Detailed Explanation Document

Below is a **step-by-step guide** on how to build a simple **User Management System** using **HTML, CSS, JavaScript** (Vanilla JS) for the frontend and **Spring Boot** (with Spring Security & JWT) for the backend. The project will store user data in **PostgreSQL** using **Spring Data JPA (Hibernate)**. This guide is intended to help you and your students understand the basic concepts required to build a secure, full-stack application.

---

1. Project Overview

You will create a **User Management System** with the following features:

- **User Registration and Login** using JWT-based authentication
- **CRUD** operations for **user profiles** (only after authentication)
  - **View** user profile (GET)
  - **Update** user profile (PUT)
  - **Delete** user profile (DELETE)
- **Frontend** uses **HTML, CSS, and JavaScript**:
  - **Login Page**
  - **Registration Page**
  - **Dashboard** (to view & edit user info)
- **Backend** built using **Spring Boot**:
  - **Spring Security** for authentication & authorization
  - **Spring Data JPA** with **PostgreSQL** for data persistence
  - **BCrypt** for password hashing
  - **JWT** for maintaining secure sessions

This guide helps you understand the full development cycle, from setting up your PostgreSQL database, configuring Spring Boot with security and JWT, to implementing the frontend and integrating the two via **AJAX** requests.

---

2. Technology Stack & Prerequisites

- **Java 17** (or compatible)
- **Spring Boot 2.x or 3.x** (This guide uses Spring Boot 3+ style)
- **Maven** (to manage dependencies)
- **PostgreSQL** (any version; e.g., PostgreSQL 14)
- **HTML/CSS/JavaScript** (Vanilla JS)
- **An IDE** for Java (e.g., IntelliJ IDEA, Eclipse, VSCode with Java extensions)
- **Postman** (optional but recommended for API testing)

---

3. Application Architecture

1. **Frontend (HTML/CSS/JS)**:
   - **index.html** (Login and Registration)

    - **dashboard.html** (Dashboard to view and edit user info)
    - Uses **Fetch API** to call the backend
    - Stores **JWT** in localStorage (or sessionStorage) after successful login
2. **Backend (Spring Boot)**:
    - **Controllers** handle HTTP requests (e.g., `/api/auth`, `/api/users`)
    - **Services** encapsulate business logic (e.g., user registration, updating user info)
    - **Repositories** (powered by Spring Data JPA) interact with the database
    - **Security** (Spring Security + JWT) handles authentication & authorization
    - **Database** is managed by PostgreSQL

Communication:

```
[Frontend] <-> [Spring Boot] <-> [Service Layer] <-> [Repository/DB]
```

---

4. Database Setup

1. **Install PostgreSQL** if not already installed.
2. **Create a new database**:

    ```
    CREATE DATABASE user_management_db;
    ```

3. **Create a PostgreSQL user** (role) with a password if needed, or use the default `postgres` user:

    ```
    CREATE USER my_user WITH PASSWORD 'my_password';
    ```

4. **Grant privileges** on `user_management_db` to `my_user`:

    ```
    GRANT ALL PRIVILEGES ON DATABASE user_management_db TO my_user;
    ```

---

5. Backend: Spring Boot

5.1 Project Structure <a name="project-structure"></a>

A typical Maven project structure:

```
user-management-backend
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com.example.usermanagement
│   │   │       ├── UserManagementApplication.java
│   │   │       ├── config
│   │   │       │   ├── SecurityConfig.java
│   │   │       │   └── JwtAuthenticationFilter.java
│   │   │       ├── controller
│   │   │       │   ├── AuthController.java
│   │   │       │   └── UserController.java
│   │   │       ├── dto
│   │   │       │   ├── LoginRequest.java
│   │   │       │   ├── RegisterRequest.java
│   │   │       │   ├── UserResponse.java
│   │   │       └── model
│   │   │
```

```
│   │   │           └── User.java
│   │   │       ├── repository
│   │   │       │   └── UserRepository.java
│   │   │       ├── service
│   │   │       │   └── UserService.java
│   │   │       └── util
│   │   │           └── JwtUtils.java
│   │   └── resources
│   │       └── application.properties
│   └── test
│       └── java (tests go here)
├── pom.xml
└── ...
```

