**Logic Tier**

**Login**

1.1 **Input Validation**:

* Validate the format of the entered username and password (e.g. ensure no SQL injection, correct length, allowed characters, etc.)

1.2 **Authentication Logic**:

* Match the entered credentials against the data retrieved from the database (hashed passwords are compared using secure methods like bcrypt).

1.3 **Authorization Token Generation**:

* If the credentials are valid:
  + Generate a JSON Web Token (JWT) or session token with a unique identifier for the user and an expiration timestamp.
  + Embed user roles and permissions in the token payload for subsequent authorization checks.
  + Monitor token expiration and handle token refresh requests if needed

1.4 **Error Handling**:

* If credentials are invalid:
  + Log the failed attempt with a timestamp and IP address.
  + Return an error response indicating invalid credentials, without revealing which part (username or password) failed.

**Logout**

1.5 **Input Validation**:

* Validate the presence of the authorization token in the request headers or body.

1.6 **Token Verification**:

* Decode and validate the provided token (e.g., using libraries like JWT in the backend).
* Check the token's integrity (ensure it hasn't been tampered with) and expiration status.

1.6 **Session/Token Invalidation**:

* **JWT**: Mark the token as invalid by adding it to a blacklist until expiration

1.7 **Response Handling**:

* If the token is valid and successfully invalidated, return a success response indicating the logout is complete.
* If the token is missing, invalid, or already expired, return an appropriate error message.

1.8 **Error Handling**:

* Log logout errors, such as attempts to logout with an invalid token, for monitoring and debugging purposes.

**System Health Check**

1.9 **Database Connection Check**:

* If login was successful attempt to establish a connection to the database.
* If the connection fails:
  + Log the error for debugging (e.g., timestamp, the IP of the requester).
  + Return an error response with the message: "Connection failed."
* Do the same process if user presses the Run Health Check button.

1.10 **Data Fetching**:

* If the connection is successful:
  + Query the database to retrieve toll company-specific data, including:
    - Number of toll stations managed by the company.
    - Number of active vehicle tags registered under the company.
    - Total number of vehicle passes recorded for the company in the database.

1.11 **Response Handling**:

* Format the fetched data into a JSON response, for example:

{

"status": "success",

"toll\_company": {

"name": "Example Toll Co.",

"toll\_stations": 25,

"tags": 12500,

"passes": 450000

}

}

**Reset Toll Stations**

1.12 **Input Validation**:

* Ensure the uploaded file is provided and in CSV format.
* Check for the correct file name (tollstations2024.csv) or prompt the user to upload the correct file.
* Validate the file content to confirm all required columns are present:
  + OpID, Operator Name, Toll Station ID, Toll Station Name, Position Marker, Locality, Road, Lat, Long, Email, Price1, Price2, Price3, Price4.

1.13 **File Parsing**:

* Read the uploaded CSV file using a library like pandas (Python).
* Parse each row and convert it into a structured format (JSON).

**Data Tier**

2.1 **Database Schema for Users**:

* A **Users** table stores:
  + user\_id: Primary key, unique identifier.
  + username: Unique username.
  + hashed\_password: Securely hashed password.
  + role: User roles (e.g., admin, operator).
  + created\_at and updated\_at: Timestamps for record tracking.

2.2 **Database Schema for Token/Session Management**:

JWT:

* + A **TokenBlacklist** table may store:
    - token\_id: Identifier of the JWT token (e.g., hash of the token).
    - user\_id: Identifier of the user.
    - added\_at: Timestamp when the token was blacklisted.
  + On logout:
    - Insert the token ID into the blacklist.

2.3 **Database Schema:**

**TollCompanies** table:

* + company\_id: Unique identifier.
  + name: Toll company name.
  + created\_at, updated\_at: Timestamps.

**TollStations** table:

* + op\_id: Operator ID (integer).
  + operator\_name: Name of the operator (string).
  + station\_id: Unique Toll Station ID (integer).
  + station\_name: Name of the toll station (string).
  + position\_marker: Position marker (string or integer).
  + locality: Locality of the toll station (string).
  + road: Road name or identifier (string).
  + latitude: Geographical latitude (decimal).
  + longitude: Geographical longitude (decimal).
  + email: Email address for station communication (string).
  + price1, price2, price3, price4: Pricing categories for tolls (decimal).

**VehicleTags** table:

* + tag\_id: Unique identifier.
  + company\_id: Foreign key linking the tag to a toll company.
  + vehicle\_info: Associated vehicle details.
  + status: Active or inactive.

