Project Title: Smart City Management

### Architecture: Mern stack

### Technologies: React.js (Frontend), Express.js (Backend), xampp (Database)

### API Support: REST API

**📌 1. Problem Definition**

"Urban environments face increasing challenges in managing public services such as traffic, disaster response, utilities, waste, and law enforcement. The Smart City Management System aims to centralize and digitize the monitoring, reporting, and optimization of these services using modern web technologies, providing both citizens and administrators with real-time data, efficient workflows, and actionable insights."

**📌 2. Software Requirements Specification (SRS)**

## 1. **Introduction**

### 1.1 Purpose

The purpose of this system is to provide a centralized digital platform that empowers city officials to manage urban services efficiently, while allowing the public to view service-related data. It integrates real-time visual data like charts and maps to offer transparency and assist in decision-making.

### 1.2 Scope

This full-stack web-based system offers:

* **Modules**: Disaster Management, Traffic Monitoring, Utilities, Waste Management, Transport Tracking, Crime Reporting, Complaints, Billing.
* **Role-based Access**:
  + Officials: Can perform full CRUD operations.
  + Public: Can only view real-time data.
* **Frontend**: React.js with Pie/Bar Charts and Google Maps.
* **Backend**: Node.js, Express.js, Sequelize ORM, JWT-based authentication.
* **Database**: MySQL

### 1.3 Intended Audience

* Government/City Officials
* Public Citizens
* Developers & Testers

### 1.4 Assumptions & Dependencies

* JWT secret key is configured via .env.
* MySQL and Node.js are installed.
* Map APIs (e.g., Google Maps) have valid API keys.
* The app is deployed on a secure HTTP/HTTPS server.

## **2. Overall Description**

### 2.1 Product Perspective

The system integrates multiple city management services into a unified dashboard. Each module operates as a microservice and can be independently scaled or extended.

### 2.2 User Classes and Characteristics

| **Role** | **Description** |
| --- | --- |
| Public | Can view disaster reports, traffic, utility status, etc. via charts and maps. |
| Official | Authorized personnel with full CRUD access to all service modules. |

### 2.3 Operating Environment

* Web browser with JavaScript support
* Backend: Node.js environment
* Database: MySQL
* External APIs: Google Maps, Chart.js/Recharts

## **3. System Features (Functional Requirements)**

### ✅ Authentication

* Secure login using JWT
* Role verification using middleware
* Session validation on every API call

### ✅ Disaster Management

* Public: View disasters by area on map and chart
* Official: Add/update/delete disasters

### ✅ Traffic Monitoring

* Public: View traffic status per area
* Official: Add/update traffic data, suggestions

### ✅ Utilities Management

* Public: View electricity and water data with graphs
* Official: Submit maintenance data, track complaints

### ✅ Waste Management

* Public: View bin status
* Official: Update bin collection status

### ✅ Transport Tracking

* Public: View routes, real-time location on maps
* Official: Modify schedules, assign vehicles

### ✅ Crime Reporting

* Public: Submit crime reports
* Official: View/manage reports, analyze crime types

### ✅ Complaint Management

* Public: Raise city service complaints
* Official: Track, assign, and resolve complaints

### ✅ Billing

* Official: View/generate utility bills (admin only)

## **4. Non-Functional Requirements**

### 4.1 Performance

* Support 50+ concurrent users
* REST APIs return responses in < 1 second

### 4.2 Security

* JWT-based access control
* Role verification middleware:
* Password hashing via bcrypt

const isOfficial = (req, res, next) => {

if (req.role !== "official") return res.status(403).json({ message: "Access denied" });

next();

};

### 4.3 Usability

* Mobile-responsive frontend
* Dashboard with Google Maps, Bar & Pie Charts
* Form validations and error prompts

### 4.4 Scalability

* New modules can be plugged in (e.g., parking, environment)
* Charts and map overlays are modular

## **5. External Interfaces**

### 5.1 User Interface

* Built using React.js
* Role-based dashboards
* Charts: Pie, Bar (Recharts/Chart.js)
* Maps: Google Maps (with color-coded markers)

