**Employee System Database**

Technical Manual

Class: CPSC 488 Section 2

04/21/2023

Maya Smallwood [mns1013@sru.edu](mailto:mns1013@sru.edu)

Paul Schlabach [pms1006@sru.edu](mailto:pms1006@sru.edu)

Bradon Walters [bdw1006@sru.edu](mailto:bdw1006@sru.edu)

**Contents**

[Overview](#_Toc132912711)

[UMLS](#_Toc132912712)

[Implementing Registration & Login System](#_Toc132912713)

[Implementing Importing & Downloading Excel files](#_Toc132912714)

[Pre-Populate Database Importing a SQL script file](#_Toc132912715)

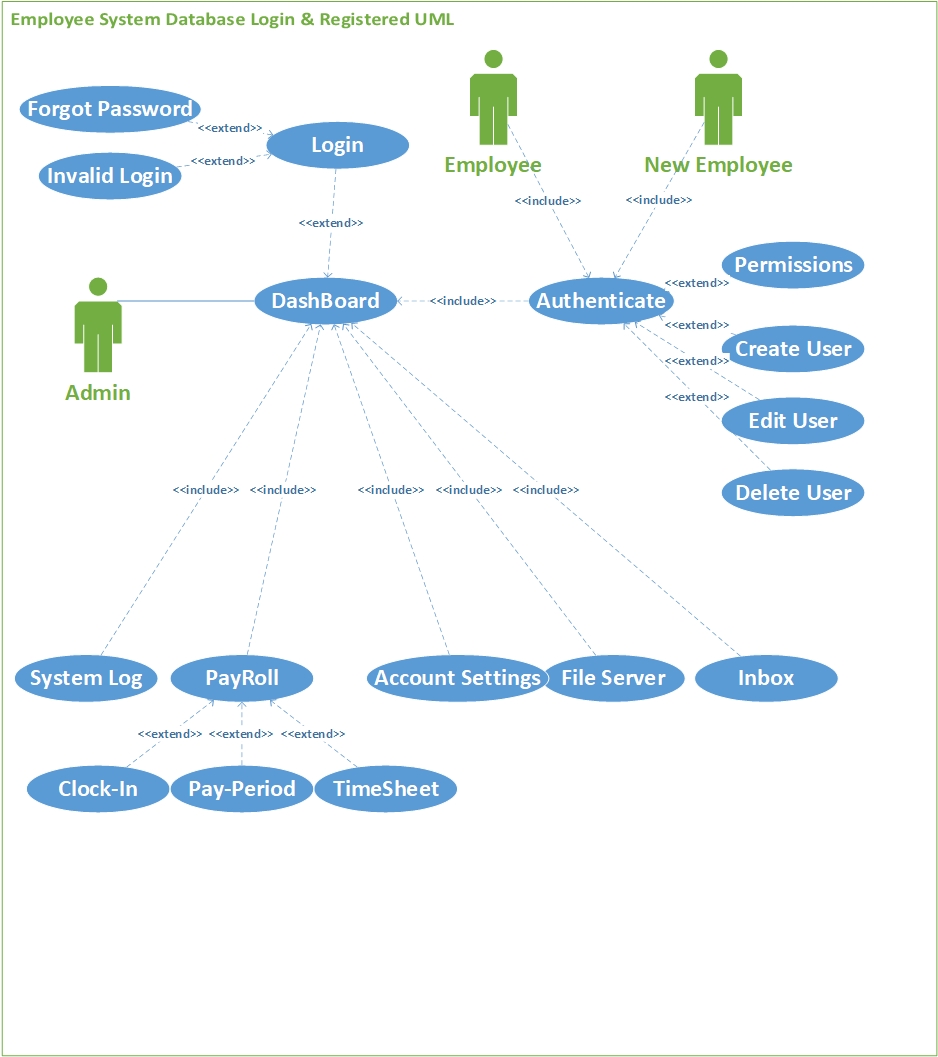
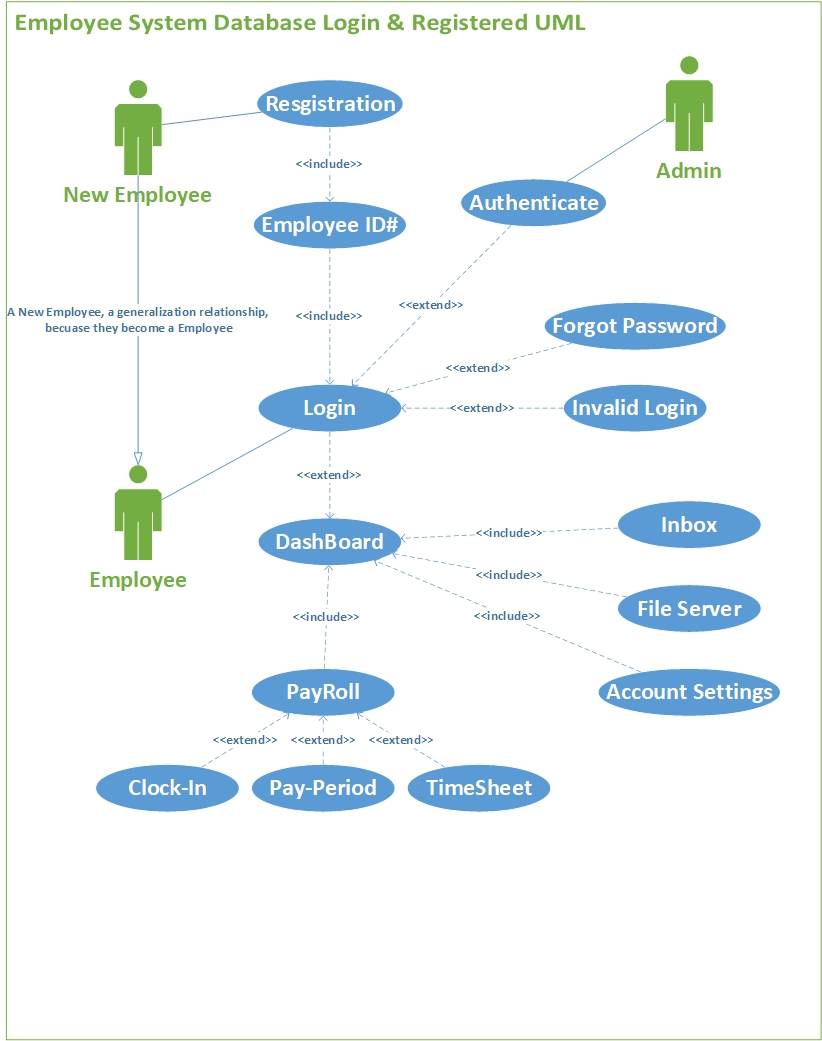
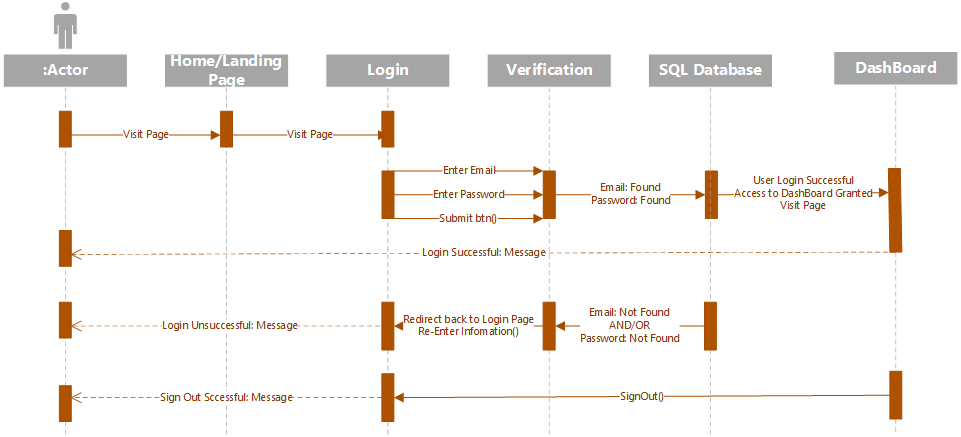
[Resources](#_Toc132912716)

