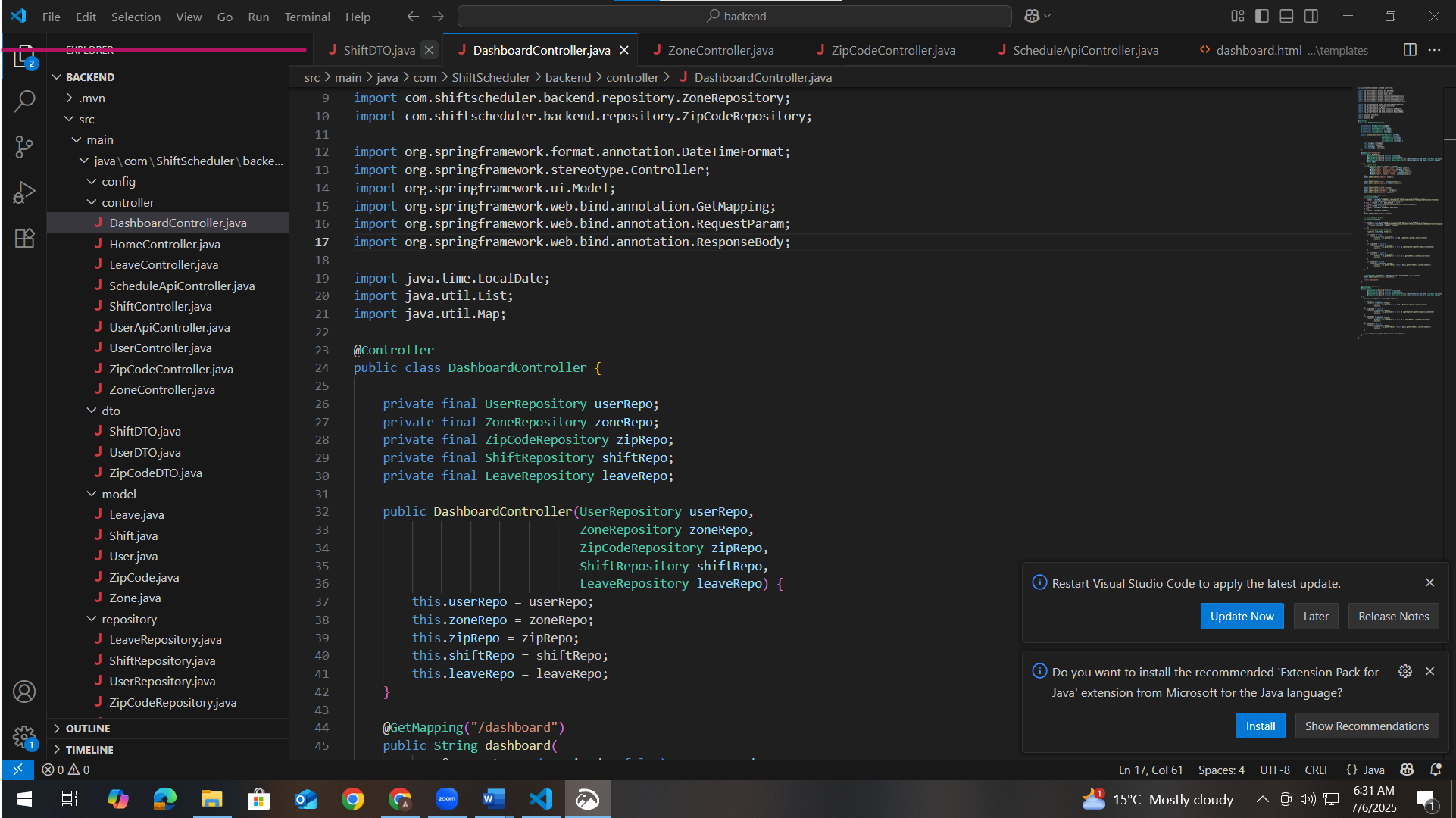
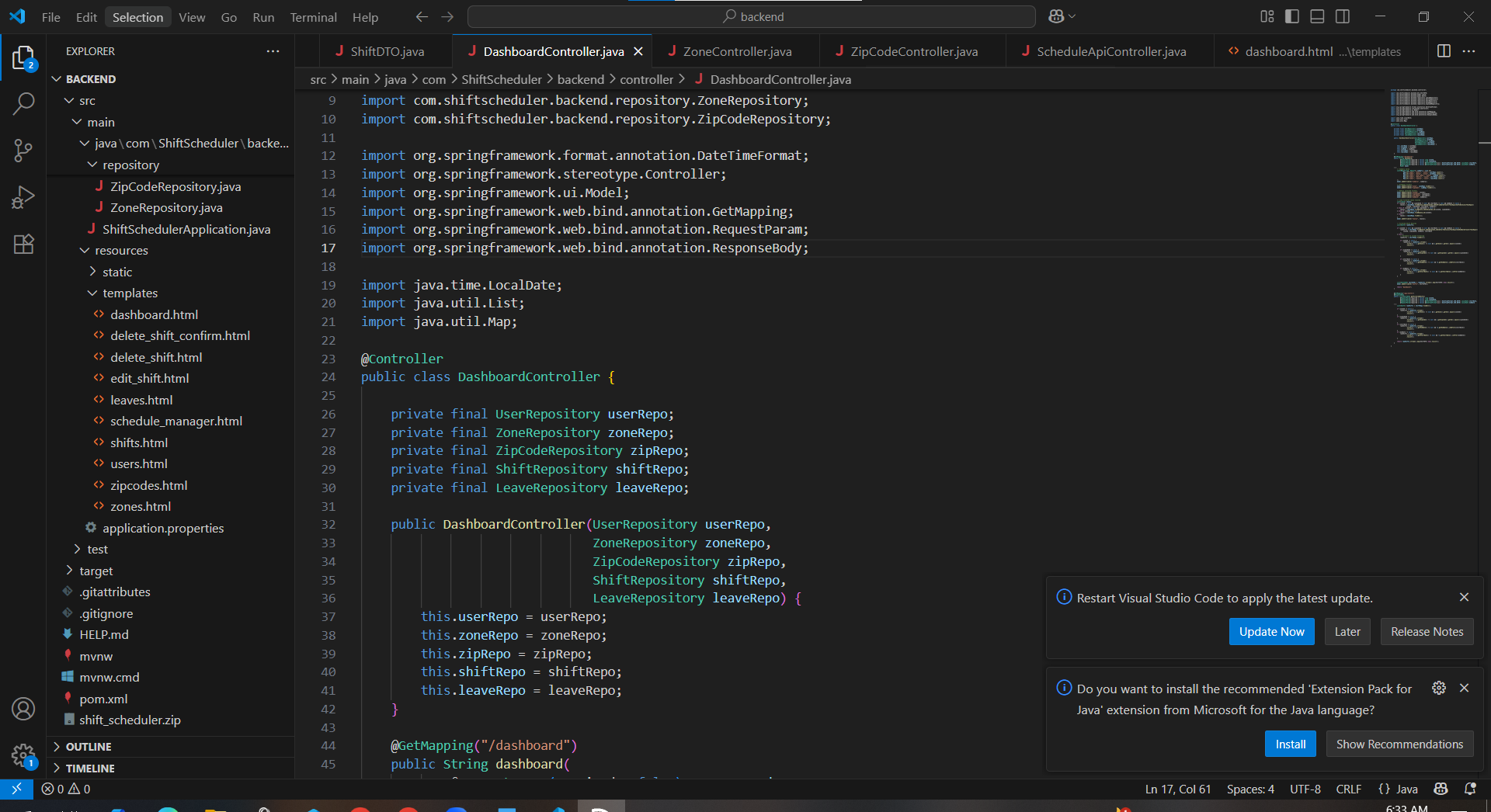
**Welcome and Project Structure Overview**