### 5.2 Hardware Interfaces

* Optional GPS-based inputs for transport tracking

### 5.3 Software Interfaces

* Google Maps API
* Chart.js or Recharts for data visualization
* RESTful API (backend)
* JWT and bcrypt for auth

### 5.4 Communication Interfaces

* HTTPS for all communications
* JSON format for data transfer

## **👥 System Users and Roles**

The system defines two primary user roles, each with distinct permissions and access levels to ensure secure, appropriate use of the platform.

| **Role** |  |  |  |  |  | **Description** |  |  | **Permissions** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Official** |  |  |  |  |  | Authorized government or municipal personnel responsible for managing city services. |  |  | - Full CRUD operations on all modules (Disaster, Traffic, Utilities, Transport, Crime, Waste, Complaints, Billing) - Access to admin dashboards and analytics - Role-based login via JWT |
| **Public User** |  |  |  |  |  | Regular citizens accessing city data or submitting reports. |  |  | - View-only access to public datasets (e.g., traffic, disasters, utilities) - Can raise complaints or crime reports - No access to add/update/delete operations |

### 🔐 Role Enforcement

Role-based access control (RBAC) is enforced using JWT and middleware. For

example:

const isOfficial = (req, res, next) => {

if (req.role !== "official") return res.status(403).json({ message: "Access denied" });

next();

};

### 🧭 Navigation Based on Role

| **Page/Module** |  | **Public Access** | **Official Access** |
| --- | --- | --- | --- |
| Home |  | ✅ | ✅ |
| Login/Register |  | ✅ | ✅ |
| Disaster Management |  | 🔍 View only | ✅ Full CRUD |
| Traffic Monitoring |  | 🔍 View only | ✅ Full CRUD |
| Utilities & Billing |  | 🔍 View only | ✅ Full CRUD |
| Waste Management |  | 🔍 View only | ✅ Full CRUD |
| Transport Info |  | 🔍 View only | ✅ Full CRUD |
| Crime Reports |  | 📩 Submit | ✅ Full Access |
| Complaints |  | 📩 Submit | ✅ Track/Resolve |

**🔗 Folder structure(backend)**

├── 📁 models

│ ├── bill.model.js

│ ├── complaint.model.js

│ ├── crime.model.js

│ ├── disaster.model.js

│ ├── index.js

│ ├── traffic.model.js

│ ├── transport.model.js

│ ├── user.model.js

│ ├── utility.model.js

│ └── waste.model.js

├── 📁 repositories/

│ ├── bill.repository.js

│ ├── complaint.repository.js

│ ├── crime.repository.js

│ ├── disaster.repository.js

│ ├── traffic.repository.js

│ ├── transport.repository.js

│ ├── user.repository.js

│ ├── utility.repository.js

│ └── waste.repository.js

backend/

├── app.js

├── .env # (JWT\_SECRET\_KEY, DB credentials)

├── 📁 config/

│ ├── db.config.js

│ └── sequelize.js

│

├── 📁 controllers/

│ ├── bill.controller.js

│ ├── complaint.controller.js

│ ├── crime.controller.js

│ ├── disaster.controller.js

│ ├── traffic.controller.js

│ ├── transport.controller.js

│ ├── user.controller.js

│ ├── utility.controller.js

│ └── waste.controller.js

│

├── 📁 middleware/

│ └── auth.middleware.js

├── 📁 services/

│ ├── bill.service.js

│ ├── complaint.service.js

│ ├── crime.service.js

│ ├── disaster.service.js

│ ├── traffic.service.js

│ ├── transport.service.js

│ ├── user.service.js

│ ├── utility.service.js

│ └── waste.service.js

├── 📁 routes/

│ ├── bill.routes.js

│ ├── complaint.routes.js

│ ├── crime.routes.js

│ ├── disaster.routes.js

│ ├── traffic.routes.js

│ ├── transport.routes.js

│ ├── user.routes.js

│ ├── utility.routes.js

│ └── waste.routes.js

## **🗂️ Garbage Management Project Structure**

### 🧠 ****domain/**** – Core Domain Models

Contains the business entities representing core data structures, independent of frameworks or databases.