**Passes** table:

* + pass\_id: Unique identifier.
  + station\_id: Foreign key linking the pass to a toll station.
  + tag\_id: Foreign key linking the pass to a vehicle tag
  + timestamp: Date and time of the vehicle pass.

**Operations:**

Count Toll Stations(), Count Vehicle Tags(), Count Passes()

**Delete Existing TollStation Data () :** Remove all current records from the TollStations table to avoid duplication.

**Insert new records** parsed from the CSV file into the TollStations table:

**Data Validation ()**: Validate data types for each field:

* + Ensure latitude and longitude are valid decimal values.
  + Confirm email is properly formatted.
  + Ensure price fields are non-negative decimals.

**Presentation Tier**

3.1 **Login Form**:

* Provide fields for username and password.
* Include client-side validation for required fields (e.g., ensure both fields are filled).

3.2 **Form Submission**:

* Send the credentials securely to the server using HTTPS.
* Use REST API to make a POST /login request with:

{"username": "example\_user",

"password": "example\_password" }

3.3 Login **Response Handling**:

* On success: Redirect the user to the application home page.
* On failure: Display an error message (e.g., "Invalid username or password") without providing specifics.

3.4 **Session Management**:

* Return to log in page when the token expires

3.5 **Logout Trigger**:

* Provide a logout option in the user interface (e.g., a "Logout" button in the navigation bar).

3.6 Logout **API Call**:

* On clicking the logout button, send a POST /logout or DELETE /session request to the server with the token included in the request headers for authentication: {"Authorization": "Bearer <user-token>"}

3.7 Logout **Response Handling**:

* If the server confirms the logout:
  + Redirect the user to the login page or show a confirmation message (e.g., "You have been logged out successfully").
* If the logout fails (e.g., invalid token):
  + Show an appropriate error message (e.g., "Failed to log out. Please try again.").

3.8 **User Interface Update**:

* Disable or hide features requiring authentication (e.g., user profile or dashboard links) once the token is removed.

3.9 **Health Check Trigger**:

* Provide a button ("Run Health Check").
* Example API call for health check:

{

"endpoint": "/health-check",

"method": "GET",

"headers": {

"Authorization": "Bearer <token>"

}

}

3.10 Health Check **API Response Handling**:

**On Success**: Display the results to the user:

Health Check Successful!

Company: Example Toll Co.

Toll Stations: 25

Tags: 12,500

Passes: 450,000

Else Display an error message: "Connection failed."

**ΕΝΔΕΙΚΤΙΚΗ ΕΚΤΕΛΕΣΗ**

### ****Login****

1. **User Interaction (Presentation Tier)**:
   1. User enters their username and password in the login form and submits it.
2. **Processing (Logic Tier)**:
   1. Validate the input.
   2. Retrieve the user record from the data tier.
   3. Verify the hashed password matches.
   4. Generate and return an authorization token if successful.
3. **Data Handling (Data Tier)**:
   1. Store or update token information if required (depending on the authentication mechanism used).
4. **Response (Presentation Tier)**:
   1. If login succeeds, the user is redirected to the application with the token used for future API calls.
   2. If login fails, display a meaningful error.

**Logout**

1. **User Interaction (Presentation Tier)**:
   1. The user clicks the "Logout" button in the UI.
   2. The frontend sends the logout request, including the token in the headers.
2. **Processing (Logic Tier)**:
   1. The server validates and decodes the token.
   2. The server invalidates the token by:
      1. Adding it to the blacklist (JWT).
3. **Data Handling (Data Tier)**:
   1. The data tier performs the necessary database operations (e.g., deleting the session record or blacklisting the token).
4. **Response (Presentation Tier)**:
   1. If successful:
      1. The frontend clears the token from local storage or cookies.
      2. The user is redirected to the login page.
   2. If unsuccessful:
      1. An error message is displayed.

**Health Check**

**User Interaction (Presentation Tier)**:

* 1. The user initiates a health check from the admin interface (Run Health Check Button)
  2. The request is sent to the server via a GET /health-check endpoint with the user’s token.

**Processing (Logic Tier):**

* 1. Validate the token and check user permissions.
  2. Attempt to connect to the database.
  3. If the connection is successful:
     1. Fetch the required data (number of stations, tags, passes).
     2. Format and send the response back to the user.
  4. If the connection fails:
     1. Log the error and send an error response to the user.

**Data Handling (Data Tier)**:

* 1. Execute the SQL queries to retrieve the requested data.
  2. Return the query results to the logic tier.

**Response (Presentation Tier)**:

* 1. Parse and display the results in the user interface.
  2. If successful, show the company details and database summary.
  3. If failed, display an error message.