Functionality documentation

Content

Last updated: 13th of December 2024 15:02, by Marius

*src/main/java/com.zerolooksgood.demo.EmployeeRepositoryDemo/**

Controllers

*http://{domain}/***
*/controller/**
The controllers are what is used to control communication between the user and the backend application. It decides what happens when the user inputs different URL endpoints in their browser. This application features two controllers, the following documentation will explain their purpose and exactly what their methods do. The Controllers will automatically assume that the html files they are returning are stored in *src/main/resources/templates/**.

EmployeeController

@RequestMapping("/employees")
/employees/*
/EmployeeController/
The employee controller is the controller that does most of the work and is the main application and has the request mapping of "/employees/", this means that all the endpoints under this controller will have "/employees/**" as their endpoint, this is not a required feature, but it is recommended to separate the different controllers and/or sections of your application. . It handles all endpoints in the application (other than root) and is therefore vital to the application's functionality. It contains the following methods:

***/list/*
@GetMapping("/list")
list():
This method is what operates as the application's homepage and is the page that you're redirected to once you log in. This method calls the **findAll()** method from the employee service to extract all the employees stored in the database, this method returns the employees as a list sorted by ID in ascending order. The method then removes the first employee from the list because this is an employee used only for testing purposes. The method then adds this list as an attribute to a list and returns the list-employees html file for the user to view.

***/showFormForAdd/*
@GetMapping("/showFormForAdd")
showFormForAdd():
This method has the purpose of allowing the user to add a new employee to the database (if they have the sufficient permissions), when it's called the method creates an empty employee object which will be filled by the user later. It then adds the empty object to the model. It also actually adds a custom title to the model because it shares html form with the update method. It then returns the employee-form html file.

***/showFormForUpdate?employeeId={employeeId}/*
@GetMapping("/showFormForUpdate")
showFormForUpdate():
This method serves the purpose of allowing the user to edit an already existing employee (if they have the sufficient permission). When the method is called it searches its endpoint for a parameter named "employeeId" because this is what it uses to know which employee it is going to edit, it then checks if the employee exists and if it doesn't it redirects to the employee-does-not-exist error page. If the employee exists it extracts this employee form the database in the form of an employee object and sends it to the html form along with a custom header with the model, it then returns the employee-form to the user.

***/showFormForDelete?employeeId={employeeId}/*
@GetMapping("/showFormForDelete")
showFormForDelete():
This method serves the purpose of asking the user if the user wants to delete an employee (with sufficient permissions). This method doesn't delete the employee but displays them and asks if the user is sure that they want to delete this employee. This method also checks for an employeeId parameter to know which employee it is going to display, this method also contains the same logic as showFormForUpdate to check if an employee with that Id actually exists. If they do and the user is sure they wish to delete the employee, then the method will send a post request to **/delete/ containing the employee as an object to delete the employee.

***/delete/*
@PostMapping("/delete")
deleteEmployee():
The method serves the purpose of deleting the employee from the database, it uses JPA-repository's built in functionality to delete the employee. It receives what employee to delete through the attribute of the POST request and then it uses the findById() function to find the employee's id and then delete them by their ID. After this has been perform the user is then redirected back to the main page.

***/save/*
@PostMapping("/save")
saveEmployee():
This method receives a post request with an employee object in it, it then uses JPA-Repository's built-in functionality to save the object to the database. If the object already has an id (an employee is being updated) then it will override the already existing employee with hat id, and if the object doesn't have an id, then it will save it as a new one and it will get an Id generated by the database. After the employee has been saved the user is redirected back to the main page.

SecurityController

/
**/SecurityController/*
This controller doesn't have a request mapping so all it's method just use the root directory as their base mapping. This Controller contains most of the security related mappings of the application (and the root mapping).

***/*
@GetMapping("/")
index():
This is the method used for handling the root directory, it does nothing other than redirect the user to **/employees/list/.

***/access-denied/*
@GetMapping("/access-denied")
accessDenied():
This method is what the user is redirected to if they try to access a page of the website that they don't have the sufficient permissions to access, the method adds a custom header and body to a model and sends it to the error page html. It does this because the error messages use the same html file.

***/employee-does-not-exist/*
@GetMapping("/employee-does-not-exist")
employeeDoesNotExist():
This is where the user is directed if they try to update or delete a user that doesn't exist, this method uses the same logic as accessDenied(), just with a different header and body.