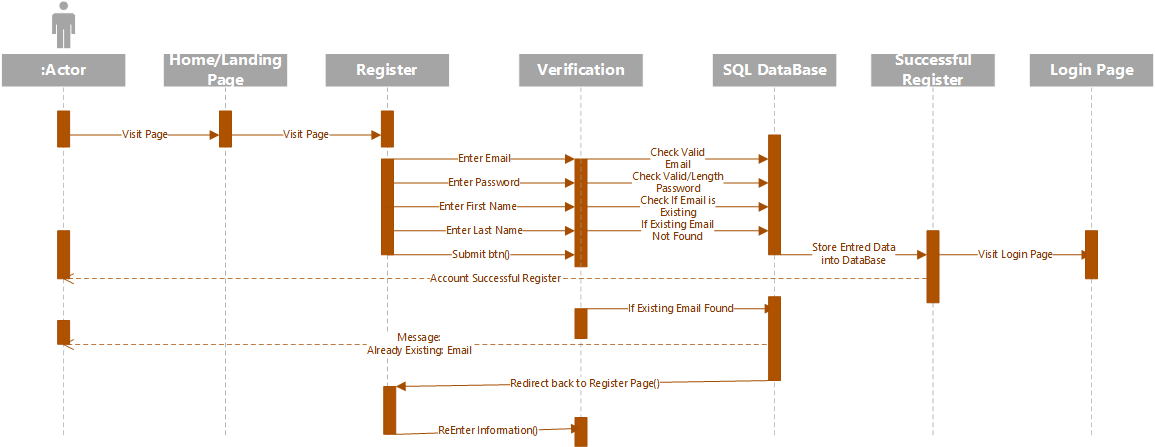
[Application Properties](#_Toc132912717)

# Overview

This workspace is intended to act as an addition to an existing project or can be configured into something more; this workspace is made up of an employee system database that utilizes MySQL Workbench to manage employees while using Spring Boot Thymeleaf Model, View, and Controller (MVC) framework to develop the web applications. The program allows the registration of new users, editing of users, and deletion of users. It also has login authentication based upon whether the user’s role is Admin or an Employee taking the user to the apportion landing page. It enforces the password policy using the Spring Framework Security of the BcryptPasswordEncoder library. While also using Models that create entities and connections to MySQL. To use this program with their projects, programmers should copy the classes, controller format, models, and security config into their projects. The Users and UserRepository may need to be configured for your likely, considering that our main classes work with our registration pages..

# UMLS

Figure 1. Case Diagram for Admin ControlFigure 2. Case Diagram for Employee Control  
Figure 3. Sequence Diagram for User Login

  
Figure 4. Sequence Diagram for User Register

# Implementing Registration & Login System

*(For more references in relation to Thymeleaf documentation please look at our “SpringBoot Thymeleaf documentation” document)*

**Configuration, implements security authentication into the project’s webpages:**

***SecurityConfig.java-***This is our project’s security implementation, that uses the following tag, @*EnableWebSecurity* & @*Configuration*, this connects to a @Bean *SecurityFilerChain*, Defines in Spring and specifies what parts of the website requires a login to access in this case every page that is not listed in *.requestMatchers(“ ”).permitAll()* will require the user to login before they gain access to any other pages.

