DALHOUSIE UNIVERSITY

Faculty of Computer Science

Dalhousie University

CSCI 5308
Quality Assurance

Project Report
Job Portal - MyCareer

Submitted By: (Group 8)

Abinaya Raja - B00799562 (abinaya.raja@dal.ca)
Aniruddha Chitley - B00808320 (aniruddha.chitley@dal.ca)
Jessica Castelino- B00804805 (jessica.castelino@dal.ca)
Sarmad Noor-B00799557 (sarmad.noor@dal.ca)

Introduction

We have developed a job portal known as MyCareer for the university students who are looking for full time as well as co-op job opportunities. The target user base for our application are the graduate Computer Science students studying at Dalhousie University. There are three user personas involved in this project – Co-op coordinators, employers and students. The functionalities of each of these user personas is mentioned below.

Co-op administrators:

1. Co-op administrators can login to the website.
2. Authenticated co-op administrators can perform the following tasks:
    i. Approve or decline registration requests from employers after background check
    ii. Add new students to give them access the portal
    iii. Revoke the access of students to the portal
    iv. Delete the employer to revoke their access from the portal

Employers:

1. The employers can register themselves on the website
2. The employers who have received an approval from co-op administrators can login to the portal and perform the below tasks:
    i. Post new jobs
    ii. View jobs posted in the past
    iii. Close a job posting that is already created, temporarily close a posted job, and permanently delete the job posting
    iv. Edit an existing job posting
    v. View the details of the students who have applied for the job and access the application package (resume, cover letter, etc)
    vi. Change the status of the applicant (under review, accepted, rejected, etc)
    vii. Filter applicants based on the application status
    viii. Manage the company profile information and personal information

Students:

1. Students can login to the website
2. Authenticated students can perform the following tasks:
    i. View the job postings posted by employers
    ii. View the jobs they have applied for in the past along with the status
    iii. Apply for jobs and upload relevant documents
    iv. Filter jobs based on location and co-op work term

Tools used

- Eclipse, VSCode – IDE for development
- MySQL workbench – connecting to MSQL remote database
- PUTTY – to connect to Azure remote server.
- GitBash

Client-Side Technology:

- HTML
- CSS
- JQuery
- JSP

Server-Side Technology:

- Java

Framework:

- Spring Boot

Database:

- MySQL

Source Code Control:

- Github

Continuous Integration Tool:

- Jenkins

Ticket Controlling Tool:

- Trello

Cloud Server Used:

- Heroku
- Azure (for demonstration of server logs)

Continuous Integration Implementation

Jenkins is used for continuous integration and deployment on Heroku cloud.

The build and deployment steps are not hardcoded, but they are configured in Jenkins configuration console.

