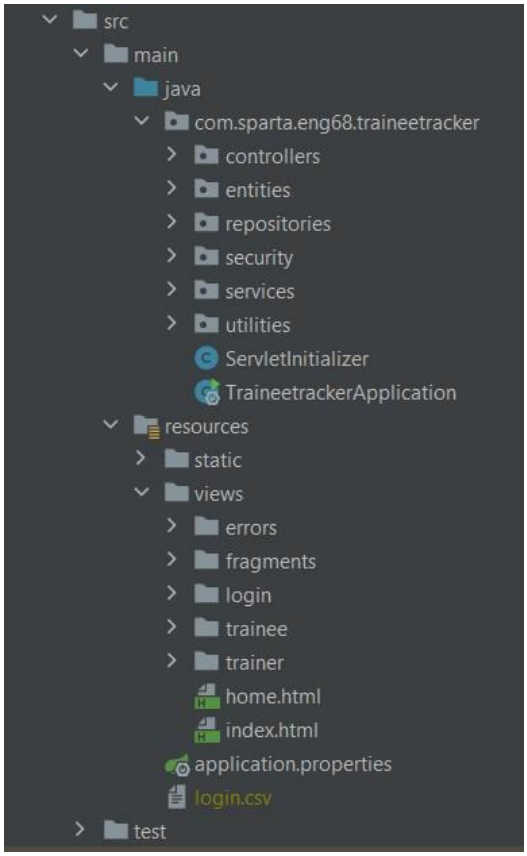
Trainee Tracker Website

Developer Documentation

Contents

Package Structure

Below is an image of the general package structure for this project.



Java Code

For this project, the base package from the *java* folder is *com.sparta*, as is standard convention for our Sparta projects. Then, *eng72*, to signify that we are from class Engineering 72 (you may change this to reflect your own class) and finally *traineetracker*, to reflect what the actual project is about. As such, the base package is *com.sparta.eng72.traineetracker*.

Within this, the project contains the following packages:

- *controllers*
- *entities*
- *repositories*
- *security*
- *services*
- *utilities*

The *controllers* package contains all of the controllers for the project

The *entities* package contains the entities, which are the tables in the database, such as Trainees and Courses

The *repositories* package contains all of the repositories which are abstractions of the database, which allow for easier access to the data. They extend CRUD (Create, Read, Update, Delete) repositories.

The *security* package in particular contains the classes and code pertinent to logins and authentication.

The *services* package contains the services used to access records from the database by certain values, e.g. getting a trainee by their ID or getting a set of trainees by their class ID.

The *utilities* package contains helper classes that aid in making the project work.

HTML Files

The HTML files are located within *resources/views*. From this folder, files are generally organised into packages based on the kind of user who would be accessing those web pages, or the kinds of web pages they are.

The current packages in which the HTML pages are organised are as follows:

- *errors*
- *fragments*
- *login*
- *trainee*
- *trainer*

The *errors* package contains the various error pages that will be shown on the website if a page or requested item cannot be found, or if access to something is denied.

The *fragments* package does not actually contain HTML files that are shown as web pages; they contain HTML scripts that define different components of the other webpages, for example, the headers, footers and navigation bars.

The *login* package contains the web pages related to logging in, authentication and login details.

The *trainee* package contains the web pages directly accessible by trainees. Most of the HTML files in this folder have the "trainee" prefix.

The *trainer* package contains the web pages directly accessible by trainers. Most of the HTML files in this folder have the "trainer" prefix.

Should there be a new type of user who is able to access the website, for example, an admin, convention would be such that you would create a new folder in the *views* folder called "admin".

Branching Strategy

The initial project structure and skeleton should be set up by one person and then pushed to the *dev* branch.

Each time a team member wants to work on a requirement, user story or a feature, they should create a branch from *dev* and implemented it on that branch. The branch should be named according to what is being developed, for example; functionality that allowed trainees to view their weekly reports was developed on a branch called "weekReportPage".

After development and thorough testing, individual branches should make merge requests back to the *dev* branch. Before anyone makes another branch to work on a new feature,

everyone should merge their current branches back to the *dev* branch one by one, so conflicts can be resolved. Once merges have been made, everyone can then pull from the *dev* branch again to work on new functionality and everyone is branching from the same version of the project.

Project Conventions

Package Suffixes

For entities (classes in the *entities* package), the convention is that their classes are simply nouns. When creating a Persistence from the database, the entities being mapped are suffixed with "Entity" by default; for example, "User" would have been "UserEntity" instead. Remove the "Entity" suffix so that the names are as they appear in the database.

