Group15 – CSCI 5308 Quality Assurance

Nibir Mukherjee
B00768437
nibir.mukherjee@dal.ca

Rohit Gollarahalli
B00779758
rohit.gollarahalli@dal.ca

PRODUCT:

Name: FindMyEmployer

Technology Used: Python Flask, HTML5/CSS3, JavaScript/JQuery, Bootstrap, Jinja template, MySQL

URL: https://qagroup15-findmyemployer.herokuapp.com/

Description: Our application is a replica of the LinkedIn website. We have tried to mock the functionalities as much as possible. There are a lot of job search engines out there in the market, but comparing the job competition, it is better to have as many such websites as possible. Our application has the following functionalities:

- Login/Sign Up
- Profile Homepage View
- Uploading profile pictures
- Posting status and live status feed on profile page
- Updating profile page/Updating user details
- Changing password
- Sending message to users
- Viewing Jobs and Adding Jobs for employers/ Viewing Jobs and Applying to Jobs for employees
- Business Logic: i) Job applying/posting limitation based on plans
  ii) Message functionality enabled/disabled based on plans

CONTINUOUS INTEGRATION

At the very start of the project, we set up our continuous integration environment via GitHub->Jenkins->Microsoft Azure. We were new to the Jenkins tool and had a bit of hurdles in setting up the software. But eventually we were able to set up the environment. For Azure, we selected free trial account and create a webapp from Azure CLI. The problem that we faced from Azure was that, we had limitations as it was a free trial. New accounts needed to be created every month and that was a roadblock as we are a group of only two people. Moreover, Azure only allows a data-quota of 165MB and we were exceeding the amount after a certain period of time. It takes 24 hours to reset the quota. So, facing all these problems, we planned to shift to Heroku.

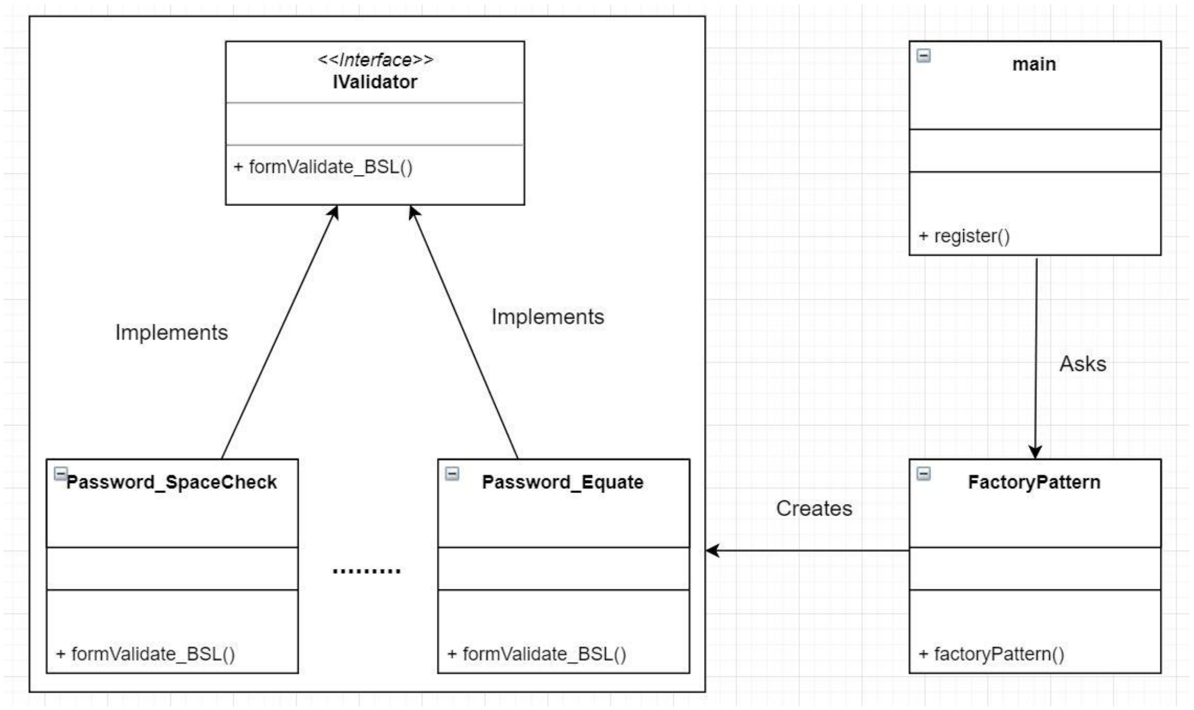
Deploying flak to Heroku was very straight forward. Heroku is free and has a higher limit of data-quota. So, using it was smoother than Azure. Heroku has the feature of directly deploying from GitHub. So, we don't need Jenkins for deploying to Heroku. But, we were using a private repository, so we had to seek the admin rights from our coordinator before we can hook our Heroku server to our GitHub repository. Moreover, we didn't need any hardcoded configuration for implementing the continuous integration. So, finally, for the production environment we have chosen,