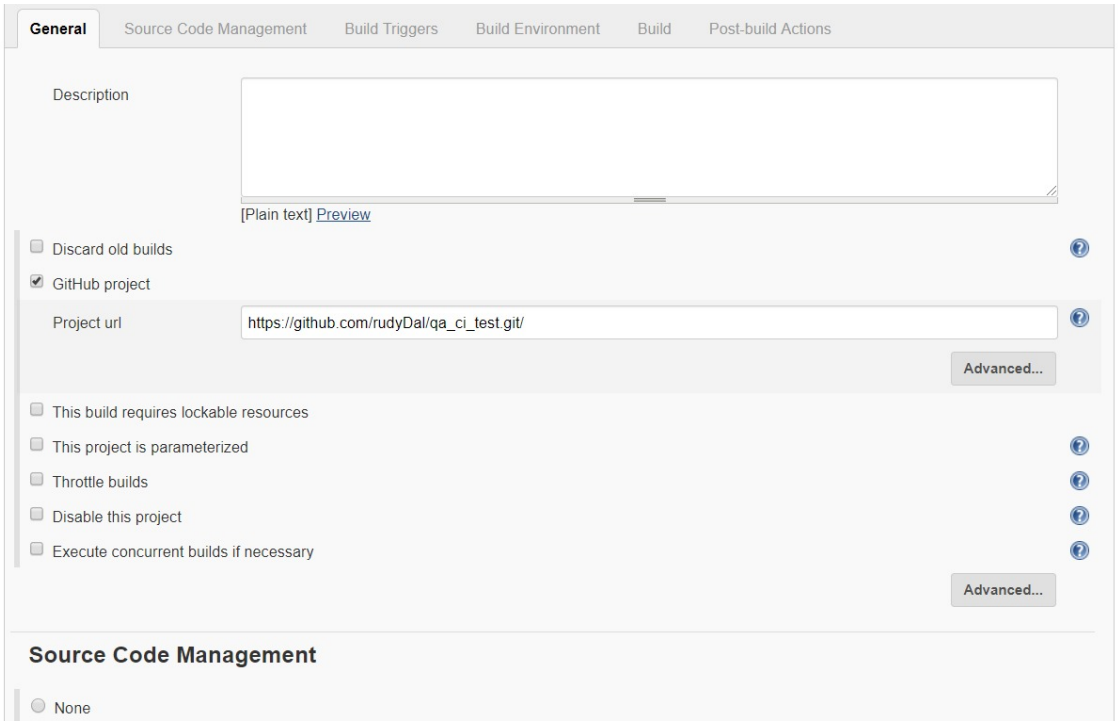

Figure 1: Git repo URL to fetch the code

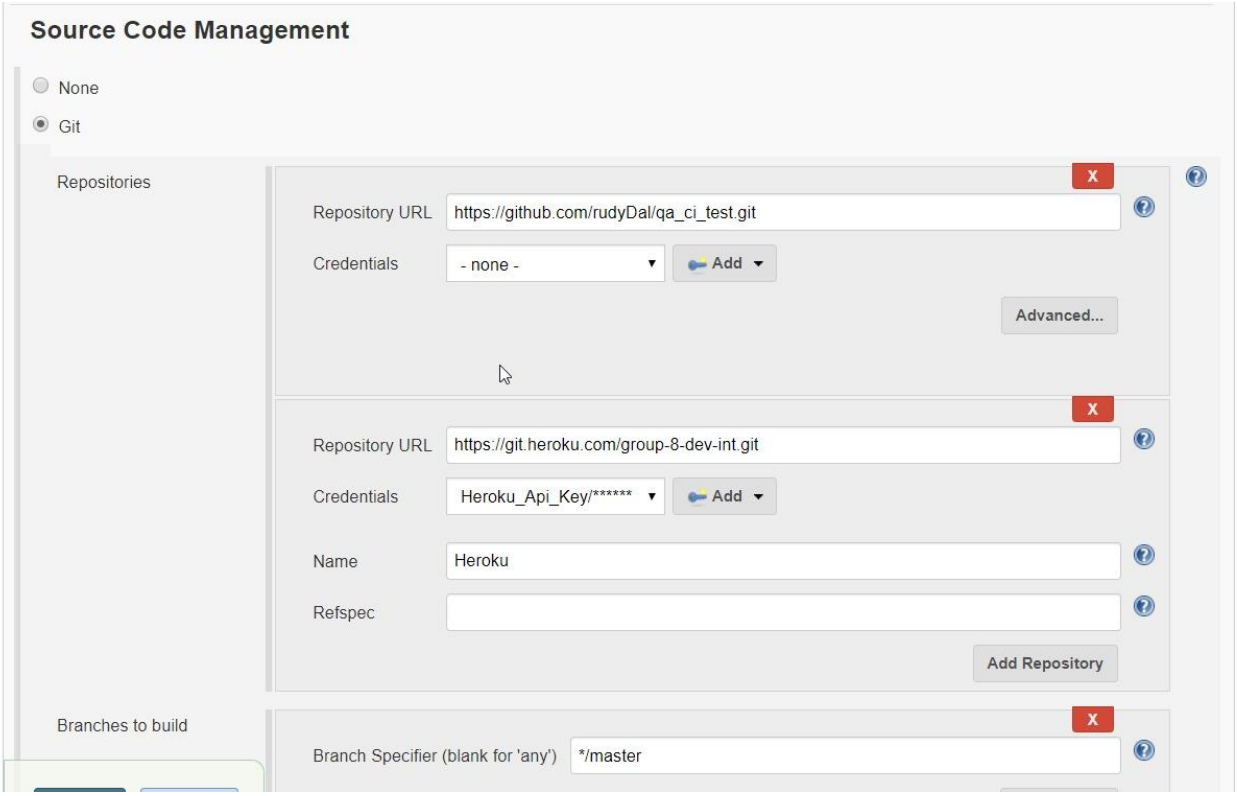

Figure 2: Code repository and cloud URL

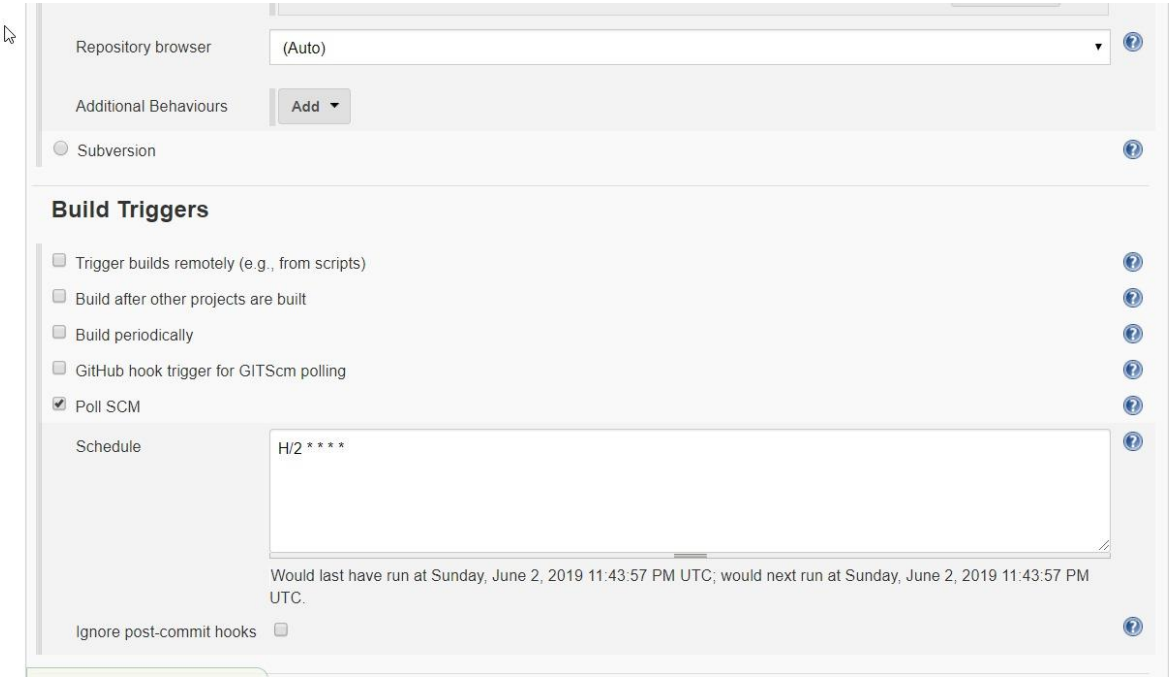

Figure 3: Build Triggers

Configuration based Logic

Use case 1: Showing jobs to students based on pre-requisite courses for the job matching the student's completed courses.

Use case 2: Showing all the courses to the student posted by employers on My Career portal.

myCareer.properties

```
Case 1
## Configuration for showing jobs to students.
job_type=prerequisite

Case 2
## Configuration for showing jobs to students.
job_type=all
```

Figure 4: Configuration values

Java Code:

Figure 5: Folder structure for configurable business logic

FetchAllJobs.java is the class corresponding to the configuration "all" and **FetchPrequisiteCourseJobs.java** is the class corresponding to the configuration "prerequisite". In the **ConfigLogicClassLoader.java,** a Hash map is defined which consists of configuration and corresponding class mapping. Now based on the value of the "job_type" property in **mycareer.properties** file, the object of relevant class is fetched from the HashMap and correct procedure is returned.

Figure 6: Configuration Loader

Design Patterns

- ✓ **Template method pattern**

  Template method pattern is used to reuse the code for creating connection, preparing call statement, and execution of stored procedure; meanwhile facilitating the type of operations to be performed and ways to handle the parameters and results to be decided at runtime. We choose to use template method in this case because it avoids repeating of code wherever data is to be stored to or retrieved from the database. It facilitates a centralized class where the common activities can be performed, thereby reducing the number of files/classes/methods to be changed for any reason.

```
public abstract class JdbcManager
{
    private Connection con = null;
    private CallableStatement callStatement = null;
    private Map<String, Integer> procResults;
    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    public Map<String, Integer> executeProcedure(String procedureName, String mapperObjectName, Object dto
    {
        try
        {
            createDBConnection();
            prepareProcedureCall(procedureName);
            procResults = executeProc(callStatement, mapperObjectName, dtoObject, additionalParam);
        }
        catch(Exception ex)
        {
            logger.error("Error occurred while executing the statement " + procedureName, ex.getMessage())
        }
        finally
        {
            DatabaseConnection.closeConnection(con);
        }
        return procResults;
    }
```

Figure 7: Template method design pattern

JdbcManager abstract class contains the “executeProcedure” that contains common tasks to be performed while executing the store procedure.
We created 4 classes InsertHandler, SelectHandler, UpdateHandler, and DeleteHandler to perform the operations insert, select, update, and delete respectively. These classes have their own way of performing the operations it is supposed to do.