**Controllers, implemented with “Model View Controller” (MVC) allowing calls to listed/assigned webpages:**

Requires the @controller tag that allows Thymeleaf to locate and call web pages from the src/main/resources/templates folder. Additionally, the tags that are associated with the methods related to Thymeleaf as in the following: @PostMapping is used to map HTTP POST requests onto specific handler methods. It is used to handle form submissions and create new resources. @RequestMapping is a general-purpose annotation used to map HTTP requests onto specific handler methods. It can be used to handle any HTTP method (GET, POST, PUT, DELETE, etc.). @GetMapping is a specific type of @RequestMapping annotation used to handle HTTP GET requests.

***LoginController.java-***Configures how our Login and Registration pages are to communicate with our Thymeleaf forms that are embedded in HTML.

***MainController.java-***Refences web pages that link to one another, as well as referencing the USER modeled table to display listed users that are store in the database, “TestingListUsers”.

***UserController.java-***Referencing to the USER modeled tables in the database in requires to grabbing all users, deleting, adding, and updating.

**Models, implements a connection to MySQL Workbench creating the required tables and method/call references to pull/push to and from the database:**

***User.java-*** Our USER Model creates an @Entity table allowing us to store any registration/pushed data into the database and will show up according based on the below attributes listed under @Id.

***UserPrincipal.java-*** The UserPrincipal class helps pull specific data from the database whenever that User is granted authority and authentication from our security config.

**Repositories, Connecting to Models and database queries:**

***MyUserDetails.java-*** Verifies the user’s information all you to pull info from the USER model.

***UserRepository.java-*** Connects to the database in the Queries when selecting data.

**Services, Connects to controllers, database, and authority settings:**

***UserService.java-*** Pulls a data, connecting to the UserController.

***MyUserDetailsService.java-*** Loads the logged in user, based on that username, allowing authority and making sure that the user existing in the database.

**src/main/resources/templates, HTML pages that require no authentication and one that does require authentication:**

*Every HTML page requires the following in the HTML Tags:*

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">

*Every HTML page that requires authentication needs the following in the HTML Tags:*

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"

xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity6">

**Require no authentication/no login:**

***index.html-*** This is our home/landing page whenever you type in “localhost:8080” this is are main directory that takes you into the login, registration.

***registration.html-***

***registrationsucces.html-*** This is a linking page that just returns a message to the user that they successfully created a account, we advise to check your database scheme to make sure it appears, it then will direct the user to go to the login page.

***login.html-*** This web page is where we create a form that allows a connection to our database to look for any of the information type in from the **<input> tag** while searching through that specific database. Additionally, the form is made up of the opening tag:   
“th:action="@{/login}" method="post" th:object="${user}”

th:action and th:object are two attributes used to bind a form to a Spring MVC controller, in this case it connects to our LoginController.

***loginsuccess.html-*** Similar to the “***registrationsucces.html***” page it returns a message to the user that they successfully logged into our system, and their information was found in the database. It then will direct the user to go to the dashboard page.

***TestingListUsers.html-*** On this page allows, the user to view a list of users that are stored in the database, additionally this webpage gives a view of how Thymeleaf attributes are able to communicate with different tables within the database.

**Requires authentication/login:**

***dashboard.html-*** Dashboard is our main page that requires the user to login in before they can access it, meaning it requires authentication.

# Implementing Importing & Downloading Excel files

**Controllers:**

***ExcelController-***

This is a Java Spring Controller class called ExcelController that handles requests related to Excel files. It provides three methods that map to specific URL paths.

The first method, "getFile()", handles a GET request to download a blank/pre-filled Excel file. It retrieves the file using the fileService, then sets the response headers to indicate that it should be treated as a downloadable attachment.

The second method, "upload()", handles a POST request to upload a filled timesheet file. It accepts a MultipartFile object as a request parameter and validates that it is an Excel file using a helper class called ExcelHelper. If it is valid, it saves the file using the fileService and returns a success message in the response body. Otherwise, it returns an error message. This can also be stored to a certain data table based on where it’s being set when it’s being uploaded.

The third method, "getAllTimeSheets()", handles a GET request to retrieve a list of all uploaded timesheet files. It retrieves the list using the fileService and returns it as a JSON response. If the list is empty, it returns a status code indicating that there is no content. If an exception occurs, it returns an internal server error status code.

***MainController-***

This is a Java Spring Controller class called MainController that handles requests related to linking between MVC webapges.

**Exception:**

This class just returns if there is an message errors as well as indicating if the uploaded file is bigger than what is allowed.