A walk through the complete code behind the Shift Scheduler application.

I used Java with Spring Boot for the backend, and Thymeleaf with Leaflet.js on the frontend. The folder is organized into key packages: controller, model, dto, repository, and the Thymeleaf templates.





**How to run the application**

**Prerequisites**  
Ensure the following are installed and configured:

* Java JDK 17 or later with the JAVA\_HOME environment variable set correctly.
* MySQL Server running on the default port (3306).
* MySQL Workbench for importing the database dump.
* Port 8080 is free (or modify the application.properties file to use a different port).
* An active internet connection for Maven to download dependencies.

**1: Set Up the Database in MySQL Workbench**  
Open MySQL Workbench and connect to the local MySQL server.

Create the database by running the following SQL command in a new query tab:

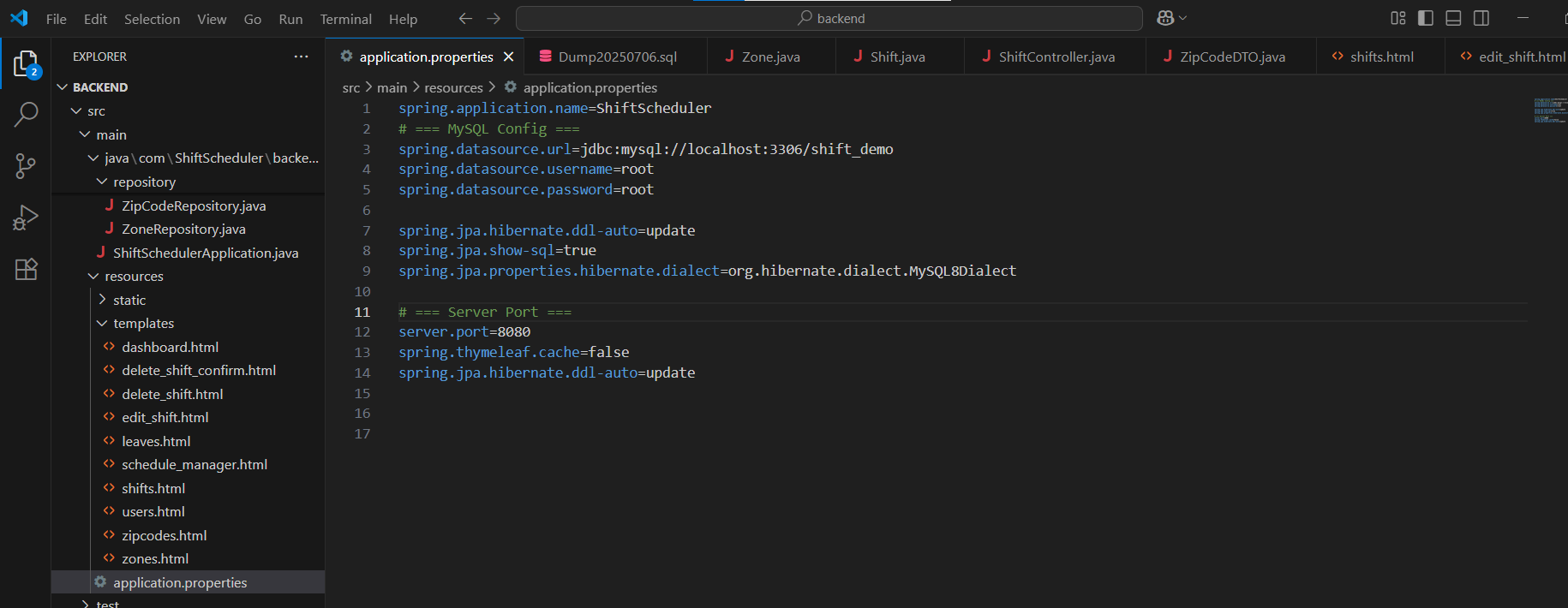
*CREATE DATABASE shift\_demo;*

Go to the top menu and choose **File > Open SQL Script**, then select the provided MySQL dump file.