✓ **Singleton pattern**

```
package com.dal.mycareer.propertiesparser;

import java.io.FileNotFoundException;

public class PropertiesParser {

    private static final ClassLoader CLASS_LOADER = MethodHandles.lookup().lookupClass().getClassLoader();
    static Logger LOGGER = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
    private static Properties prop;
    private static InputStream inputStream;

    static {

        try {
            prop = new Properties();
            String propFileName = "mycareer.properties";

            inputStream = CLASS_LOADER.getResourceAsStream(propFileName);

            if (inputStream != null) {
                prop.load(inputStream);
            } else {
                throw new FileNotFoundException("property file '" + propFileName + "' not found in the classpath");
            }

        } catch (IOException ex) {
            LOGGER.error("Error loading properties file : " + ex.getMessage());
        } finally {
            try {
                inputStream.close();
            } catch (IOException e) {
                LOGGER.error("Error closing input stream : " + e.getMessage());
            }
        }
    }

    public static Properties getPropertyMap() {
        return prop;
    }

}
```

Method to get the singleton instance of Properties map

Figure 8: Singleton design pattern in PropertiesParser.java class

In the above case the when the classes loader loads the **PropertiesParser.class**, a Properties map of **mycareer.properties** file is created and this singleton instance is used by other classes reading from the properties map.

```java
package com.dal.mycareer.emailengine;

import java.lang.invoke.MethodHandles;

public class SendGridSingleton {

    private static SendGrid instance;
    static Logger logger = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

    // Returns singleton instance of SendGrid
    public static SendGrid Instance() {

        logger.debug("SendGrid instance : START");

        if (null == instance) {
            Properties propertyMap = PropertiesParser.getPropertyMap();
            instance = new SendGrid(propertyMap.get("sendgrid.api.key").toString());
        }
        logger.debug("SendGrid instance : END");
        return instance;
    }

}
```

Figure 9: Singleton design pattern in SendGrid class

The Instance() method returns a singleton instance of third-party email service (SendGrid) which is used by other methods to send email to Students and Employers after successful onboarding.