* **admin.model.js** – Admin entity schema (e.g., roles, permissions)
* **bin.model.js** – Garbage bin model (e.g., location, status, capacity)
* **complaint.model.js** – Complaints raised by users (e.g., issues with waste collection, bin overflow)
* **driver.model.js** – Drivers responsible for waste collection (e.g., personal information, assignments)
* **location.model.js** – Location data for bins and works (e.g., coordinates, bin placement)
* **user.model.js** – Public user information (e.g., name, contact info, address)
* **work.model.js** – Work assignments/records for drivers (e.g., assigned routes, collection status)

### 🏗️ ****infrastructure/**** – System Infrastructure Layer

Houses database configurations and data access logic.

#### 🔧 **config/**

Contains configuration files related to external services and databases.

* **db.config.js** – Database connection settings (e.g., MongoDB connection or Sequelize for relational DB)

#### 🗃️ **repositories/**

Implements the actual DB operations for each entity. This abstracts the logic of interacting with the database.

* **admin.repository.js** – CRUD operations for the **Admin** model
* **bin.repository.js** – CRUD operations for the **Bin** model
* **complaint.repository.js** – CRUD operations for the **Complaint** model
* **driver.repository.js** – CRUD operations for the **Driver** model
* **location.repository.js** – CRUD operations for the **Location** model
* **user.repository.js** – CRUD operations for the **User** model
* **work.repository.js** – CRUD operations for the **Work** model

### 🧩 ****usecase/**** – Application Logic Layer

Encapsulates the business rules and use cases for the system. This layer coordinates domain logic and repository calls.

* **admin.usecase.js** – Business logic for admin functionalities (e.g., managing users, roles, etc.)
* **bin.usecase.js** – Business logic for bin-related tasks (e.g., tracking collection, assigning bins)
* **complaint.usecase.js** – Business logic for handling complaints (e.g., managing user complaints)
* **driver.usecase.js** – Business logic for driver work assignments (e.g., scheduling, work tracking)
* **user.usecase.js** – Business logic for user-related operations (e.g., submitting complaints, checking bin status)
* **work.usecase.js** – Business logic for work assignments and tracking

### 🌐 ****interface/**** – Presentation/API Layer

Defines the external interface (HTTP) for the application.

#### 📋 **controllers/**

Contain request handlers that call use cases and return responses.

* **admin.controller.js** – Controller for admin actions (e.g., user management)
* **complaint.controller.js** – Controller for handling complaints
* **driver.controller.js** – Controller for managing driver operations
* **bin.controller.js** – Controller for managing garbage bins
* **work.controller.js** – Controller for work assignment and status updates
* **user.controller.js** – Controller for public user actions (e.g., submitting complaints)

#### 🚦 **routes/**

Defines REST endpoints and maps them to controller actions.

* **admin.routes.js** – API routes related to admin (e.g., user management)
* **complaint.routes.js** – API routes for complaints (e.g., submitting/viewing complaints)
* **driver.routes.js** – API routes for driver operations (e.g., assigning work, viewing work status)
* **bin.routes.js** – API routes for garbage bins (e.g., viewing status, updating bins)
* **user.routes.js** – API routes for users (e.g., viewing bins, submitting complaints)

### 🛠️ ****Root-Level Files****

The root folder holds key project files and configurations.

* **.env** – Environment variables (e.g., DB connection string, JWT secret key, port)
* **app.js** – Main Express application setup and middleware configuration
* **package.json** – Project dependencies, scripts (e.g., npm run start, npm run dev)
* **README.md** – Project documentation, setup guide, usage instructions

### 📁 ****Additional Considerations****

#### Middleware

* **auth.middleware.js** – Handles authentication (JWT) and role-based access (public vs. official)

#### Utilities

* **logger.js** – Logging utility (optional, for debugging and monitoring purposes)