Ensure shift\_demo is selected in the database dropdown.

Click **Execute** to run the script. This will create all necessary tables and insert data.

**2: Configure application.properties**  
Open the file:



The username and password should be set as the existing ones used in the local MySQL setup.

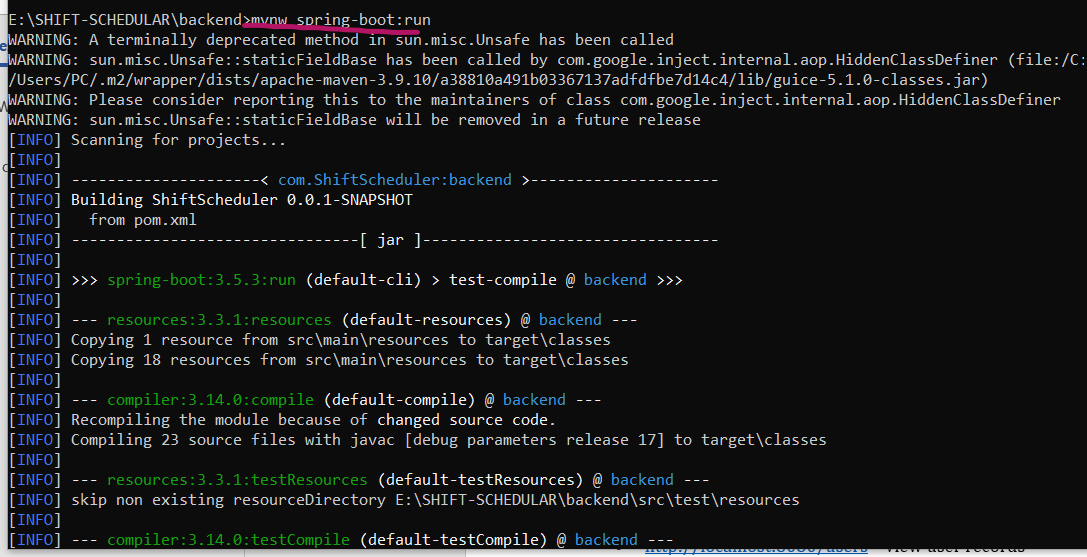
**3: Build and Run the Project**  
Open **Command Prompt**.

Navigate to the project’s backend directory where the mvnw wrapper file is located:

cd E:\SHIFT-SCHEDULAR\backend

Run the application using the Maven wrapper:

mvnw spring-boot:run

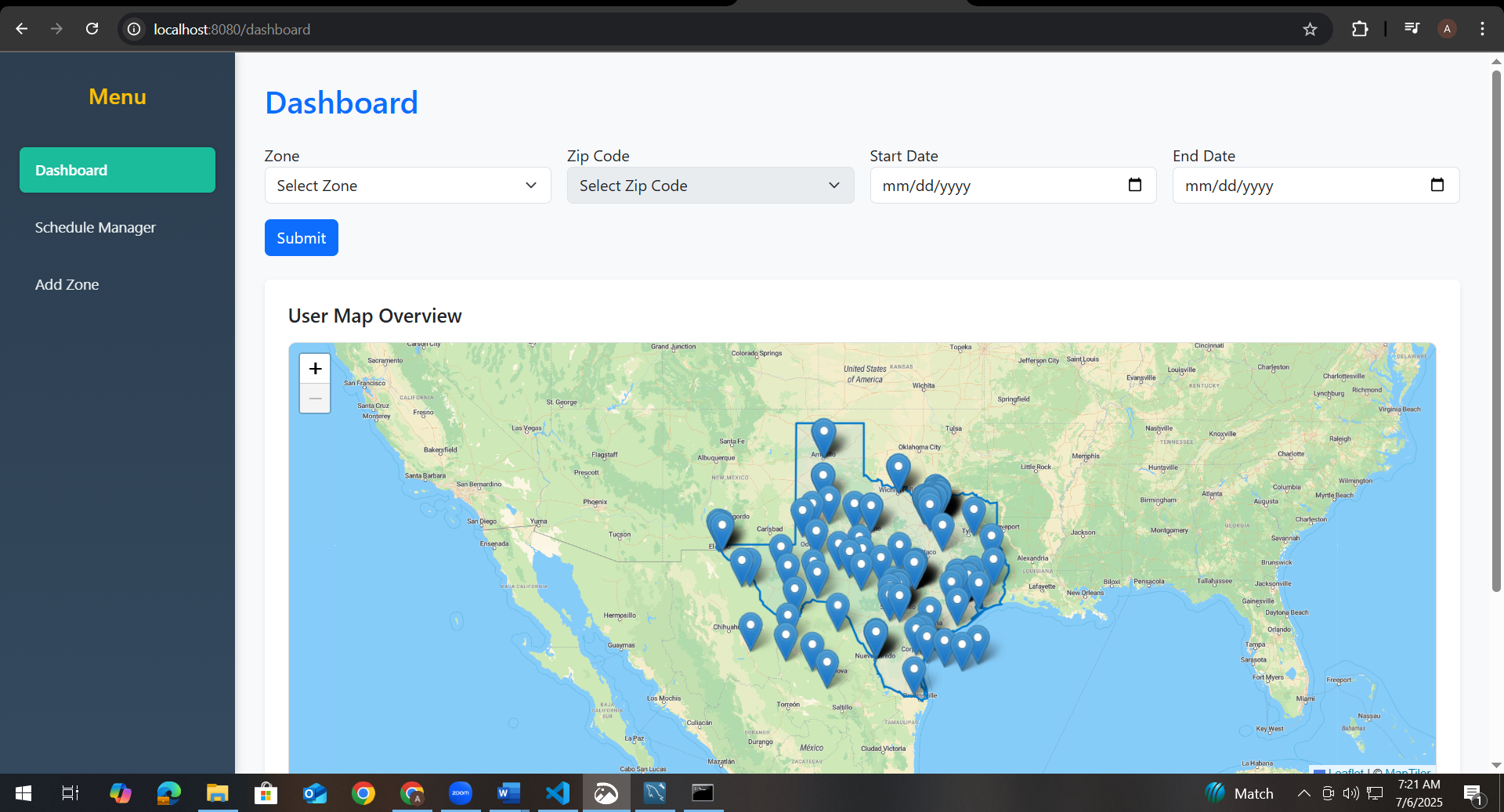


Maven will build the project, resolve dependencies, and start the embedded Tomcat server. Once the startup process completes successfully, a message will confirm that the application is running on port 8080.