Separation of Layers

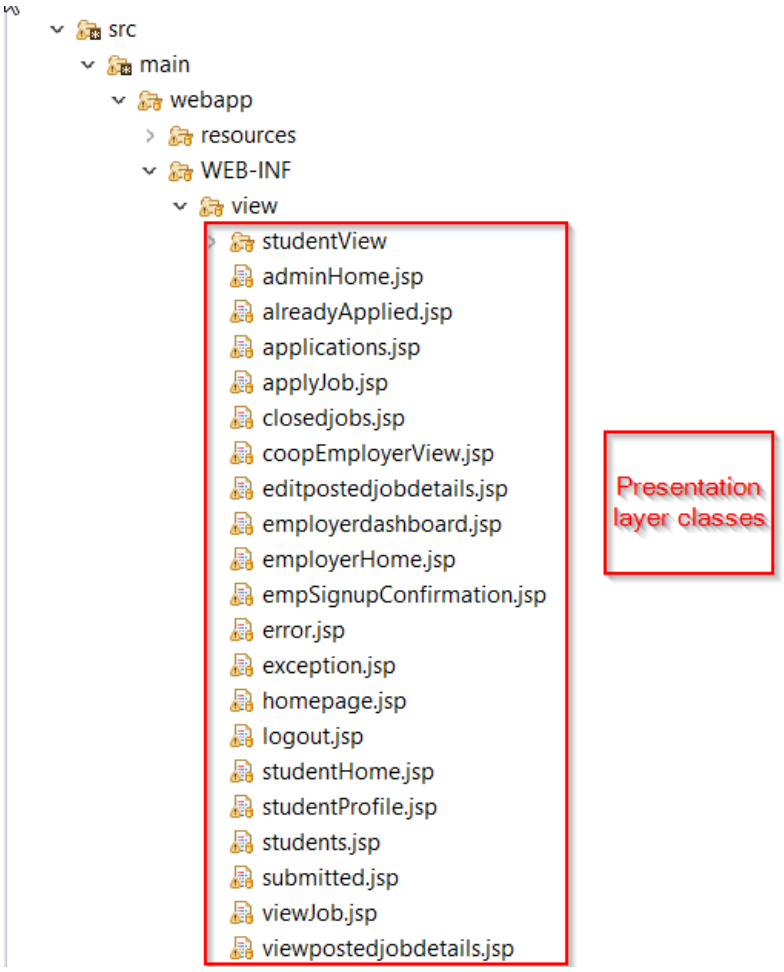

Figure 10: Presentation Layer

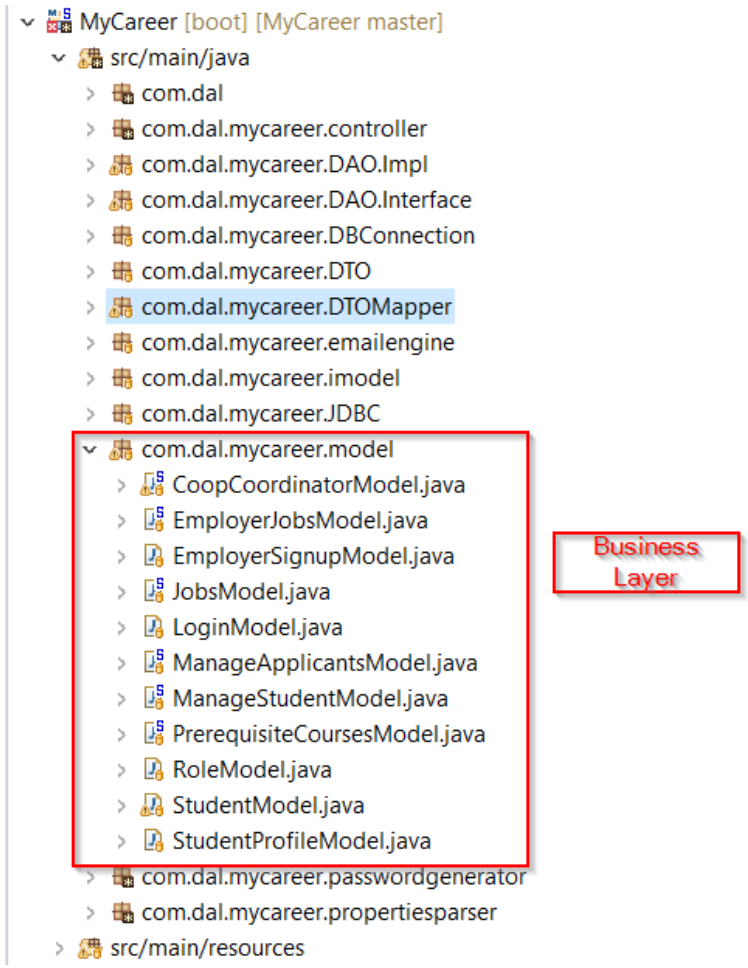

Figure 11: Business Layer

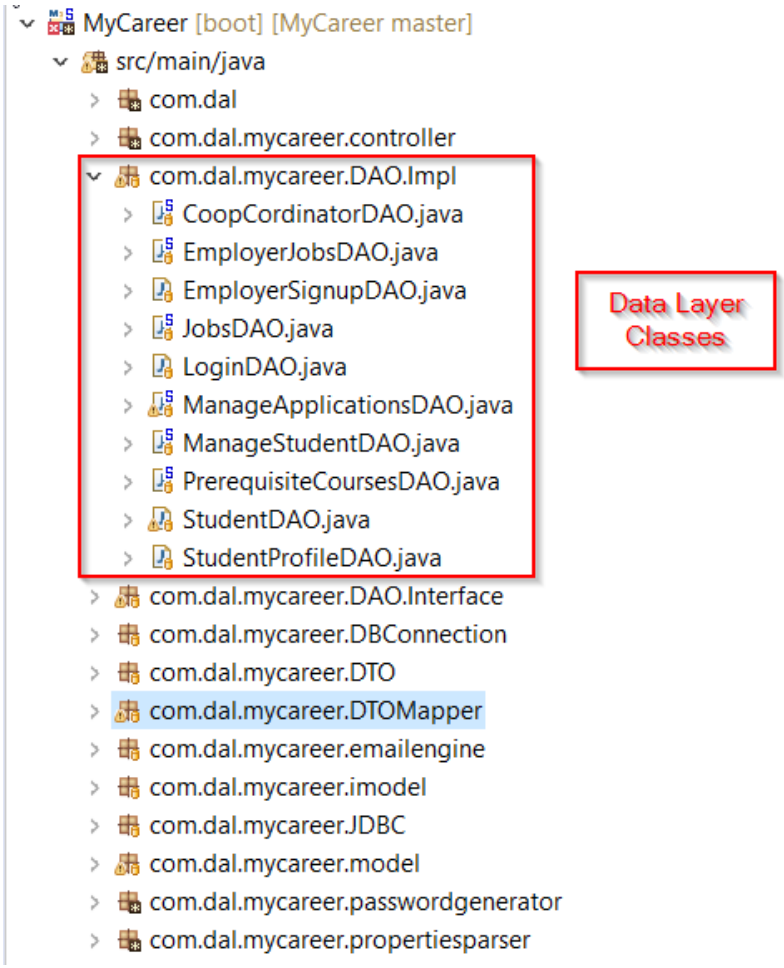

Figure 12: Data Layer classes

