 1](#_Toc177999308)

[3.2 backend technologies 1](#_Toc177999309)

[3.3 Framework 1](#_Toc177999310)

[4. Database implementation 1](#_Toc177999311)

[4.1 Entity Relationship Diagram 1](#_Toc177999312)

[4.2 Database design 1](#_Toc177999313)

[5. Testing 1](#_Toc177999314)

[5.1 Unit Testing 1](#_Toc177999315)

[5.2 System Testing 1](#_Toc177999316)

# 1. Introduction

The **Campus Inspection System** is a web-based application designed to modernize and streamline the process of building inspections at Northwest Missouri State University. As outlined in the Project Identification document, this system replaces a manual, paper-based workflow with a secure, reliable, and user-friendly digital platform. It provides a centralized hub for administrators to manage campus buildings, define custom inspection checklists, and oversee user accounts. The system also equips university technicians with a digital tool to conduct inspections efficiently, record findings accurately, and report issues in real-time. The primary objective is to enhance operational efficiency, improve data integrity, and ensure the overall safety and maintenance of campus facilities.

# 2.Requirement Analysis

## 2.1 Software Requirements

**Development Environment:**

* **Node.js (v20.19.4):** JavaScript runtime environment for executing the backend server.
* **npm (v11.5.2):** Package manager for installing and managing project dependencies for both the backend and frontend.
* **Angular CLI:** The command-line interface for creating, managing, and building the Angular frontend application.
* **Code Editor:** A modern integrated development environment (IDE) such as Visual Studio Code, IntelliJ
* **API Testing Tool:** A tool like Postman for testing and validating the backend API endpoints during development.
* **Git & GitHub:** For distributed version control and source code management.

**Deployment & Runtime:**

* **Web Browser:** Any modern browser (e.g., Google Chrome, Microsoft Edge, Safari) for accessing the web application.
* **PostgreSQL Server:** The relational database server required to store all application data.

## 2.2 Hardware Requirements

**Development Machine:** A standard computer or laptop with at least 8GB of RAM, a multi-core processor, and sufficient disk space for the project codebase, tools, and dependencies.

**End-User Device:** Any desktop computer, tablet, or mobile device with a modern web browser and a stable internet connection.

## 2.3 Functional Requirements

**Authentication & User Management:**

* **FR 1.1:** The system shall allow users to log in and out securely with role-based access (Admin or Technician)
* **FR 1.2:** The system shall allow Admins to create, view, update, and deactivate user accounts.

**Building & Checklist Management:**

* **FR 2.1:** The system shall allow Admins to create, view, update, and delete campus buildings.
* **FR 2.2:** The system shall allow Admins to create unique inspection checklists for each building, specifying data types (Numeric, Yes/No, Text).
* **FR 2.3:** The system shall allow Technicians to view a list of buildings and fill out its specific inspection checklist.
* **FR 2.4:** The system shall allow Technicians to submit completed checklists, automatically recording their identity and a timestamp.

**Reporting and History:**

* **FR 3.1:** The system shall allow Admins to generate reports by filtering inspection data by building, date range, or technician.
* **FR 4.1:** The system will allow Admins to see the data submitted by technicians.
* **FR 4.2:** The system shall allow Technicians to view a history of their own past submissions.

## 2.4 Non-Functional Requirements

**Performance:** The system must be fast and responsive, with pages loading in under 3 seconds and API calls completing in under 2 seconds, even with multiple concurrent users.

**Security:** All user data must be secure, with passwords being hashed and access to features strictly controlled by user roles (Admin vs. Technician).

**Usability:** The application must be intuitive and easy to use with minimal training, and it must be fully functional on modern web browsers like Chrome and Firefox on both desktop and tablet devices.

**Availability:** The system must be highly available and operational during standard university working hours.

# 3. Programing languages and tools

## 3.1 Frontend technologies

**Programming Language:** **TypeScript**

* We are using TypeScript, a statically typed superset of JavaScript. It was chosen to improve code quality, enable better tooling and autocompletion in the IDE, and catch potential errors during development rather than at runtime.

**Libraries & Tools:**

* **Angular:** The frontend is built using the Angular framework, which provides a structured, component-based architecture for creating scalable and maintainable single-page applications.
* **HTML & CSS:** Standard HTML5 and CSS3 are used to structure the content and apply custom styling to the application.
* **Bootstrap:** We are using the Bootstrap library as our primary CSS framework. It provides a robust set of pre-built components and a responsive grid system, which allows us to rapidly develop a professional, consistent, and mobile-friendly user interface that adheres to the project's green and white theme.

## 3.2 backend technologies

**Database:** **PostgreSQL**

A powerful, open-source object-relational database system was selected for its reliability, data integrity features, and robust performance in handling the relational data of users, buildings, and inspections.

**Database Management Tool:** **pgAdmin**

We are using the pgAdmin graphical user interface (GUI) for all database interactions during development. This tool allows for efficient schema design, data management, and query execution.

## 3.3 Framework

**Frontend: Angular**

* Angular is the comprehensive framework used to build the entire user-facing side of the application. It manages everything from the user interface components and page navigation (routing) to handling user input and communicating with the backend API. Its component-based structure allows us to create reusable UI elements (like headers and forms) for a consistent user experience.

**Backend: Node.js and Express**

* **Node.js** serves as the server-side JavaScript runtime environment, allowing us to build a fast and scalable backend.
* **Express.js** is the web application framework that runs on top of Node.js. We use it to create our RESTful API, defining the routes (e.g., /api/users/login, /api/buildings), handling HTTP requests from the Angular frontend, and implementing all the business logic for the application.