**🔗 Folder structure(frontend)**

├───redux/

│ │ ├───actions/

│ │ │ ├───authActions.js

│ │ │ ├───disasterActions.js

│ │ │ └───trafficActions.js

│ │ ├───reducers/

│ │ │ ├───authReducer.js

│ │ │ ├───disasterReducer.js

│ │ │ ├───trafficReducer.js

│ │ │ └───utilityReducer.js

│ │ └───store.js

│ ├───styles/

│ │ ├───App.css

│ │ ├───HomePage.css

│ │ ├───Login.css

│ │ ├───AdminDashboard.css

│ │ └───Disaster.css

│ ├───App.js

│ ├───index.js

│ ├───reportWebVitals.js

│ └───setupTests.js

└───.env

└───package.json

└───README.md

└───webpack.config.js

└───.gitignore

└───src/

│ ├───assets/

│ │ ├───background.jpg

│ │ ├───earthquake.png

│ │ ├───icon1.png

│ │ ├───icon2.png

│ │ └───icon3.png

│ ├───components/

│ │ ├───DisasterDetails.js

│ │ ├───TrafficDataSearch.js

│ │ ├───TrafficMap.js

│ │ ├───Navbar.js

│ │ ├───WasteManagement.js

│ │ └───PrivateRoute.js

│ ├───pages/

│ │ ├───Home.js

│ │ ├───About.js

│ │ ├───AdminDashboard.js

│ │ ├───UserDashboard.js

│ │ ├───Disaster.js

│ │ ├───Traffic.js

│ │ ├───Waste.js

│ │ └───Login.js

## **🗂️ Frontend Folder Structure**

### 🌍 ****public/**** – Public Assets

Contains files that are accessible publicly, such as images, icons, and the main HTML file.

* **index.html** – The main HTML file that is served to the browser. Contains the root div where the React app will be rendered.
* **favicon.ico** – Icon for the application tab in the browser.
* **robots.txt** – Instructions for web crawlers about which pages to index or follow.
* **manifest.json** – Defines app metadata for PWA (Progressive Web App) functionality.

### 🧩 ****src/**** – Main Application Source Code

All the core components and logic for the frontend reside in this directory.

#### **app.js**

The main entry point for the React app, where the root App component is defined and routing is initialized.

#### **index.js**

The entry file that connects React to the DOM, rendering the App component and applying global contexts (e.g., Redux provider).

### 🧠 ****components/**** – Reusable UI Components

Contains all the small, reusable components for the application UI. These components are used throughout the app for consistent UI rendering.

* **Navbar.js** – The main navigation bar for the app with links to various pages like home, dashboard, etc.
* **DisasterDetails.js** – Displays detailed information about a disaster or emergency (e.g., risk, affected area).
* **DisasterForm.js** – A form to report disasters or submit related data.
* **TrafficMap.js** – A map component that visualizes traffic data or traffic status using Google Maps.
* **TrafficDataSearch.js** – A search bar for querying traffic data based on specific criteria (e.g., area, date).
* **WasteManagement.js** – A component to display and manage garbage bins, waste collection status, etc.
* **UtilityDashboard.js** – Displays utility consumption data and alerts.
* **PrivateRoute.js** – Protects routes by ensuring that only authenticated users can access certain pages (role-based).

### 📄 ****pages/**** – Page-level Components

Contains full page components that are typically linked to different routes. These pages use smaller components and encapsulate the main logic for each specific view.

* **Home.js** – The homepage that introduces the Smart City Management System.
* **About.js** – A page providing information about the platform and its features.
* **AdminDashboard.js** – A dashboard for the Admin user to manage the system, including viewing complaints, managing users, etc.
* **UserDashboard.js** – A dashboard for public users where they can view reports or file complaints.
* **Disaster.js** – A page displaying disaster-related information and relevant alerts.
* **Traffic.js** – Displays live traffic data and reports.
* **Waste.js** – Displays the status of garbage bins, collection schedules, etc.
* **Utilities.js** – Displays utility consumption data and reports for public users.

### 🛠️ ****redux/**** – Redux Store Configuration

Contains the Redux state management setup for handling app-wide states, such as authentication, user data, and disaster reports.

#### **actions/**

Contains the actions dispatched to the Redux store.