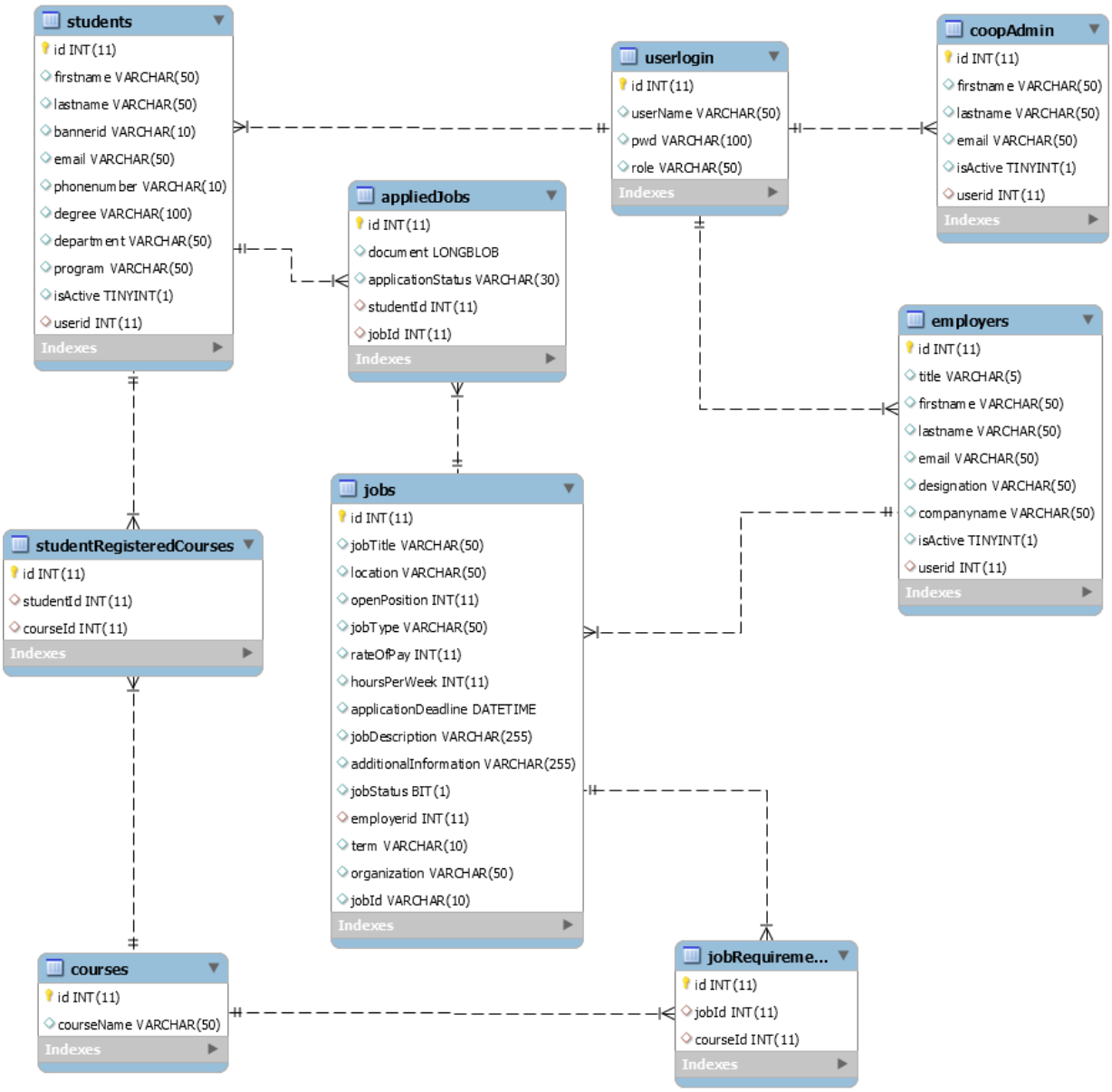

Figure 13: Database UML diagram

Unit test case overview

We have 36 test cases inside the hierarchy **src/test/java**. The package for each test class is same as that of the Java class for that test.