5.2 Key Files Explained

1. **`UserManagementApplication.java`**
   The main class that starts the Spring Boot application (contains `main` method).
2. **`User.java` (model)**
   Defines the **User** entity for the database with properties like `id`, `username`, `email`, `password`.
3. **`UserRepository.java`**
   An interface extending `JpaRepository<User, Long>` for CRUD database operations on the `users` table.
4. **`UserService.java`**
   Contains business logic for user registration, login, profile retrieval, update, and deletion.
5. **`AuthController.java`**
   Handles `/api/auth/register` and `/api/auth/login` endpoints.
6. **`UserController.java`**
   Handles `/api/users/{id}` endpoints for **GET**, **PUT**, and **DELETE** user operations.
7. **`SecurityConfig.java`**
   Configures **Spring Security** (HTTP security, JWT authentication filter, password encoder, etc.).
8. **`JwtAuthenticationFilter.java`**
   A **filter** that intercepts each request to validate the JWT token from the `Authorization` header.
9. **`JwtUtils.java`**
   Utility class for generating and validating JWT tokens.
10. **`application.properties`**
    Contains database connection details and other configuration properties.

5.3 Security & JWT Integration

1. **Password Hashing**: Use `BCryptPasswordEncoder` to hash passwords before saving them to the database.
2. **JWT Generation**: Once the user is authenticated (via `/api/auth/login`), generate a JWT with user information.
3. **JWT Validation**: For every incoming request to protected endpoints (e.g., `/api/users/*`), the `JwtAuthenticationFilter` extracts the token from the `Authorization` header, validates it, and sets the authentication context if valid.
4. **Authorization**: Only authenticated users can access their own data via routes like `/api/users/{id}`.

5.4 API Endpoints

1. **`POST /api/auth/register`**
   - Body: `{ "username": "testUser", "email": "test@email", "password": "123456" }`
   - Registers a new user, hashes password with BCrypt, and stores the user in the DB.
   - Returns 201 (Created) if successful, or 400 if user already exists.
2. **`POST /api/auth/login`**
   - Body: `{ "username": "testUser", "password": "123456" }`
   - Authenticates a user. If valid, returns a JWT token in the response body (or as a header).
   - Returns 200 if successful, 401 if incorrect credentials.
3. **`GET /api/users/{id}`** (Requires JWT in `Authorization` header)
   - Retrieves user details (username, email).
   - Only the user with the matching `{id}` can retrieve their details.
4. **`PUT /api/users/{id}`** (Requires JWT in `Authorization` header)
   - Allows the user to update their details (username, email, or password).
   - Only the user with the matching `{id}` can update their info.
5. **`DELETE /api/users/{id}`** (Requires JWT in `Authorization` header)
   - Deletes a user's account from the database.
   - Only the user with the matching `{id}` can delete themselves.

5.5 Running the Spring Boot Application

1. **Configure `application.properties`** with your PostgreSQL credentials:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/user_management_db
spring.datasource.username=my_user
spring.datasource.password=my_password

spring.jpa.hibernate.ddl-auto=update

# JWT secret key (demo purposes only, use a secure key for production!)
jwt.secret=MyJwtSecretKeyForDemo
jwt.expiration=86400000
```

2. **Run** the application:
   - Using Maven: `mvn clean install` then `mvn spring-boot:run`
   - Using your IDE: run the `UserManagementApplication.java` main method
3. Once started, you should see something like:

```
Tomcat started on port(s): 8080 (http) ...
Started UserManagementApplication in ... seconds
```

   The backend is now running on **`http://localhost:8080`**.

5.6 Testing with Postman

1. **Register a new user**
   - Method: `POST`
   - URL: `http://localhost:8080/api/auth/register`

- Body (JSON):