**4: Access the Application via Web Browser**  
Open any modern web browser and go to:

bash

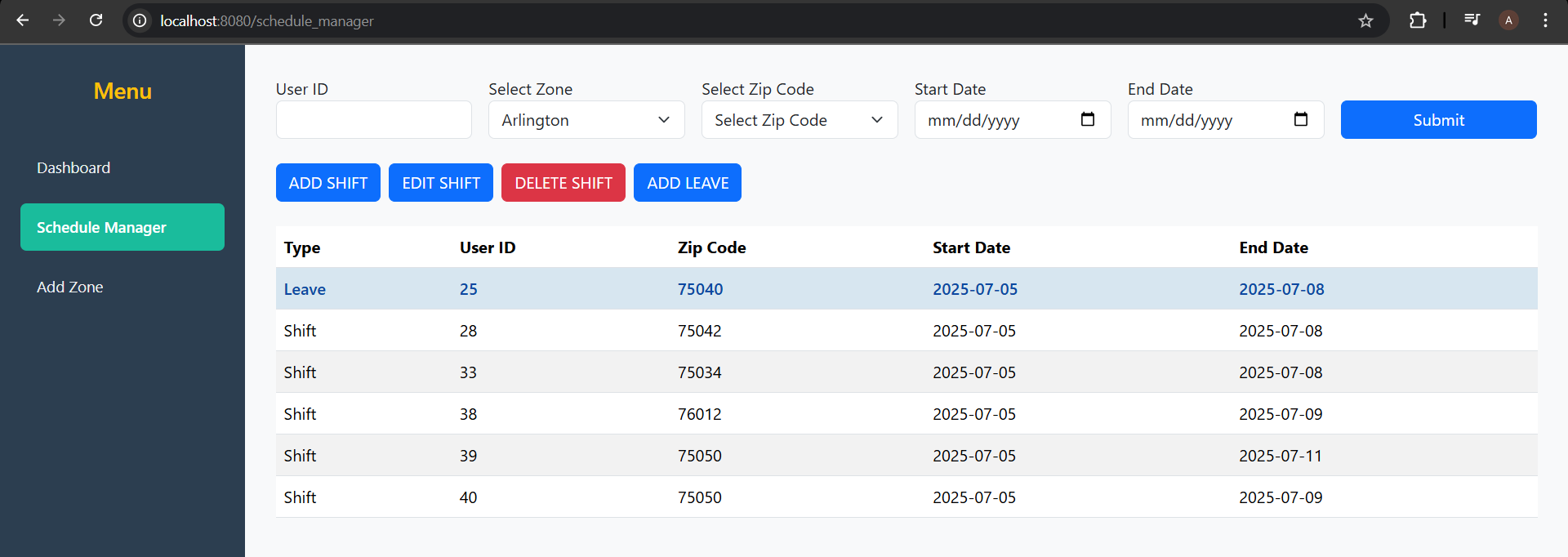
<http://localhost:8080/>



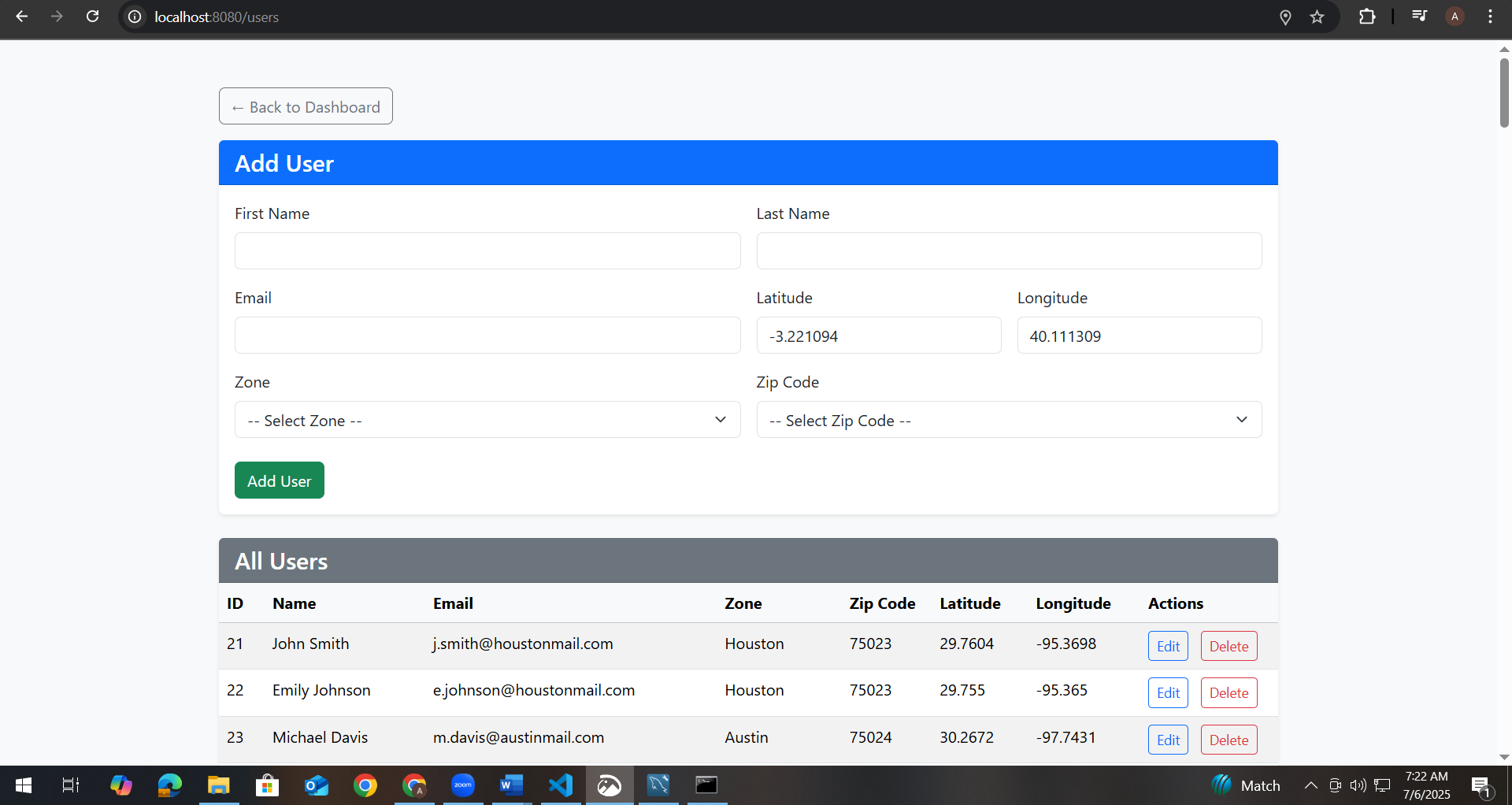
This will load the dashboard interface, displaying the map, zone/zip filters, and user data table.