**Helper:**

***ExcelHelper*-** that provides functionality to create and read Excel files in the **.xlsx** format.

1. The class imports several classes from external libraries:
   * **org.apache.poi** for handling Excel files in Java.
   * **org.apache.tomcat.util.http.fileupload.ByteArrayOutputStream** for handling output streams.
   * **org.springframework.core.io.InputStreamResource** for handling input streams.
   * **org.springframework.jdbc.core.RowCallbackHandler** for handling JDBC rows.
   * **org.springframework.web.multipart.MultipartFile** for handling uploaded files.
2. The class defines three constants:
   * **TYPE** is a **String** that defines the MIME type of an Excel file in the **.xlsx** format.
   * **HEADERs** is an array of **String**s that defines the column headers for the Excel file.
   * **SHEET** is a **String** that defines the name of the worksheet in the Excel file.
3. The class defines two methods:
   * **dataToExcel** creates an Excel file from a list of **TimeSheets** objects. It first creates a new **Workbook** object, then creates a new **Sheet** object within that workbook. It then writes the column headers to the first row of the sheet, and iterates through the list of timesheets, writing the values to the subsequent rows of the sheet. Finally, it writes the workbook to a new **ByteArrayOutputStream** object, which is then used to create a new **ByteArrayInputStream** object that is returned.
   * **excelToData** reads an Excel file from an input stream and returns a list of **TimeSheets** objects. It first creates a new **Workbook** object from the input stream, then gets the **Sheet** object with the name specified by the **SHEET** constant. It iterates through the rows of the sheet, skipping the first row (which contains the column headers), and creates a new **TimeSheets** object for each row. It then iterates through the cells in the row, populating the fields of the **TimeSheets** object. Finally, it adds the **TimeSheets** object to a list, which is returned when all rows have been processed.
4. The class also defines a static method **hasExcelFormat** that takes a **MultipartFile** object (representing an uploaded file) and returns **true** if the file is in the **.xlsx** format, and **false** otherwise.

**Messages:**

***ResponseFile-*** Creates the constuctor layout for setting message names

***ResponseMessage-*** helps returns messgaes to the user

**Models:**

***TimeSheets-*** This is a Java code for a model class named "TimeSheets", which represents timesheets that employees use to record their work hours. The class is annotated with the "@Entity" annotation, which marks it as a JPA entity. It has three attributes: "id", "title", and "description". The "id" attribute is marked with the "@Id" annotation, which makes it the primary key of the entity. The class has a default constructor and a constructor with two parameters (title and description), and it also has getters and setters for all attributes. Finally, the class has a "toString()" method that converts the attributes to a string for debugging purposes.

***FileDB-*** This model structure of a file uploaded by a user in a web application, and provides getters and setters to access its properties.

**Repository:**

***ExcelRepsotiory-*** Connection through Timesheet model and long variables

***FileDBRepository-*** Connection through FileDB model and String variables

**Service:**

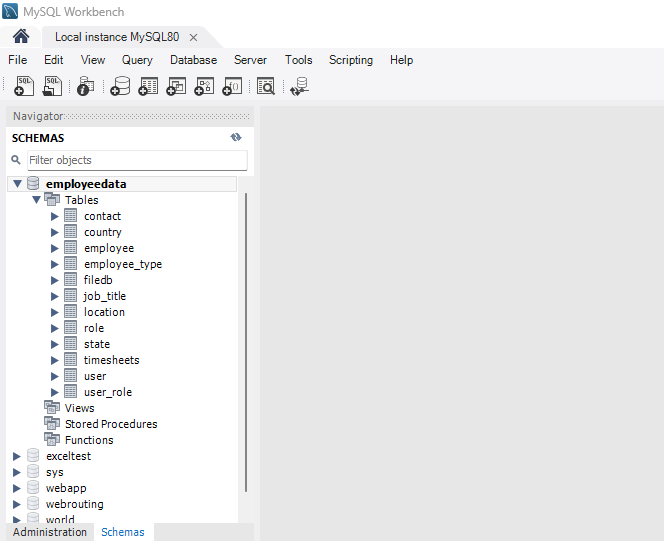
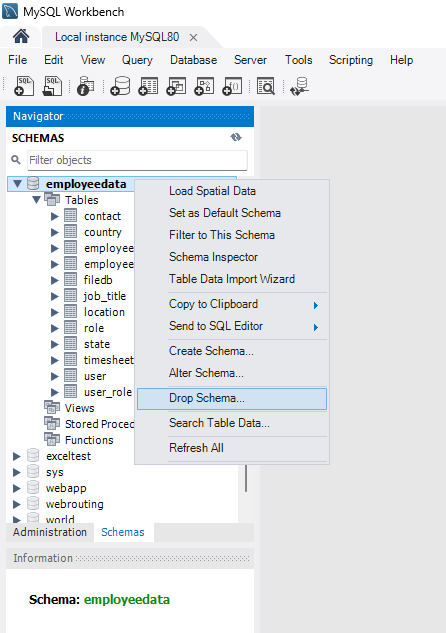
Both Services run a try and catch to make sure no errors occur, and if so act accordingly.

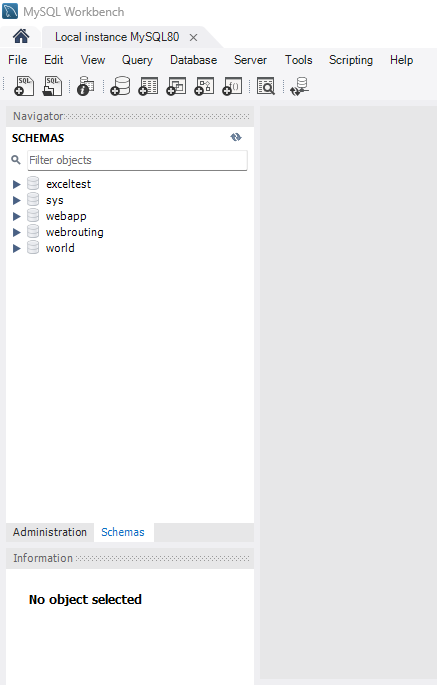
**HTMLS:**

***timesheet.html-*** Allow the user to download a blank timesheet and to allow the user to upload a timesheet to be stored in the database