# 4. Database implementation

## 4.1 Entity Relationship Diagram

A screenshot of a computer

AI-generated content may be incorrect.

## 4.2 Database design

|  |  |  |  |
| --- | --- | --- | --- |
| **Table Name** | **Column Name** | **Data Type** | **Constraints / Notes** |
| **User** | id (PK) | String | Primary Key |
| email | String | Unique, Required |
| password | String | Required, Hashed |
| firstName | String | Required |
| lastName | String | Required |
| role | Role (Enum) | ADMIN or TECHNICIAN |
| **Building** | id (PK) | String | Primary Key |
| name | String | Unique, Required |
| location | String | Optional |
| **ChecklistItem** | id (PK) | String | Primary Key |
| question | String | Required |
| type | QuestionType | BOOLEAN, NUMERIC, TEXT |
| buildingId | String | Foreign Key to Building, Required |
| **Inspection** | id (PK) | String | Primary Key |
| buildingId | String | Foreign Key to Building, Required |
| userId | String | Foreign Key to User, Required |
| submittedAt | DateTime | Auto-generated |
| **InspectionAnswer** | id (PK) | String | Primary Key |
| booleanAnswer | Boolean | Optional |
| numericAnswer | Float | Optional |
| textAnswer | String | Optional |
| inspectionId | String | Foreign Key to Inspection, Required |
| checklistItemId | String | Foreign Key to ChecklistItem, Required |
| **Issue** | id (PK) | String | Primary Key |
| comment | String | Required |
| resolved | Boolean | Default: false |
| inspectionAnswerId | String | Foreign Key to InspectionAnswer (One-to-One) |

# 5. Testing

## 5.1 Unit Testing

**System URL (Local Development):** <http://localhost:4200>

***Note****:* This URL is for local development only and requires the complete project environment (backend, frontend, and database) to be set up and running on the user's machine.

**TC01:** User Sign In Page:

A screenshot of a computer

AI-generated content may be incorrect.

**TC02:** User Dashboard for Admin:

A screenshot of a computer

AI-generated content may be incorrect.

Test Data:

[S576433@nwmissouri.edu](mailto:S576433@nwmissouri.edu)

Pass: 1234

A screenshot of a computer

AI-generated content may be incorrect.

Status Note: The back-end functionality for an administrator to create new technician accounts has been fully implemented and tested. The front-end user interface for the technician-specific dashboard, which they are redirected to after logging in, is under development.

**TC03:** Add a user functionality – Admin can create user and assign role as admin/ technician:

A screenshot of a computer

AI-generated content may be incorrect.

Provided test data:

A screenshot of a computer

AI-generated content may be incorrect.

User account created successfully:

A screenshot of a computer

AI-generated content may be incorrect.

Note: User profile management frontend part is under development:

A screenshot of a computer

AI-generated content may be incorrect.

**TC04:** Admin creating new building flow:

A screenshot of a computer

AI-generated content may be incorrect.

When user clicked “Go to Building Management”:

A screenshot of a computer

AI-generated content may be incorrect.

When user clicked “+Add new Building”

A screenshot of a computer

AI-generated content may be incorrect.

Admin fills out the form using below test data:

A screenshot of a computer

AI-generated content may be incorrect.

New building is added to the building list:

A screenshot of a computer

AI-generated content may be incorrect.

**TC05:** Modifying existing building feature:

A screenshot of a computer

AI-generated content may be incorrect.

**TC06:** When admin clicks “delete” button:

A screenshot of a computer

AI-generated content may be incorrect.

Note: Delete warning pop-up development is in progress:

When admin selects ok:

A green and white stripe

AI-generated content may be incorrect.

## 5.2 System Testing

**End-to-End Scenario: Admin Registers a New User**

1. **Frontend (Angular):** An Admin fills out the "Add User" form in the /admin portal. The UserService sends a POST request with the new user's data and the Admin's JWT to the backend.
2. **Backend (Node.js):** The API validates the Admin's JWT using the protect and admin middleware.
3. **Database (PostgreSQL):** Upon validation, the backend hashes a default password and uses Prisma to create a new technician record in the User table.
4. **UI Update:** The backend sends a success response, and the Angular UI displays a success message and clears the form.

**End-to-End Scenario: User Login**

1. **Frontend (Angular):** A user enters their email and password on the /login page. The AuthService sends a POST request with the credentials to the backend.
2. **Backend (Node.js):** The public login endpoint receives the request.
3. **Database (PostgreSQL):** The backend finds the user by email and uses bcrypt to securely compare the provided password with the hashed password stored in the database.
4. **UI Update:**
   * **On Success:** The backend returns a JWT. The Angular AuthService saves the token and redirects the user to their dashboard.
   * **On Failure:** The backend returns a 401 Unauthorized error, and the Angular UI displays a "Login failed" message.

**End-to-End Scenario: Admin Creates a New Building**

1. **Frontend (Angular):** An admin fills out the "Add Building" form and submits it. The Angular service sends a POST request with a nested JSON payload and the user's JWT to the backend.
2. **Backend (Node.js/Express):** The API receives the request. The authentication middleware first validates the JWT to confirm the user is an authenticated admin.
3. **Database (PostgreSQL):** The backend service then initiates a Prisma transaction, creating the new building and its associated checklist items in the database.
4. **UI Update:** The backend sends a success response, and the Angular UI navigates the admin back to the building list, which automatically refreshes to show the newly created building.