Other available URLs:

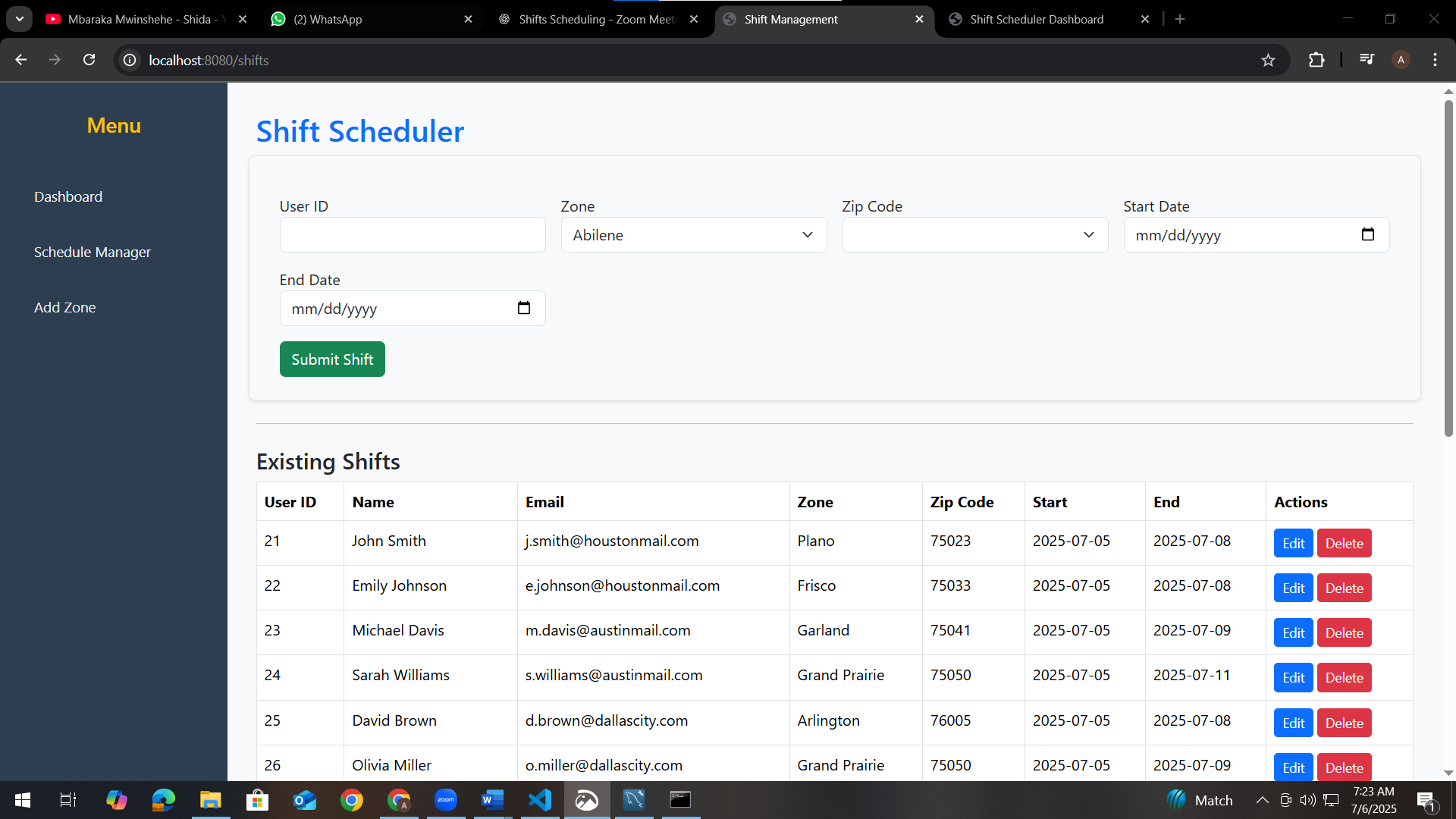
* <http://localhost:8080/schedule_manager> – manage shifts and leaves