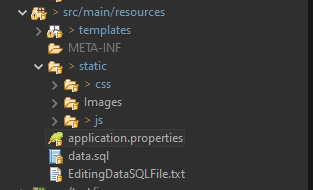
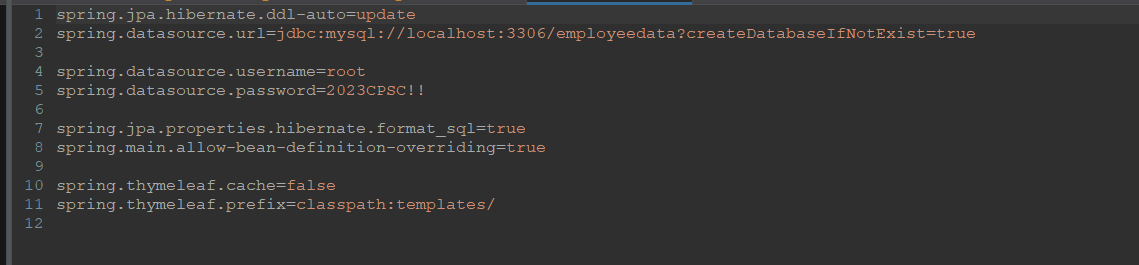
***viewTimesheet.html-*** Allows the user to view data from the database that corresponds to the Timesheet model data, and allows to view data from using the apache.poi library

# Pre-Populate Database Importing a SQL script file

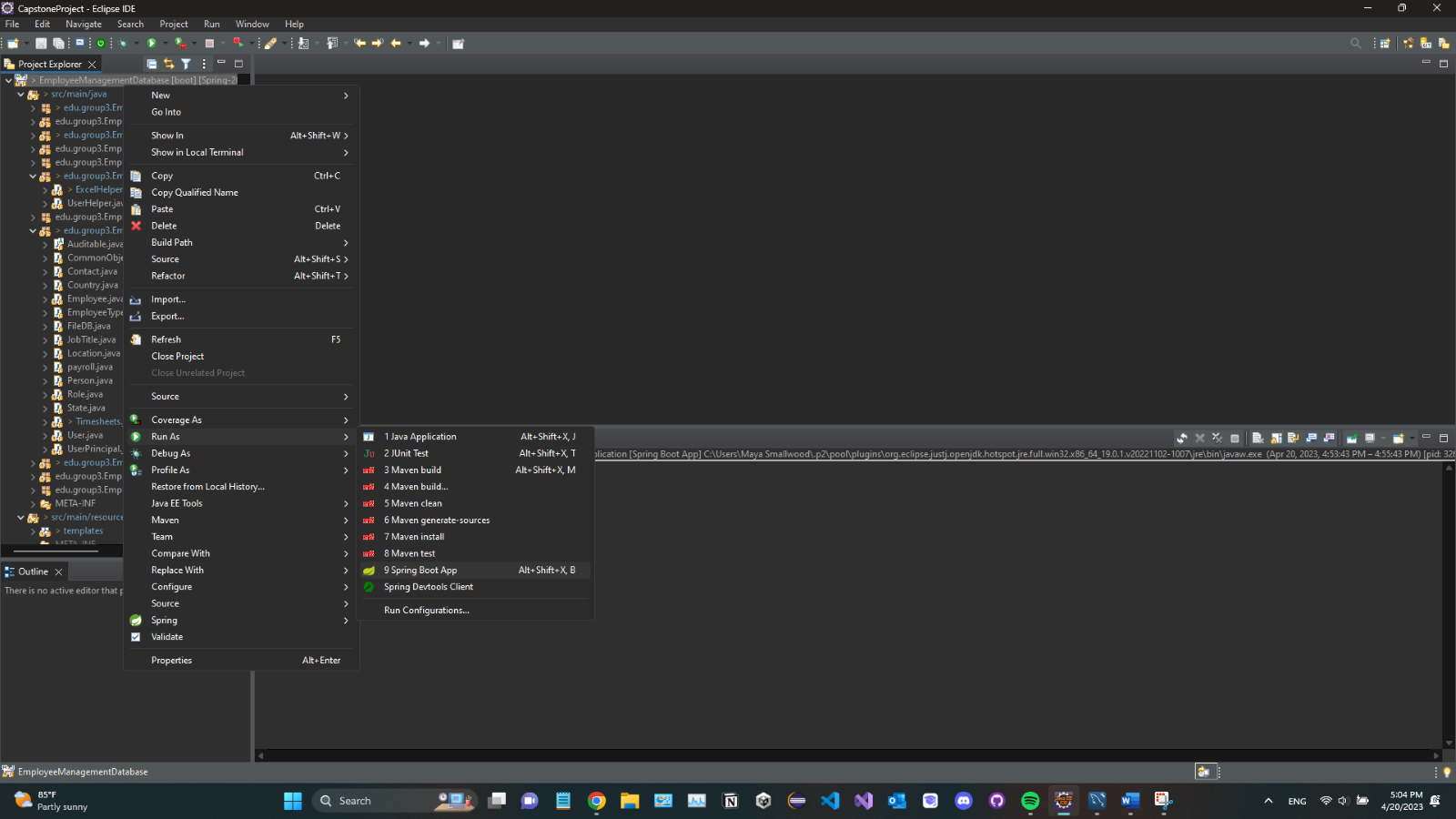
1. Drop the schema corresponding to our database name "employeedata" to ensure that the system can add in the new data. To do this, right-click on the schema and select "drop" in MySQL Workbench.  
     