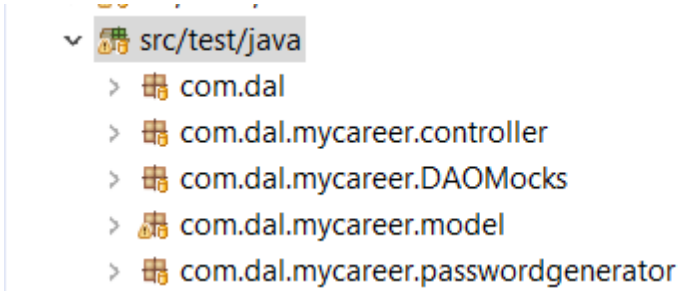

Figure 14: JUnit test classes hierarchy

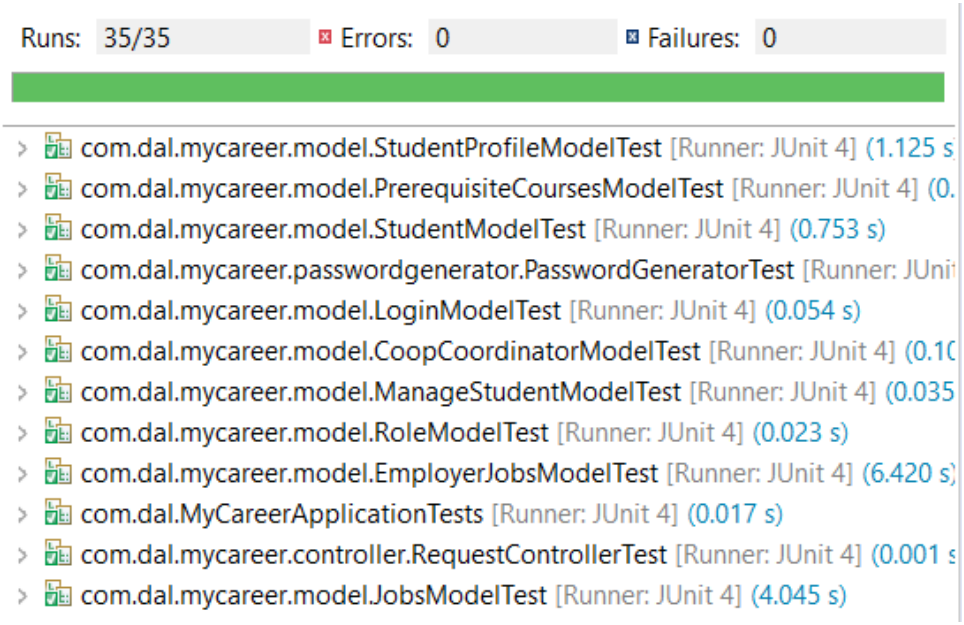

Figure 15: Test case execution report

Naming/Spacing Convention

We have agreed to use camel case for method names and variable names and Pascal case for class names. We agreed to have a single line of space between each method.

Refactoring performed

- Reduced the duplicate code for executing stored procedures by implementing template method pattern.
- Created class level constants for frequently used string in the class rather than hardcoding the string everywhere in the class.
- Created separate classes for various models and moved the methods to these classes. In order to avoid Single responsibility principle violation by any model classes.
- Moved the common fragment of code outside the conditional fragments.
- Extracted variables from complicated expressions in a temporary variable that explains its purpose.
- Grouped parameters and introduced a parameter object wherever possible.
- Replaced error code with exceptions that are handled by a Global Exception Handler.
- Created getting and setting methods to access fields of an object so that coupling can be removed.
- Split temporary variables to ensure that every assignment a separate variable exists, unless it is not a loop variable.

Technical debt

- ✓ We almost completed the entire project when we learnt about the design patterns. The T**emplate method design** pattern we implemented in the project requires complete rewrite of data layer code. Since we did not get enough time to consume the classes that we introduced to perform database operations throughout the project, we have completed the refactoring of the data layer code in 50% of the data layer classes and demonstrated the implementation of Template method design pattern in **Employer JobDAO.java.** Due to time crunch we could not implement this Design pattern in other DAO classes.
  We have generalized the classes (**InsertHandler.java, SelectHandler.java, UpdateHandler.java, and DeleteHandler.java**) and methods that we introduced to handle the database operations to support the execution of stored procedures that perform any DML operations. Hence, the refactoring of the rest of the data layer classes can be completed easily by consuming these classes.

Member's Contribution

Member Name	**List of classes**	**Methods**
Abinaya Raja	EmployerJobsController.java	activeJobs
		getActiveJobs
		getClosedJobs
	JobsController.java	closeJob
		openJob
	ManageApplicationsController.java	viewApplicants
		updateApplicationStatus
	ManageStudentController.java	RegisterStudent

	EmployerJobsDAO.java	getActiveJobs
		getClosedJobs
		fetchJobByStatus
	JobsDAO.java	updateJobStatus
	ManageApplicationsDAO.java	getApplications
		updateApplicationStatus
	ManageStudentDAO.java	RegisterStudent
	PrerequisiteCoursesDAO.java	addStudentCompletedPrereq
	JobsMapper.java	
	StudentDetailsMapper.java	
	StudentsMapper.java	
	DeleteHandler.java	
	ProcedureParamLoader.java	
	SelectHandler.java	
	UpdateHandler.java	
	EmployerJobsModel.java	getActiveJobs
		getClosedJobs
	JobsModel.java	updateJobStatus
	ManageApplicationsModel.java	getApplications
		updateApplicationStatus
	ManageStudentModel.java	RegisterStudent
	EmployerJobsDAOMock	getActiveJobs
		getClosedJobs
	JobsDAOMock.java	
	ManageStudentDAOMock.java	setLstRegisteredStudent
		getLstRegisteredStudent
		RegisterStudent
	PrerequisiteCoursesDAOMock.java	
Aniruddha Chitley	PropertiesParser.java	
	ConfigLogicClassLoader.java	
	FetchAllJobs.java	
	FetchJobs.java	
	FetchPrerequisiteCourseJobs.java	
	StudentProfileModel.java	
	RoleModel.java	
	LoginModel.java	
	PasswordGenerator.java	
	EmployerApprovalEmail.java	
	EmployerRejectionEmailImpl.java	
	SendGridSingleton.java	
	SendGridEmailService.java	SendHTML()
		SendEmail()
	StudentProfileController.java	