***/showLoginForm/*
@^GetMapping("/showLoginForm")
showLoginForm():
This is the page that the user is redirected to by spring security if they're not authenticated, it simply returns the login html.

DAO

/dao/
The DAO (Data Access Object) is what the application uses to communicate with the database.

EmployeeRepository

**/EmployeeRepository/*
Our application uses JPA Repository, this means that we don't have to do much coding

ourselves as JPA-Repo comes with a lot of built in functionality. We can construct custom methods (such as findAllByOrderByIdAsc() by naming it in a specific way:
findAll: states that the methods purpose is to find all employees.
ByOrder: states that the employees are to be sorted.
ById: states that the order is to order by Id.
Asc: states that the order is to be ascending.

Entity

/entity/
The entity is an object that takes the place of a database row. The information read from the database is converted into these objects so that they can be handled by the application. Our DAO is what performs this conversion from database row to java object.

Employee

**/Employee/*
The Employee object is what resembles the employee database rows in our application, it has four variables that resemble the four different columns of the database and has an annotation marking the Id as a generated variable, this means that our application doesn't generate the Id for any new employee that we create, the database does this once it is handed an employee object without an Id by the DAO.

The entity also contains getters and setters for every attribute. These may appear as if they're unused, but the JPA-Repo uses these whenever it constructs a new employee from a database row.

Security

/security/
Our application uses spring security to secure our application. This means that we don't have to do much coding as a lot of these features come built in, they simply must be configured. Spring security comes with its own built inn login page, but you can also use your own custom login page.

SecurityConfig

**/SecurityConfig/*
The Security Config class is where we configure all the security features of our application, spring security uses this class to dictate how to grant authorisation and what levels of authorisation you need for different parts of the application.

userDetailsManager():
The user details manager is what spring security uses to find the users that can be used to log in and what. You could have local in-memory users (hardcoded users), but our application connects to a set of tables in our database that store the users and the roles, the data source is the same as for the entity and JPARepository and is defined in application.properties. Spring automatically provides the method with the data source when it gets constructed and if you follow the default database schema for spring security then you won't have to do anything other than that.

filterChain():
The filter chain is what dictates the different rules for the http requests, ours contains four different configurable elements:

- **.authorizeHttpRequest():**
  Dictates the conditions for the http requests:
- .requestMatchers("{endpoint}.hasRole("{role}"):
  Dictates which role is required to access the specified endpoint.
- .anyRequest().authenticated():
  States that all the request to the application must be authenticated (logged in).
- **.formLogin():**
  Specifies information about the login process.
- .loginPage("{endpoint}"):
  Dictates which endpoint the user is sent to log inn.
- .loginProcessingUrl("{endpoint}"):
  Dictates which endpoint the login information is sent to in order to process the authentication, you can just use the built-in endpoint "/authenticateTheUser".
- .permitAll():
  Permits all requests to the login and authentication endpoint, even if the user isn't logged in.
- **.logout():**
  Dictates the details regarding the logout process.
- permitAll() allows all users to use the log out function.
- **.exceptionHandling():**
  Dictates specific details about the security exception handling.
- .accessDeniedPage("{endpoint}"):
  Dictates which endpoint the user is sent to if they lack sufficient permission to access a page.

Service

/service/
The service is the part of the application that implements the functionality of the DAO and allows us to use it in our application, it is also here we define all the methods that we'll be using to read and write to the database.

EmployeeService

**/EmployeeService/*
An interface for our EmployeeServiceImpl which dictates what functions the service class will have.

EmployeeServiceImpl

**/EmployeeServiceImpl/*
This is the implementation of the employee service, which contains all the methods that we use to perform actions on and with the database. It contains the following methods:

findAll():
This method serves the purpose of extracting all the employees stored in the database, it does this by using the custom method we made in JPA-Repository. It takes no input and returns a List<Employee> where the employees are sorted by id.

findById():
This method serves the purpose of returning a single employee when provided with an integer that is their Id. It does so by using JPA-Repo's built in function **findById()** and saves the result as an Optional<Employee> variable, it then returns the employee stored in this variable.

save():
This method serves the purpose of saving an object to the database, it uses JPA-Repo's built in function **save()** for this. If the employee object has an Id, then it will override the row in the database that has that id and if it doesn't have an Id then it will create a new row.

deleteById():
This method serves the purpose of deleting an existing employee from the database, it does this by using the built-in JPA-Repo function **deleteById().**

exists():
This method serves the function of checking if an employee with a given id exists in the database. It does this by using the built-in JPA-Repo function **findById()** and stores the result in an Optional<Employee> variable. It then checks if an employee is present in the variable, if there is then it returns true, if there isn't then it returns false.