* **authActions.js** – Contains actions related to user authentication (e.g., login, logout).
* **disasterActions.js** – Actions for fetching and managing disaster data.
* **trafficActions.js** – Actions for managing traffic-related data.
* **utilityActions.js** – Actions for managing utility consumption data.
* **transportActions.js** – Actions for handling transport-related data.

#### **reducers/**

Contains reducers that manage how the application state changes in response to actions.

* **authReducer.js** – Handles state related to user authentication (e.g., logged-in user details).
* **disasterReducer.js** – Manages state for disaster-related data.
* **trafficReducer.js** – Manages traffic-related data in the app.
* **utilityReducer.js** – Manages utility data.
* **transportReducer.js** – Manages transport-related data.

#### **store.js**

Sets up the Redux store and combines all reducers for state management.

### 🎨 ****assets/**** – Static Assets

Contains images, icons, and other media assets used throughout the application.

* **background.jpg** – A background image used across the app.
* **disasterIcon.png** – Icon representing disaster data.
* **trafficIcon.png** – Icon used for traffic-related components.
* **wasteIcon.png** – Icon used for waste management-related components.

### 📋 ****styles/**** – Global Styles

Contains global or shared CSS files used across the project.

* **App.css** – The main CSS file to style the general layout and global components.
* **Disaster.css** – Specific CSS for disaster-related components and pages.
* **HomePage.css** – Styles for the homepage component.
* **Login.css** – Styles for the login page.
* **AdminDashboard.css** – Styles for the admin dashboard page.

### 🛠️ ****Root-Level Files****

* **.env** – Environment variables such as API keys, server URLs, and credentials for different environments (development, production).
* **package.json** – Project dependencies, scripts (e.g., start, build), and other configurations.
* **README.md** – Documentation explaining how to set up, run, and use the project.
* **webpack.config.js** – Configuration file for Webpack bundling.
* **.gitignore** – Specifies files and directories that should not be tracked by Git (e.g., node\_modules, .env).

### 🧩 ****Folder Structure Summary****

This structure is designed to clearly separate concerns:

* **components** hold small UI elements.
* **pages** handle full-page views for specific routes.
* **redux** manages state globally.
* **assets** hold all images and static files.
* **styles** provide global styling.

The **App.js** file serves as the main entry point, while **index.js** links everything together. The **.env** and **package.json** files ensure that environment variables and dependencies are correctly set up.

This structure is modular, clean, and easy to maintain as the project scales.