   

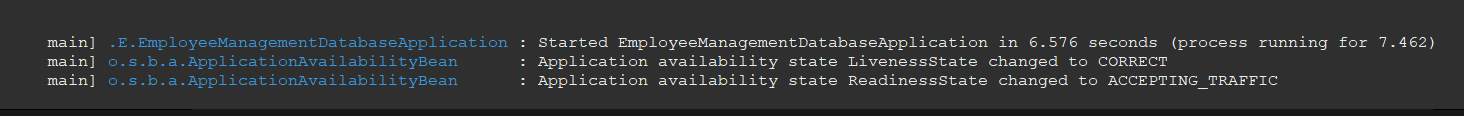


1. After importing our project to Eclipse IDE 2022-12, navigate to the "application.properties" file and ensure that the "spring.datasource.username" is set to "root" and that the "spring.datasource.password" is updated to your Workbench password. Failure to change the password will result in an inability to create or access the database.

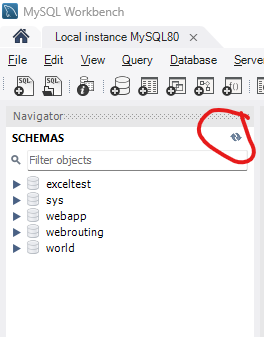
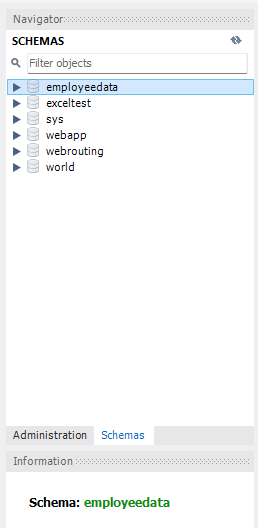
  


1. Right-click on our project and select "Run As SpringBoot App" to start the project. The system will display a message indicating that the project has been created successfully.

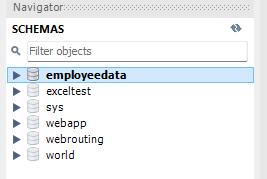




1. Return to MySQL Workbench and refresh the Scheme list for the database to display.

1. Double-click on the "employeedata" schema in the list to make it bold, indicating that you are focusing on this database. Then select Administration. Then select Data Import/Restore.



Graphical user interface, application

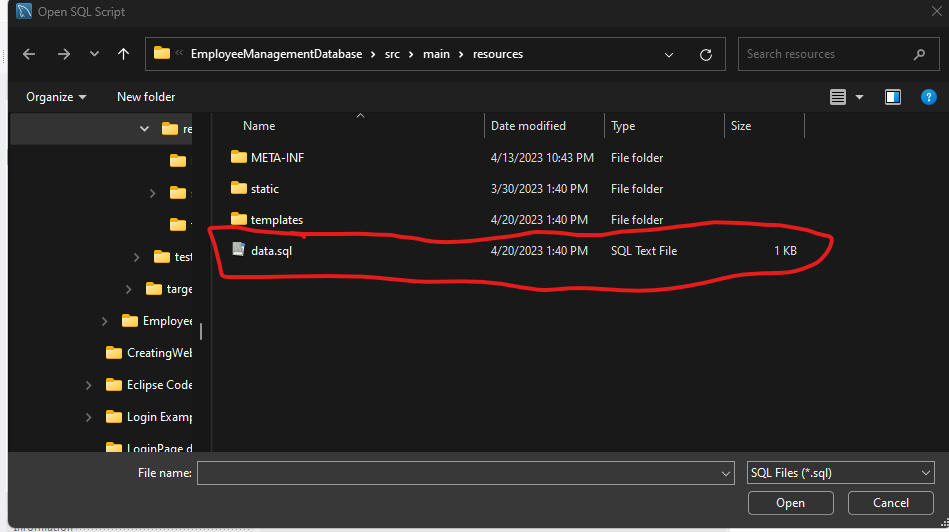
Description automatically generated  
Graphical user interface, text, application

Description automatically generated

1. Next Select Import from Self-Contained File, then point to where the project is stored, and select the data.SQL file, and finally make sure your Default Target Schema is set to the correct one.

Graphical user interface, application

Description automatically generated

Navigate to the "data.sql" file located in YourMainDict\EmployeeManagementDatabase\src\main\resources where you have the project stored   