EmployeeRepositoryDemoApplication

**/EmployeeRepositoryDemoApplication/*
This is the spring boot application itself, when you run it, it checks all the files that are in its own directory and below. It will read annotations and properly construct the necessary classes.

*src/main/resources/**

Templates

/templates/
The templates directory holds all the html files that the controllers use to display things for the users, the controllers automatically look in this directory when they return a html file unless explicitly told otherwise.

Employees

/employees/
The employees directory is under templates and contains all the html files related to the main functionality of the application they are the following: delete-form.html, employeeform.html, and list-employees.html.

delete-form.html:
**/delete-form.html*
The delete form has the purpose of displaying to the user what employee they're about to delete, it receives the employee's object from the model and displays this in a table with one row, it then warns the user that they're about to permanently delete the employee and provides them with a button that when pressed, send a post request to {domain}/employees/delete with the employee object that the user wishes to delete, it also contains an abort button that when pressed simply redirects the user back to the home page.

employee-form.html
**/employee-form.html*
The employee form is used by both the update and save service; it has a header that changes depending on what the model has as its attribute. It also prefills the form with the information in the object provided by the model, if the user is creating a new employee, then there will be no information in the object so the form will be blank, but if the user is editing an employee, then the form will be prefilled with the already existing information about that user. The form's submit button sends a post request to the {domain}/employees/Save endpoint which takes the object and saves it to the database, the page also contains a button that sends the user back to the home page.

list-employees.html

**/list-employees.html*

This page serves as the main page for the application, it is provided with a list of objects by the model and converts this into a table for the user to see, the page also contains buttons to add, update, and delete employees, but these are hidden from view unless you have the correct roles. The page also features a button that logs the user out.

error-pages

/error-pages/

The error pages are the html files for the custom error pages (like the access denied or couldn't find employee pages). The error pages actually use the same html file, just with different headers and bodies that are provided by the model. The page also features a button that sends the user back to the main menu.

Security

/security/

The security pages are all the html files related to the application's security features, as of now this only includes a custom login page. It contains a form that when filled out and submitted sends the login information to spring security which checks it against the users in the database and if the login information is correct the user is logged in. However, if the login information is wrong the page sees this as an appended parameter named "error" and displays "incorrect username or password". The same also happens if a user is logged out, except that the text displays "successfully logged out" instead.

Application Properties

src/main/resources/application.properties

This file contains the application's properties; they decide how the application operates. The default properties are not displayed in this file and will remain at their default values unless you write them out with custom properties in the file. The following are the properties of our application:

- **server.port=** : This property decides which port the application will be running on. By default, it is 8080.
- **spring.datasource.url=** : This property decides the link to your data source (database)
- **spring.datasource.username=** : The username to your data source
- **spring.datasource.password=** : The password to your data source

Testing

*src/test/java/com.zerolooksgood.demo.EmployeeRepositoryDemo/**
This directory contains the application's test, these are ran to find out if the application behaves as expected, as of now the application only has test for the different mappings and the login process, but not integration testing for the connection between the database and the application.

Database

This application uses PostgreSQL.
The database for this application is split into two parts:

The **Employee Directory** consist of the table **employees** and is where the application stores the data about its employees, the table has four columns that are:

- **id**: Is the identifying number for the employee
- **first_name**: Stores the first name of the employee
- **last_name**: Stores the last name of the employee
- **email**: Stores the email of the employee

The employee stored as Id 1 is John Doe (john.doe@mail.com), this employee is used for testing purposes and is excluded from the table that displays all the employees.

The **Users Directory** consist of the two tables **users** and **authorities**. These two are built up in a very specific way to be compatible with spring security's JDBC users system, the **users** table contains the three columns:

- **username**: Stores the user's username.
- **password**: Stores the user's password.
- **enabled**: Decides whether the user is active or not (1 for active and 0 for disabled)

Whilst the **authorities** table contains the two columns:

- **username**: Store's the username that the role is linked to.
- **Authority**: Stores the username's role (ROLE_{role}

The two **username** columns are linked to each other to connect the users to their roles. If a user has more than one role then these are not written on the same row in the **authorities** table, they are input as separate rows.