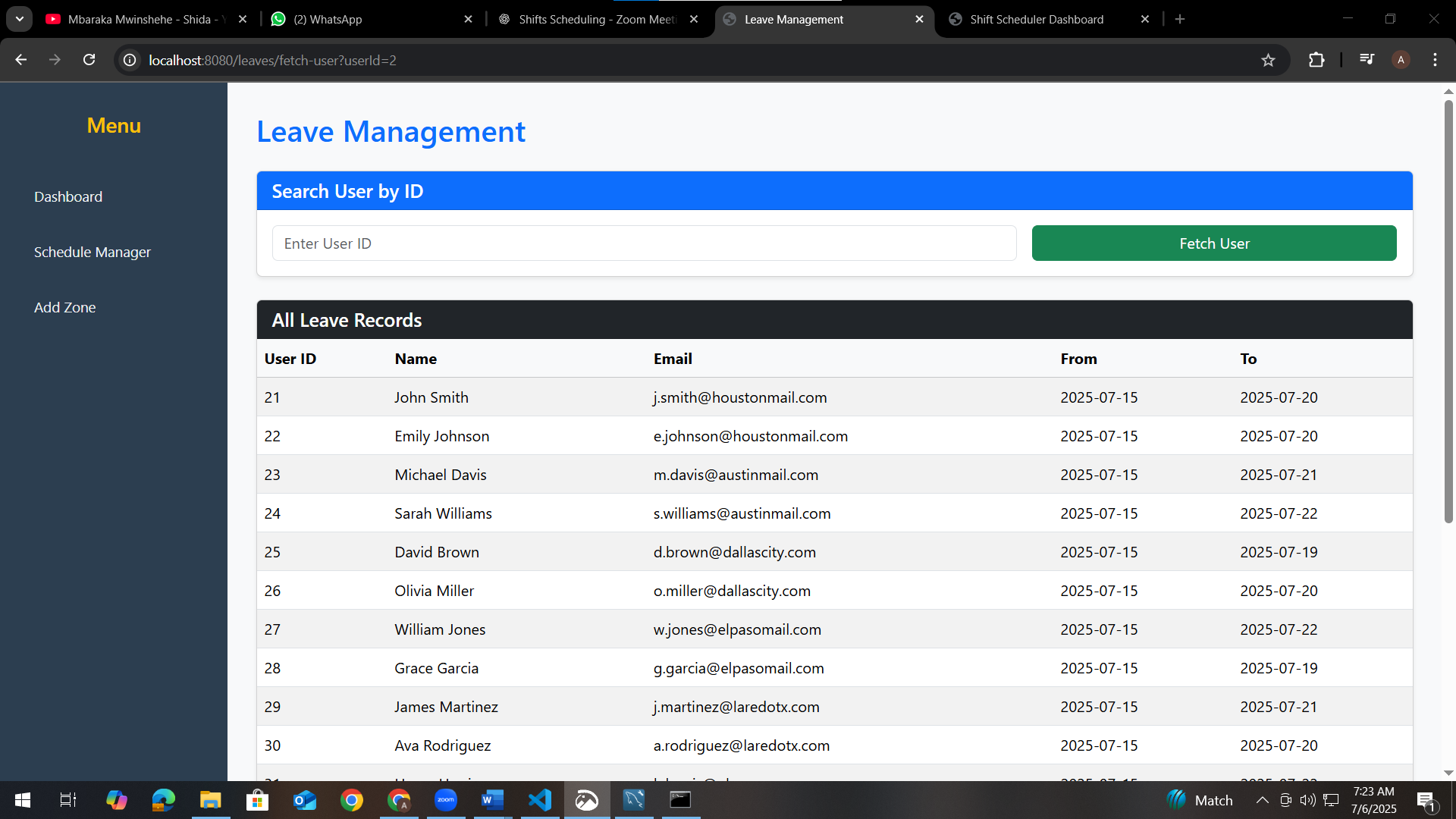
* <http://localhost:8080/users> – view user records



* <http://localhost:8080/shifts> – view shift records



* <http://localhost:8080/leaves> – view leave records



Let’s now go through each part, one by one.

**1. The controller Package**

Let’s start with the controller package. This is where we handle all HTTP requests and route them either to a page or to a backend API.

**Controller Package – Code Walkthrough**

In the controller package, I implemented the logic that handles all HTTP requests. This is where user interactions from the browser are translated into backend operations or page rendering. The controllers are grouped into two categories: those that return **views** (HTML pages), and those that provide **REST APIs** for dynamic data operations.

**1. DashboardController.java**

I created DashboardController to handle requests to the /dashboard route. When the dashboard page loads, this controller retrieves all zones from the database and passes them to the view using the model. These zones populate the Zone dropdown in the filter form. Although this controller doesn’t return JSON, it plays a key role in initializing the map and form elements with backend data.

**2. HomeController.java**

The HomeController is a simple class that handles root-level routing. If we decide to build a homepage, landing page, or redirect logic for authentication in the future, this controller provides a clean entry point.

**3. LeaveController.java**

This controller manages the leave form operations. When a supervisor inputs a user ID to register leave, LeaveController fetches the user, displays their details, and handles the submission of leave dates. It also populates the list of existing leave records, which are shown at the bottom of the leave management page.

**4. ScheduleApiController.java**

I used this controller to expose REST API endpoints for shift operations. It includes endpoints for adding, updating, and deleting shifts via the /api/shifts route. These APIs are consumed by the forms in the schedule manager page, allowing dynamic submission and real-time database updates. This controller is responsible for all backend shift logic that doesn’t involve page rendering.

**5. ShiftController.java**

This controller handles the HTML view for managing shifts. When a user accesses the /shifts URL, ShiftController loads all the necessary data — including zones, zip codes, and user details — into the form. It supports the display of existing shifts in a table format and provides routing to shift edit and delete confirmation pages.

**6. UserApiController.java**

This is one of the most critical controllers in the entire project. It exposes the /api/users endpoint, which returns a list of UserDTO objects that power both the dashboard map and table. It accepts optional query parameters such as zone, zip, startDate, and endDate for filtering. Based on the filters, it queries the relevant shifts and joins them with user, zone, and zip data to return clean, structured user records. This controller enables all real-time map filtering functionality.