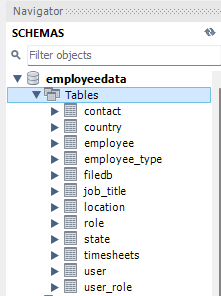
Then select Start Import

Graphical user interface, text, application

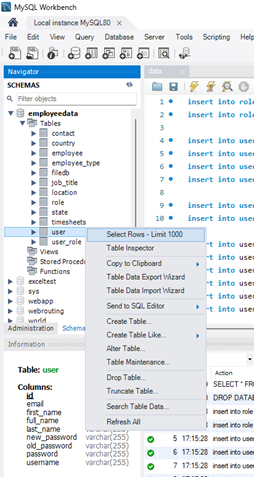
Description automatically generated

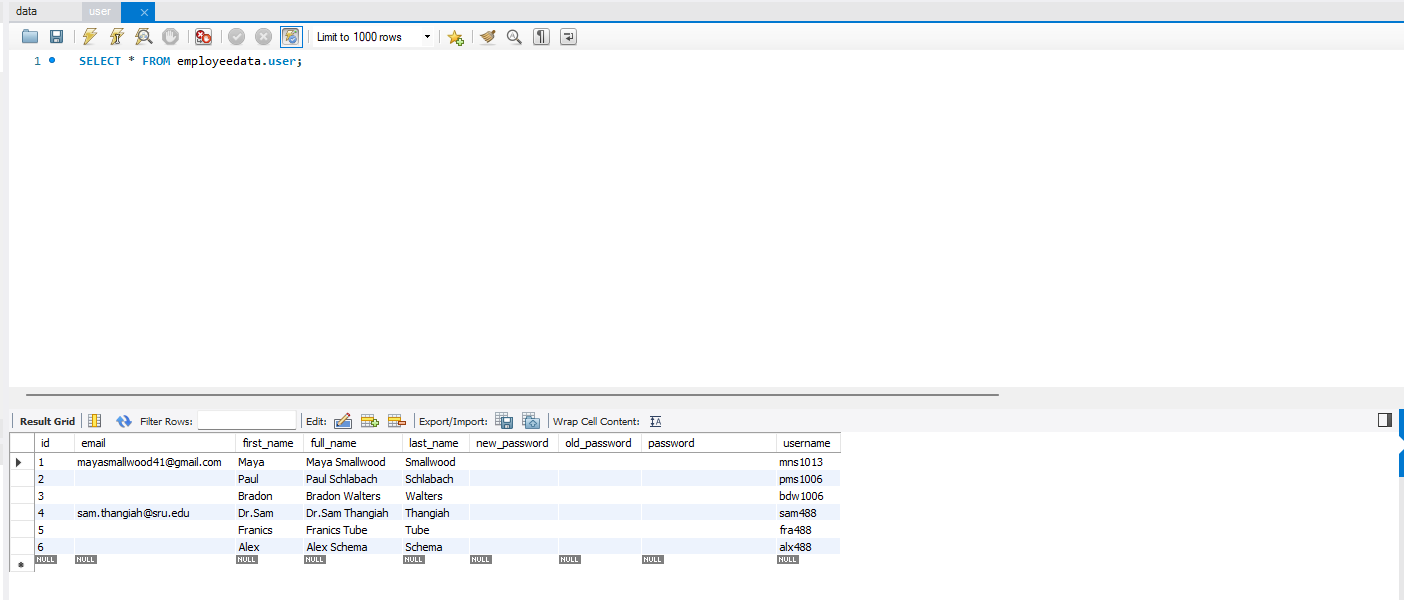
1. Graphical user interface, text, application

   Description automatically generatedOnce it is done running and importing data it will look like the following:
2. Next go back to Schemas and Refresh the schema list again and select the drop-down menu to display the tables.



1. To ensure that the data was entered correctly, select "user" and right-click to display its information. The table will now be populated with data from the data file and will display on the corresponding pages of our project.





# Resources

# Application Properties

**spring.jpa.hibernate.ddl-auto**=update

Used to embed a database and create a schema using the update attribute to update the connected Database every time a change is made and the program runs.

**spring.datasource.url**=jdbc:mysql://localhost:3306/demo?createDatabaseIfNotExist=true

The URL or link to access the MySQL workbench database. It is set to automatically create a schema rather than importing it from a SQL file.

**spring.datasource.username**=root

The username that is authorized to the server to be accessed.

**spring.datasource.password**=GAzebo23\!!

The password that is authorized to the server to be accessed. Will be required to be changed based upon the individual programmer’s password on that machine.

**spring.jpa.properties.hibernate.format\_sql**=true

**spring.main.allow-bean-definition-overriding**=true

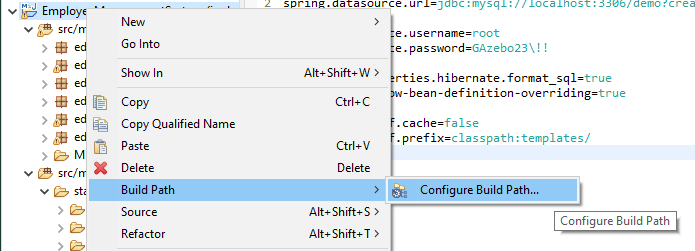
Both of these allows an overriding of the @Bean tag names, an error that would occur from time to time.

**spring.thymeleaf.cache**=false

**spring.thymeleaf.prefix**=classpath:templates/

Used to helped thymeleaf locate the HTML pages inside of templates, this is optional another fix is the following:

**Right click on your main project and navigate to the following:**

Build Path → Configure Build Path…  
  
Figure 3. Thymeleaf Build Path

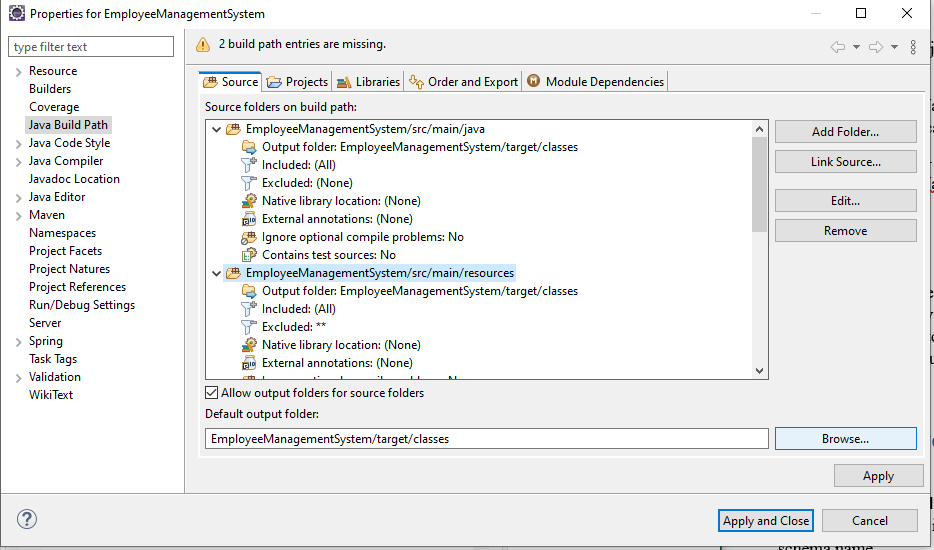
Tab over to Source → and then move down to “YourProject’sName/src/main/resources” → Double click “Excluded: \*\*”  