	LoginController.java	Login()
	RequestController.java	LoadHome()
		EmployerHome()
		Logout()
	EmployerSignUpConteroller.java	
	LoginDAO.java	
	StudentProfileDAO.java	
	DatabaseConnection.java	GetConnection()
		CloseConnection()
		CloseDatabaseComponents() - overloaded method
Jessica Castelino	RecruiterRequestsController	
	GlobalExceptionController	
	StudentJobsController	
	StudentJobApplicationController	
	ManageApplicationsController	downloadResume()
	JobsController	viewStudentSelectedJob
	RecruiterRegistrationRequestDAO	
	ManageApplicationsDAO	fetchDocument
	StudentApplicationDAO	
	StudentDetailsDAO	
	StudentJobsDAO	
	JobsDAO	fetchJob
	ManageApplicationsModel	downloadFile
	RecruiterRegistrationRequestModel	
	JobsModel	fetchJob
	StudentApplicationModel	
	StudentJobsModel	
Sarmad	CoopCoordinatorController.java	showActiveRecruiter
		deleteActiveEmployer
	EmployerJobsController.java	saveJob
		viewPostedJob
		editPostedJob
		updateJobDetails
	ManageStudentController.java	getRegisteredStudents
	PrerequisiteCoursesController	Full class
	CoopCordinatorDAO.java	Full class
	EmployerJobsDAO.java	InsertJobDetails
		viewPostedJobDetails
		updatejobDetails
	ManageStudentDAO.java	getRegisteredStudents
		isNewStudent
	PrerequisiteCoursesDAO.java	insertJobPrerequisiteCourses
		getPrerequisiteCourses
	DatabaseConnection.java	closeDatabaseComponents
	DTOMapper.java	

	JobDetailsMapper.java	
	JdbcManager.java	
	InsertHandler.java	
	CoopCoordinatorModel.java	
	EmployerJobsModel.java	InsertJobDetails
		viewPostedJobDetails
		updateJobDetails
	ManageStudentModel.java	getRegisteredStudents
	PrerequisiteCoursesModel.java	
	CoopCordinatorDAOMock.java	fetchActiveRecruiters
		deleteActiveRecruiter
	EmployerJobsDAOMock.java	InsertJobDetails
		viewPostedJobDetails
		updatejobDetails
	ManageStudentDAOMock.java	getRegisteredStudents

Member Name	**List of Presentation layer**
Abinaya Raja	employerHome.jsp (job list base page)
	closedjobs.jsp
	applications.jsp
	students.jsp (Register student popup)
Aniruddha Chitley	homepage.jsp
	logout.jsp
	error.jsp
	StudentProfile.jsp
Jessica	adminHome.jsp
	studentHome.jsp
	applyJob.jsp
	alreadyApplied.jsp
	Exception.jsp
	submitted.jsp
	viewJob.jsp
Sarmad	employerHome.jsp (create job pop up)
	editpostedjobdetails.jsp
	viewpostedjobdetails.jsp
	coopEmployerView (active employer)
	students.jsp (Registered students list base page)

Member Name	List of Stored procedures and SQL Functions
Abinaya Raja	insertStudentCompletedCourses
	sp_insertStudent
	sp_updateApplicationStatus
	getActiveJobsForEmployer
	getClosedJobsForEmployer
	sp_updateJobStatus
	sp_deleteStudent
	getRequiredCoursesForJob (function)
	getCompletedCoursesForStudent (function)
Aniruddha Chitley	IsValidLogin
	studProfile
Jessica	fetchJob
	fetchDocument
	alreadyApplied
	withdrawApplication
	applyForJob
	fetchStudentDetails
	getAppliedJobList
	getAllJobList
Sarmad	checkDupicateStudent
	fetchActiveRecruiters
	fetchRegisteredStudents
	getPostedJobDetails
	updatejobdetails
	sp_insertjobdetails
	sp_deleteActiveEmployer
	sp_getPrerequisiteCourses
	insertjobRequirementRecord
	getCompletedCoursesIds (function)

Other Project related Tasks:

- Aniruddha
    - CI/CD setup with Jenkins and Heroku.
    - Logging Implementation.
- Jessica
    - CI/CD setup with Jenkins and Heroku.
    - Deployment on Azure as logs were not visible on Heroku
- Sarmad
    - Database setup
- Abinaya
    - Database setup