**7. UserController.java**

I used UserController to render the user management interface. It supports creating new users, editing existing ones, and deleting records. It loads zone and zip code lists into the model to populate the dropdowns, and it handles the form submission logic for adding or updating user profiles, including location coordinates.

**8. ZipCodeController.java**

This controller manages zip code operations. It handles routes for creating, editing, and deleting zip codes, and loads zip code records into the frontend. It works in tandem with zones and is used heavily in the zipcodes.html and zone assignment forms. Although not exposed as an API, it provides full CRUD logic for zip code management.

**9. ZoneController.java**

I used ZoneController to implement all logic related to zones. It renders the zones.html view, allows supervisors to add or update zone names, and supports assigning multiple zip codes to each zone using a multi-select field. It also exposes an API to retrieve zip codes under a given zone — this is used by the JavaScript in both dashboard and schedule manager forms to update zip code options dynamically.

**2. The dto Package**

Now let’s look at the DTOs — these are Data Transfer Objects used to send structured data to the frontend.

**1. ShiftDTO.java**

I designed the ShiftDTO class to represent shift-specific data in a clean and transportable format. This DTO contains key fields such as:

* User reference
* Zone ID and name
* Zip code ID and value
* Start and end dates of the shift

I used this class when packaging shift data for listing shifts, editing entries, or transferring shift records between views. It simplifies how shift data is presented in tables or filtered in logic without exposing the full entity relationships.

**2. UserDTO.java**

This is the most important DTO in the entire application.

I built UserDTO to aggregate multiple pieces of data from related models — such as the User, Shift, Zone, and ZipCode entities. It includes:

* User ID, name, and email
* Latitude and longitude for map plotting
* Zone and zip code labels
* Start and end dates of the assigned shift

This DTO is returned by the /api/users endpoint and is used directly by the **dashboard.html** page. The map markers and the user table below the map are both populated from the list of UserDTO records. It ensures the frontend receives exactly what it needs, no more, no less.

**3. ZipCodeDTO.java**

The ZipCodeDTO was created for use in the zip code dropdowns. Rather than sending the full ZipCode entity — which includes zone relationships and other metadata — I simplified the response to just id and code.

This DTO is returned by the endpoint /api/zones/{zoneId}/zipcodes and used in both the **dashboard filter** and **schedule forms** to dynamically populate the zip code selector when a zone is selected.

**3. The model Package**

Now let’s move on to the model layer — this is where we define our database entities.

**1. User.java**

I began by designing the User model to represent each individual in the system.

Each user has an id, firstName, lastName, and email. I also included latitude and longitude fields to support our map-based location display on the dashboard.

I created relationships between users and other key entities:

* Each user is assigned to one Zone and one ZipCode, so I used @ManyToOne relationships for both of those.
* I also linked users to shifts and leaves using @OneToMany mappings. This allows me to easily fetch a user’s full shift or leave history when needed.

This model forms the foundation for all user-related operations in the system, including map display, shift scheduling, and filtering.

**2. Shift.java**

Next, I implemented the Shift model, which is the backbone of the scheduling logic.

Each shift includes:

* A reference to the User assigned to the shift
* The Zone and ZipCode where the shift occurs
* And the startDate and endDate of the shift

All of these are modeled using proper JPA annotations like @ManyToOne, so each shift record knows exactly who it belongs to and where it’s assigned.

I also made sure to include constructors and accessors so this entity integrates smoothly with Spring Data JPA and DTO mapping.

This is the entity used whenever a supervisor adds, edits, or deletes a shift. It’s also what the dashboard reads to filter and display data by zone, zip, or date.

**3. Leave.java**

I kept the Leave entity simple but effective.

Each leave record is tied to a User, and has a leaveDate field.

I used @ManyToOne to connect it to the user and made sure the leave date is stored in LocalDate format for proper comparison and filtering.

This model is used exclusively in the Leave Manager tab — when a supervisor records days a user is unavailable.

**4. Zone.java**

The Zone entity is designed to group users and zip codes geographically.

Each zone has an id and a name — like Austin or Dallas.

Then, I linked each zone to a list of ZipCode records using a @OneToMany(mappedBy = "zone") relationship. This is what allows me to dynamically load zip codes when a zone is selected in the dashboard or the shift form.

Zones are also linked to users and shifts, so we know where every activity is occurring.

**5. ZipCode.java**

Finally, I created the ZipCode model, which represents specific areas within a zone.

Each zip code has:

* An id and code (e.g., 75001)
* A reference to its Zone using @ManyToOne

Just like users and shifts, zip codes are tightly bound to zones, ensuring we can filter or validate data accurately.

This model is essential for populating dropdowns in the frontend and for filtering users by location.

**4. Repository Package – Code Walkthrough**

Now let’s move on to the repository package. This is where I defined all the interfaces used to interact with the database.

Each file in this package is a **Spring Data JPA repository**, meaning I didn’t need to write boilerplate SQL. Spring automatically generates the necessary query implementations at runtime based on method names and annotations.

Let me explain each repository interface one by one.

**LeaveRepository.java**

In the LeaveRepository, I extended JpaRepository<Leave, Long> so I could perform all basic operations like saving, deleting, or retrieving leave records without writing any custom SQL.

I used this repository in the leave controller to:

* Save new leave entries
* Retrieve leave history for specific users (if needed in the future)

It’s simple and efficient, since most leave operations are straightforward inserts.

**ShiftRepository.java**

This repository was more complex. I extended JpaRepository<Shift, Long> to inherit all basic CRUD operations, but I also defined **custom query methods** for filtering shifts.

For example, I added methods to:

* Find shifts by zone ID and zip code ID
* Filter shifts between two dates
* Retrieve all shifts belonging to a specific user

These methods were critical for powering the Dashboard API, which needs to return only active users within a certain zone, zip, and date range.

I also used @Query and Spring’s derived query naming conventions to avoid writing raw SQL.

**UserRepository.java**

The UserRepository was central to the application.