Figure 4. Thymeleaf Build Path

Next select Remove in the Exclusion Patterns Area”

then select Add Multiple… in the Inclusion Patterns Area:

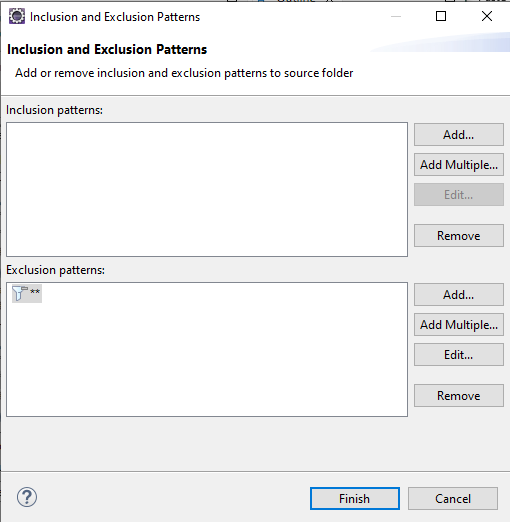
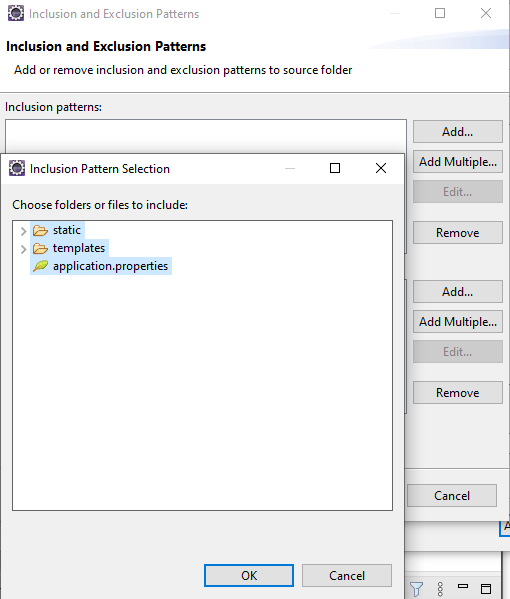
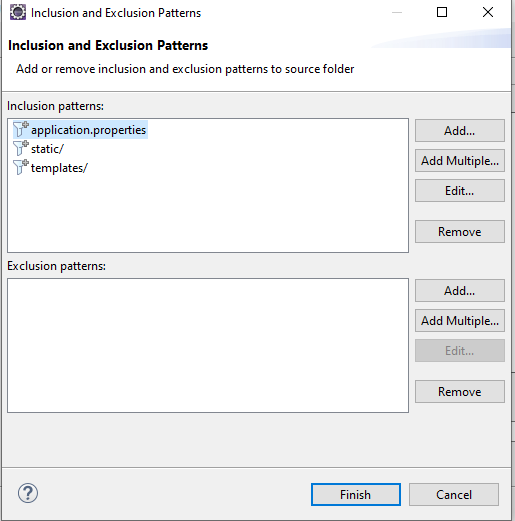


Figure 5. Thymeleaf Build Path

Do a Ctrl + A to select all the following files and then select OK  
Figure 6. Thymeleaf Build Path

Then select Finish  
  
Figure 7. Thymeleaf Build Path

Select Apply and Close and then wait for your project to finish building and now Thymeleaf will stop giving errors of: *“Cannot find template location: classpath:/templates/ (please add some templates or check your Thymeleaf configuration)”*\*Note: You may have to redo this progress if you close and reopen the project, or if you do a Maven→Update Project\*

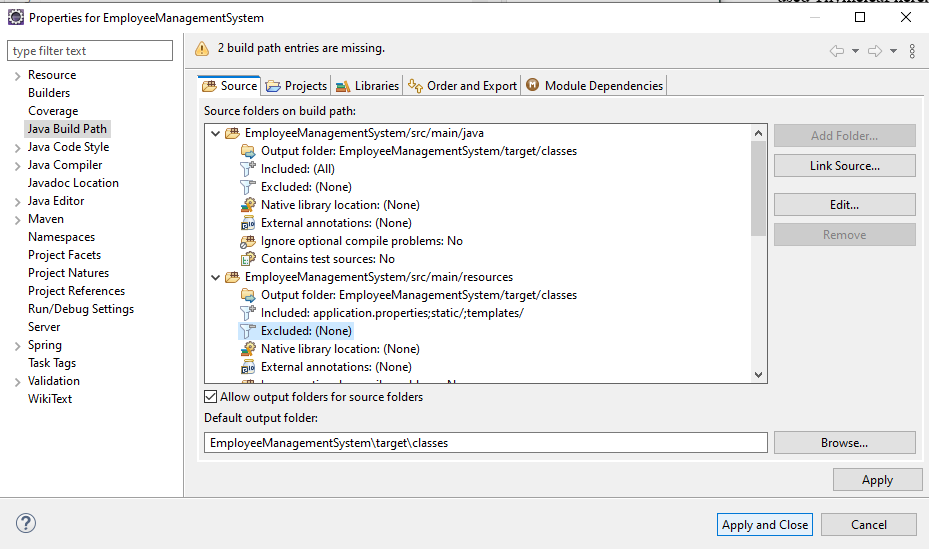


Figure 8. Thymeleaf Build Path