```
{
  "username": "john123",
  "email": "john@example.com",
  "password": "password"
}
```

2. **Login**
    - Method: `POST`
    - URL: `http://localhost:8080/api/auth/login`
    - Body (JSON):

    ```
    {
      "username": "john123",
      "password": "password"
    }
    ```

    - Response will contain a JWT token (e.g. `Bearer <jwt-here>`).
3. **Get user details**
    - Method: `GET`
    - URL: `http://localhost:8080/api/users/1` (assuming user's ID is 1)
    - Headers: `Authorization: Bearer <jwt-token>`
    - Should return user details.
4. **Update user**
    - Method: `PUT`
    - URL: `http://localhost:8080/api/users/1`
    - Headers: `Authorization: Bearer <jwt-token>`
    - Body (JSON):

    ```
    {
      "username": "johnUpdated",
      "email": "johnupdated@example.com",
      "password": "newpassword"
    }
    ```

5. **Delete user**
    - Method: `DELETE`
    - URL: `http://localhost:8080/api/users/1`
    - Headers: `Authorization: Bearer <jwt-token>`

---

6. Frontend: HTML, CSS, JavaScript

6.1 Frontend Directory Structure <a name="frontend-directory-structure"></a>

```
user-management-frontend
├── index.html       (Login/Registration page)
├── dashboard.html   (Dashboard for profile management)
├── css
│   └── styles.css
└── js
    ├── auth.js      (Handles login/registration logic)
```

```
    └── dashboard.js (Handles user profile fetching, update, deletion)
```

6.2 Key Files Explained <a name="key-files-explained-1"></a>

1. **`index.html`**:
   - Contains the **Login Form** and **Registration Form** in separate sections or modals.
   - Calls **auth.js** functions to handle form submissions.
2. **`dashboard.html`**:
   - A protected page (only accessible if the user is logged in with a valid JWT).
   - Displays the user's info and allows them to update or delete their account.
   - Uses **dashboard.js** to handle these operations.
3. **`auth.js`**:
   - Contains code that listens for submit events on the login/registration forms.
   - Makes **`fetch()`** (AJAX) requests to the backend.
   - Stores JWT on successful login.
4. **`dashboard.js`**:
   - On page load, fetches the user info from the backend using the JWT.
   - Handles the update and delete operations.
5. **`styles.css`**:
   - Simple CSS to style your forms and dashboard.

6.3 AJAX (Fetch API) Calls

To communicate with the backend, your JavaScript code will:

1. **Send requests** to endpoints like `/api/auth/register` or `/api/users/:id`.
2. **Include JSON in the request body** where necessary.
3. **Include the JWT token** in `Authorization` header when calling protected endpoints.

Example of sending a login request from `auth.js`:

```
fetch('http://localhost:8080/api/auth/login', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    username: loginUsername.value,
    password: loginPassword.value
  })
})
.then(response => response.json())
.then(data => {
  if (data.token) {
    // Save token to localStorage
    localStorage.setItem('jwt', data.token);
    // Redirect to dashboard
    window.location.href = 'dashboard.html';
  } else {
    // Show error message
    alert(data.message || 'Login failed');
  }
})
.catch(err => console.error(err));
```

6.4 Running & Testing the Frontend

1. **Open `index.html`** in your browser (or serve the folder with a simple HTTP server).
2. **Register** a new user via the **registration form**.
3. **Login** with the same credentials.
4. If the login is successful, you'll be redirected to `dashboard.html`.
5. On `dashboard.html`, you should see your username/email. You can now **update** or **delete** your account.

Note: Make sure your **CORS** settings on the backend allow `http://localhost:...` from your frontend. You can configure CORS in Spring Security or by creating a `WebMvcConfigurer` bean.

---

7. Additional Features (Optional)

1. **Role-Based Access Control (RBAC)**: Implement roles like **ADMIN** and **USER**. Admins can view all users, while users can only manage their own accounts.
2. **Pagination**: If you want a list of all users, implement pagination to handle large datasets.
3. **Deployment with Docker**: Containerize both your Spring Boot application and the PostgreSQL database.
4. **Password Reset Feature**: Add an endpoint for password reset with an email-sending mechanism.

---

8. Best Practices & Security Tips

1. **Never store plaintext passwords**. Always hash with **BCrypt** or a similarly secure algorithm.
2. **Use HTTPS** in production to protect tokens and credentials in transit.
3. **Invalidate JWTs** if a user logs out or changes their password. This often requires token blacklisting or short-lived tokens.
4. **Store JWT** in an **HTTP-only cookie** or securely in localStorage. Be wary of **XSS** attacks if storing in localStorage.
5. **Always validate input** on both frontend and backend to prevent malicious data.