**🔁** **Clean Architecture Block Diagram (Text-Based)**

**Data Flow Diagram:**

**Database**

**(Sequelize)**

**Backend**

**(Express.js)**

**Frontend**

**(React + Redux)**

↓ ↓ ↓

**Models**

**(Sequelize)**

**(bill.model, complaint.model, disaster.model user.model, etc.)**

**Pages/Views**

**(Home, Admin, Disaster, Traffic,User)**

**Controllers**

**(bill.controller,**

**traffic.controller, etc.)**

↓ ↓ ↓

**Repositories**

**(bill.repository, complaint.repo, disaster.repo, etc.)|**

**Services**

**(bill.service.js, traffic.service, user.service, etc.)**

**Components**

**(Navbar, Disaster, Details, Traffic, | Data Search,etc;)**

↓ ↓ ↓

**Middleware**

**(auth.middleware, JWT verification, access control)**

**Routes**

**(bill.routes.js, traffic.routes, user.routes.js,etc;)**

**Assets**

**(background.jpg,**

**earthquake.png, icons)**

↓ ↓ ↓

**Logger**

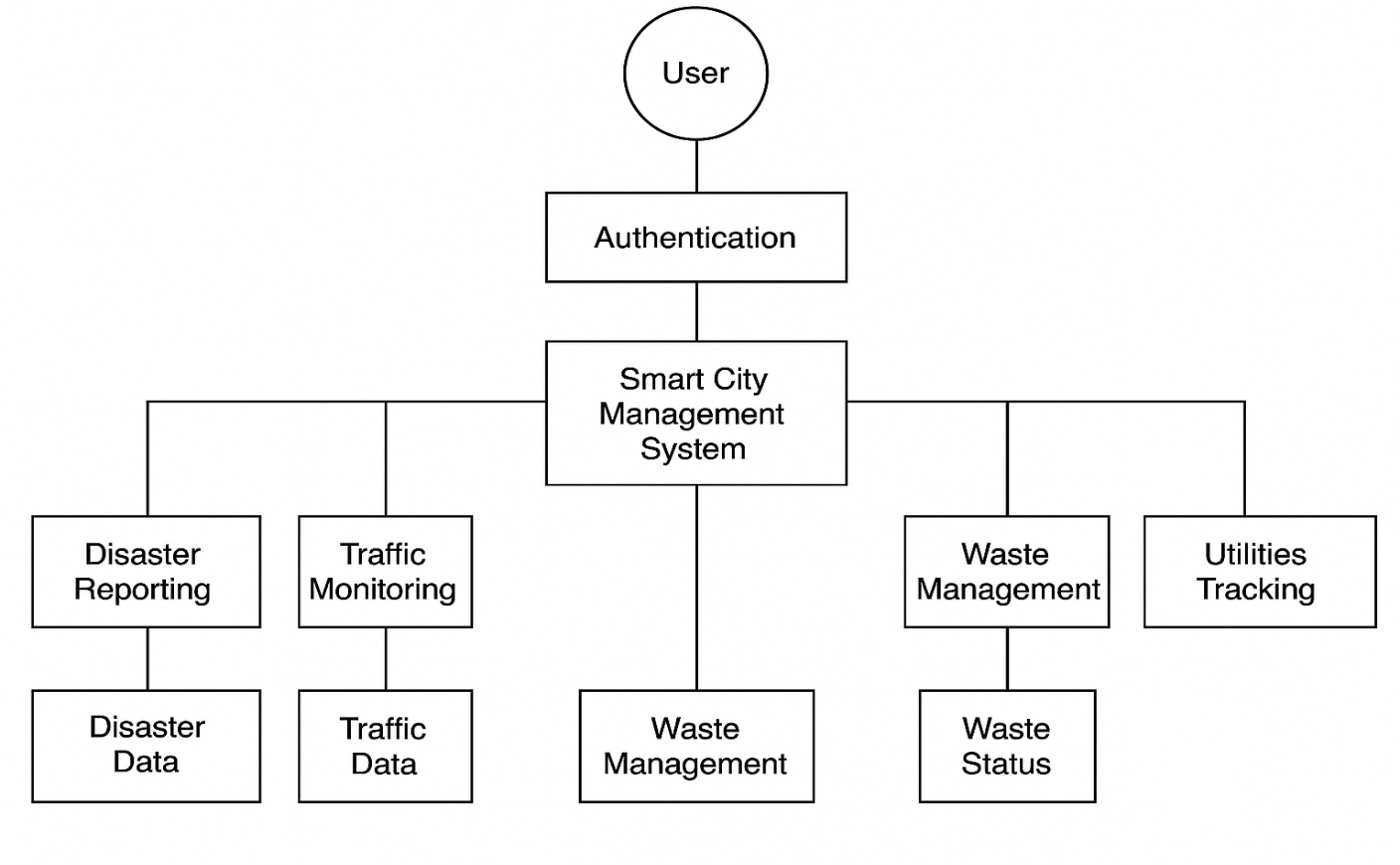
**(optional logging , for debugging, performance)**

**Utils (optional)**

**(helper functions, constants, etc.)**

**Styles**

**(App.css, Home, AdminDashboard,Disaster.css, etc.)**



**Purpose:**  
To show how data moves between the users (via the frontend), backend (API, services, routes), and database (Sequelize ORM with SQL DB) in the system.

**Components:**

**Actors:**

* **Public User:** Views disaster/traffic/waste data, submits complaints, logs in.
* **Official User (Admin):** Manages modules like traffic, waste, utility, disaster, assigns tasks, performs CRUD operations.
* **Driver:** Updates assigned tasks (e.g., waste collection status).