It extended JpaRepository<User, Long> and was used in two major places:

1. In the API that returns the list of users displayed on the dashboard
2. In the shift and leave controllers, where I fetched user details using just their ID

One key method I added was Optional<User> findById(Long id) to support user auto-filling in the frontend when a supervisor types in a user ID.

I also used joins in queries when building the dashboard, especially when assembling the UserDTO, which combines user, shift, zone, and zip code info.

**ZipCodeRepository.java**

The ZipCodeRepository was used to fetch zip codes, especially when filtering them by zone.

I created a method like:

* List<ZipCode> findByZoneId(Long zoneId)

This method was used when a zone is selected on the dashboard or shift form — so only zip codes under that zone are shown in the dropdown.

I also made sure zip codes could be created and deleted when managing zones.

**ZoneRepository.java**

In ZoneRepository, I provided access to all zone-related data.

This repository was used to:

* Load all zones for the dashboard filter
* Fetch specific zones when editing shifts or leaves
* Delete zones cleanly when required

One key method I added was to fetch **zip codes by zone**, which was essential for populating the dependent dropdowns dynamically.

I kept the relationships between zones and zip codes clean, so I could use cascading deletes if needed in the future.

**Thymeleaf Templates – Code Walkthrough**

In this part of the application, I focused on the **resources/templates** folder. This is where all the frontend views live. I used **Thymeleaf** as the templating engine to render dynamic HTML on the server side, while also integrating JavaScript for interactivity.

Let me explain the key templates I implemented.

**1. dashboard.html**

This is the core dashboard screen that supervisors see first.

I included:

* A **Zone** dropdown
* A **Zip Code** dropdown that loads dynamically based on the selected zone
* **Start Date** and **End Date** filters

Once the filters are applied, the frontend sends a request to the /api/users endpoint, which returns all relevant user and shift data.

I used **Leaflet.js** to render a live, interactive map of Texas. Each user is marked on the map with a custom icon, and clicking the icon shows user info — such as name, ID, email, zone, and zip code.

Beneath the map, I dynamically populate a **data table** using JavaScript. The table lists all filtered users with their shift start and end dates.

This template uses a combination of **Thymeleaf for initial data** (zones), and **JavaScript for real-time updates**.

**2. schedule\_manager.html**

I created this page to centralize the actions for:

* Adding shifts
* Editing shifts
* Deleting shifts
* Adding leave records

At the top, I included a filter form for zone, zip, user ID, and date. Below that, there are four main buttons — each redirects to a specific page (/shifts, /leaves, etc.).

After a shift or leave is added, the lower part of the page shows a **data table** with all filtered shift records. I added styling logic to highlight rows marked as leave to make them visually distinct.

This template combines Thymeleaf, Bootstrap styling, and JavaScript-based filter logic to handle user input dynamically.

**3. shifts.html**

This is the shift entry and management form.

At the top, I placed a **form for adding a shift**, which includes:

* User ID
* Zone and Zip Code dropdowns
* Start and End Date inputs

Once submitted, the shift is stored via a POST to /shifts/save.

Below the form, I added a **table showing existing shifts**, with columns for:

* User details
* Zone and Zip
* Start and End Dates
* Actions like Edit and Delete

Clicking Edit or Delete routes to their respective pages.

I also added JavaScript logic to dynamically load zip codes based on the selected zone using an AJAX call to /shifts/zone/{zoneId}/zipcodes.

**4. leaves.html**

This page handles leave management.

At the top, there's a **form to search for a user by ID**. Once found, their name and email auto-fill in the leave form.

The supervisor can then input a **from-date** and **to-date** to register a leave. I made sure the form pre-fills hidden zone and zip ID fields so everything stays linked properly.

At the bottom, I included a **leave records table** showing:

* User ID
* Name
* Email
* From and To Dates

If no records exist, a message appears indicating that. All content here is powered by Thymeleaf model attributes.

**5. users.html**

This is the user management page.

I designed it with a **form to add or update users**, including fields for:

* First Name, Last Name, Email
* Latitude and Longitude
* Zone and Zip Code selection

For user convenience, I added **auto-geolocation** using JavaScript — so when the form loads, it attempts to fetch the current location and populate the latitude and longitude fields automatically.

Below the form, there’s a **table of all users**, with columns for:

* Name, Email, Zone, Zip, Lat/Lng
* Actions to Edit or Delete each user

This template is important when onboarding new users or updating their location.

**6. zones.html**

I built this template to allow supervisors to manage zones and their associated zip codes.

At the top, there’s a **form to create or update a zone**, with:

* A zone name input
* A multi-select dropdown to assign one or more zip codes

After submission, the zone and its zip codes are saved to the database.

Below the form, I included a **zones table** that shows all existing zones, with options to Edit or Delete. Deleting prompts a confirmation popup to avoid mistakes.

**7. zipcodes.html**

This is a dedicated page to manage zip codes.

There’s a simple form that allows:

* Entering a zip code value
* Assigning it to a zone

Once submitted, the zip code is saved via a POST request.

Below the form, I rendered a **table listing all zip codes**, their IDs, and the zone they belong to. Actions to Edit or Delete each zip code are included.

This is essential for maintaining an accurate mapping of geographic regions within the application.

**application.properties**

Lastly, this file configures the Spring Boot application — for example, the database connection details like username, password, and JDBC URL for MySQL.

The application.properties file configures essential settings for the Spring Boot Shift Scheduler application, including database and server details. I specified the MySQL database connection by setting the JDBC URL to jdbc:mysql://localhost:3306/shift\_demo, with root as both the username and password. Hibernate is configured to automatically create or update tables using spring.jpa.hibernate.ddl-auto=update, and SQL statements are printed to the console with spring.jpa.show-sql=true for debugging. The dialect is set to MySQL8Dialect to ensure compatibility with MySQL 8, and the application runs on port 8080 unless otherwise changed. These settings must be updated on the client machine to match their local MySQL credentials and port availability before the application can be successfully launched.