However, controllers, repositories and services are all prefixed with their respective package names. Therefore, "UserController", "UserRepository", "UserService", etc. Every entity should have its respective controller, repository and service, as a means of retrieving those entities from the database and displaying them on webpages, but you may also create other ones.

Pages

In the *utilities* package, there Is a class called *Pages.java*. This class is responsible for storing each of the different pages in the project as an individual variable that is more easily accessed. Instead of needing to type out the path of the page in the project structure each time it is needed in a controller, you can statically reference the variable in which it is stored.

For example:

```
    //ERROR ============================================================
public static final String ACCESS_ERROR = "/errors/accessError";
    public static final String PAGE_NOT_FOUND_ERROR = "/errors/pagenotfounderror";
public static final String NO_ITEM_IN_DATABASE_ERROR = "/errors/itemNotFound";
```

Instead of typing "/errors/accessError", which is where the HTML file is in the project structure, you can instead use "Pages.ACCESS_ERROR", which is also more intuitive. If you create new HTML files which will become webpages, it would be beneficial to store it in a static variable and reference the variable instead.

The page variables are separated similarly to how the HTML files are split into packages; they are sectioned off into login, trainee, trainer and error sections. There is also a section for any, which signifies pages that are accessible by either trainees or trainers. Trainee page variables are also prefixed with "TRAINEE" and trainer page variables with "TRAINER" to indicate which type of user has access to those pages In particular.

Connecting to the Database

The application is connected to a database via MySQL. Using MySQL Workbench, our database is stored online, with Amazon Web Services hosting it.

To make sure that the application connects to the database when you run it, you need to define the credentials of the database in the *application.properties* file, shown below.

```
    spring.datasource.url=jdbc:mysql://${MYSQL_HOST:HOST_NAME}:PORT/SCHEMA
                                          ?serverTimezone=GMT
spring.datasource.url
    spring.datasource.username=USERNAME
spring.datasource.password=PASSWORD
spring.jpa.hibernate.ddl-auto=update     spring.jpa.show-
sql=true

    spring.thymeleaf.prefix=classpath:/views/
```

The properties in white hold the following values, which allow them to connect to the database:

- HOST_NAME traineetracker.cqito2bn96lf.eu-west-2.rds.amazonaws.com
- PORT 3306
- SCHEMA training_tracker
- USERNAME [insert your username here]
- PASSWORD [insert your password here]

Please also note that you it is important to append the server time zone to the end of the schema as is shown in the file.

Security

The website is designed such that trainees can only access pages designed for trainees; trainers can only access pages designed for trainers. This is handled through use of the *Roles.java* and *Pages.java* classes in the utilities package.

Password Encryption

User credentials are also stored in the database, in the *user* table. However, each user's password is not stored as plain text in the database. Instead, an encrypted version of the password is stored, adding a layer of account security for the users. Within the *security* package, the class *SecurityConfig.java* contains a method called *passwordEncoder()* which returns a password encoder. To test a password and its encryption, you can use the test class *SecurityConfigTests.java* in the *test* folder and use the method in there to encode the password you want to test.

Roles

Roles.java contains a set of static string variables that define the different roles that a user could assume:

- TRAINER
- TRAINEE
- FIRST_TIME_USER
- ANY (meaning the user can access any page)

Access Page

In addition to all of the static page variables, the *Pages.java* class also contains a static method called *accessPage()* which takes two string arguments:

- *requiredRole* the role the user needs in order to access the page
- *page* the path to the page the user wants to access

The method also returns a string, which determines which page the user will be directed to.

accessPage() gets the role of the user through a HTTP Servlet Request. The method returns the desired page if either of the following is true:

- the user's role is the same as *requiredRole*
- *requiredRole* is ANY (meaning the user can access any page)

If neither condition is met, *accessPage()* instead returns the error page.

The methods in each of the controller classes use the *accessPage()* method to ensure that the role of the user is checked before access is granted to them. In the example below, the *TRAINER_NEW_USER_PAGE* page is only accessible by users with the *TRAINER role*.

```
    @GetMapping("/newUser")
    public String newUserForm(Model model) {
        model.addAttribute("newUserForm", new NewUserForm());
        model.addAttribute("allGroups",    courseGroupService.getAllCourseGroups());
model.addAttribute("allTrainees", traineeService.getAllTrainees());          return
Pages.accessPage(Role.TRAINER, Pages.TRAINER_NEW_USER_PAGE);     }
```