**Processes:**

* **User logs in →** authReducer and authActions send credentials → API (/api/users/login) verifies using JWT → Access token is returned and stored.
* **User submits traffic report →** TrafficDataSearch.js → Sends POST to /api/traffic → traffic.controller.js → traffic.service.js → traffic.repository.js → Data is stored via Sequelize in the database.
* **Admin updates waste bin info →** AdminDashboard.js → PUT request → Processed by backend controller → Updated in database.
* **Driver updates complaint status →** PUT API call → Updates complaint model → Changes reflected in dashboard view.
* **Data fetch →** GET API call → Returns filtered JSON → Renders in React components (TrafficMap.js, WasteManagement.js, etc.)

**Explanation:**

[User / Admin / Driver] <----> [Frontend React + Redux]

↓

[REST API Layer (Express.js + JWT Auth)]

↓

[Controllers → Services → Repositories]

↓

[Sequelize ORM (MySQL DB)]

* Each user action (login, submit complaint, update task) is initiated from the **React frontend**, passed through Redux actions and dispatched to backend API endpoints.
* The **Express.js backend** receives the request in the route file, processes it using controllers and services, and interacts with the **Sequelize ORM models** to update or fetch data from the **SQL database**.
* JWT-based middleware ensures that only users with the correct role (public/official) can access sensitive endpoints.
* Once data is processed, the backend sends a **JSON response**, which is then rendered dynamically in the frontend components using state or Redux store.
* This allows real-time, role-based interaction with the smart city system modules (e.g., traffic, waste, disaster) while maintaining data consistency across all layers.

**🔁 Data Flow Summary:**

* **Frontend**: React components trigger actions (e.g., submit complaint).
* **API Layer**: Sends the request to Express routes/controllers.
* **Use Case**: Handles business logic and validation.
* **Repository**: Accesses sql database using models.
* **Domain Models**: Define the schema and structure of the data.

**Garbage Management System — Architecture Diagram (Text-Based)**

+-------------------------+ +-----------------------------+

| Frontend | | Backend |

| (React + Redux) | | (Node.js + Express + DB) |

+-------------------------+ +-----------------------------+

| |

| HTTP Requests (Axios/Fetch) → API Gateway |

|---------------------------------------------------->|

| |

| +-------------------------+ |

| | Express App (app.js) | |

| +-------------------------+ |

| | |

| v |

| +-------------------------------+ |

| | Routes Layer | |

| | (e.g., /api/users, /api/...) | |

| +-------------------------------+ |

| | |

| v |

| +----------------------------------+ |

| | Controllers Layer | |

| | (Handles req/res & calls service)| |

| +----------------------------------+ |

| | |

| v |

| +----------------------------------+ |

| | Services Layer | |

| | (Business logic, validation) | |

| +----------------------------------+ |

| | |

| v |

| +----------------------------------+ |

| | Repositories Layer | |

| | (Database access abstraction) | |

| +----------------------------------+ |

| | |

| v |

| +----------------------------------+ |

| | Sequelize ORM | |

| | (Model definitions & DB bridge) | |

| +----------------------------------+ |

| | |

| v |

| +----------------------------------+ |

| | Relational Database | |

| | (e.g., MySQL/PostgreSQL) | |

| +----------------------------------+ |

| |

|<----------------------------------------------------|

| JSON Response / Status Codes |

| |

+-------------------------+ +-----------------------------+

| Pages | | Middleware |

| (Home, Login, Dash) | | (Auth, JWT, Role checks) |

+-------------------------+ +-----------------------------+

| Components (Navbar, | ^

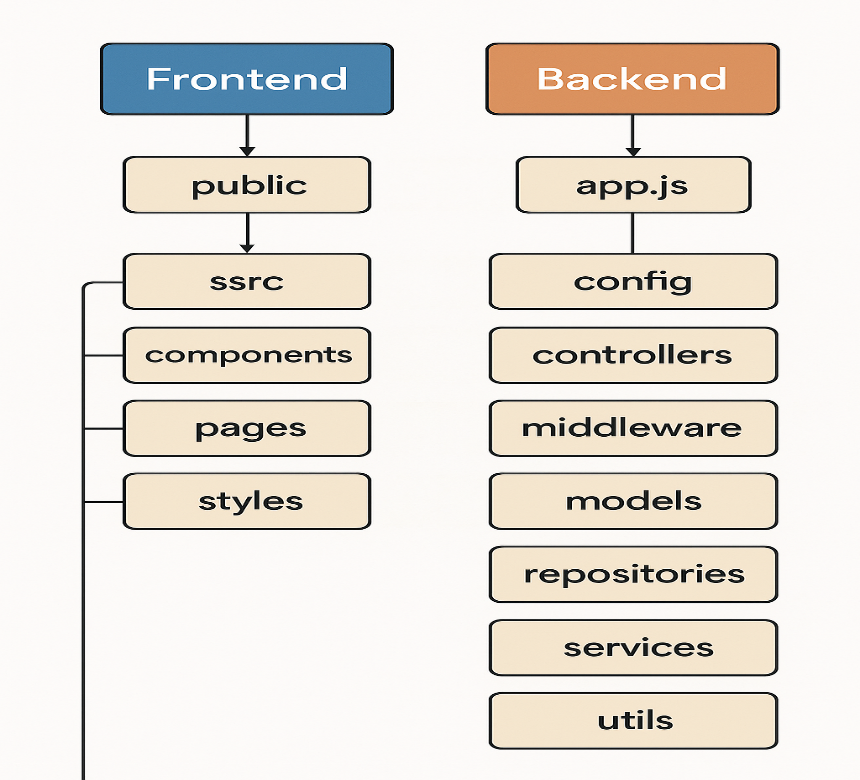
| Charts, Maps, etc.) | |

+-------------------------+ +-----------------------------+

| Assets | | Config / Utils |

| (Images, CSS) | | (env, logger, db setup) |

+-------------------------+ +-----------------------------+



🎯 Purpose:

To represent how the system is logically divided into maintainable and testable layers that separate concerns and minimize coupling.

──────────────────────────────

🔷 LAYERS & RESPONSIBILITIES

──────────────────────────────

[ Interface Layer ]

──────────────────────────────

• Handles HTTP (REST) and GraphQL requests.

• Includes:

- Express routes

- Controllers

- GraphQL resolvers

• Transforms incoming requests into a format understandable by the Use Case Layer.

• Applies authentication and validation logic.

↓

[ Use Case Layer ]

──────────────────────────────

• Contains application-specific business logic.

• Implements use cases like:

- File complaint

- Assign worker

- Track bin status

• Calls methods on interfaces (e.g., IComplaintRepository).

• Agnostic of frameworks and persistence mechanisms.

↓

[ Domain Layer ]

──────────────────────────────

• Contains core business entities and logic.

• Pure JavaScript/TypeScript classes or interfaces:

- User, Complaint, Bin, etc.

• Has no dependencies on infrastructure or frameworks.

• Models the business rules and constraints.

↓

[ Infrastructure Layer ]

──────────────────────────────

• Implements repository interfaces.

• Contains:

- Mongoose models (MongoDB schemas)

- Repositories (MongoDB logic)

- API middleware (auth, logging, rate limiting)

• Interfaces with:

- MongoDB

- Third-party APIs

- GraphQL engines

──────────────────────────────

🔁 Dependency Rule:

All dependencies \*\*point inward\*\*—the innermost layers (Domain, Use Case) know nothing about the outer layers (Infrastructure, Interface).

──────────────────────────────

📌 FLOW SUMMARY

──────────────────────────────

1. 🖥️ Frontend (React) sends a REST or GraphQL request.

2. 🌐 Interface Layer (routes/controller/resolvers) receives the request and validates it.

3. 🎯 Use Case Layer executes application logic.

4. 🧠 Domain Models are used to model data/state.

5. 🗄️ Infrastructure Layer persists/fetches data using Mongoose + MongoDB.

6. 🔁 Response flows back through the same layers to the frontend.

──────────────────────────────

✅ Benefits:

──────────────────────────────

✔ Testability – Each layer can be unit tested in isolation.

✔ Maintainability – Logic is clearly separated.

✔ Flexibility – Can swap databases or UI frameworks with minimal impact.

✔ Scalability – Easily extend use cases and plug in services.