GitHub->Heroku

DESIGN PATTERNS

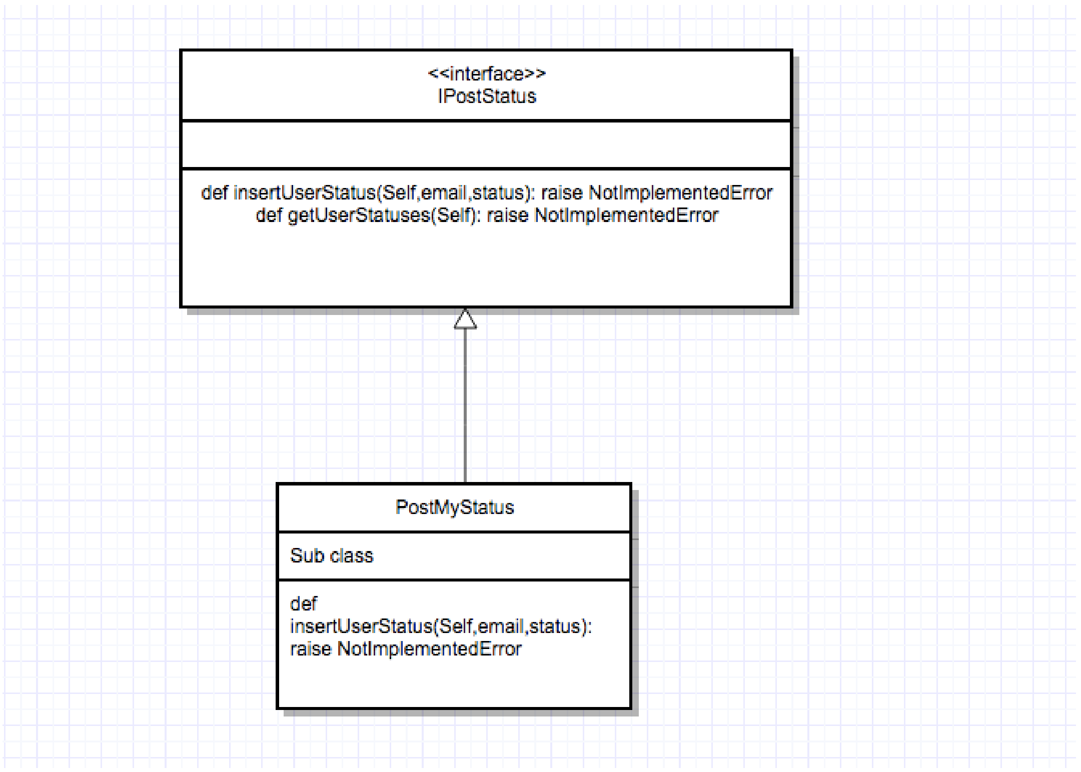
For our project, we have implemented four design patterns:-

- Singleton Pattern – For our mysql object creation and the User model class we have implemented the singleton pattern under extensions where the object once created was used throughout the application.
- Factory Pattern – We have implemented factory pattern for validating the forms. We saw that same methods were needed multiple times for validating multiple forms. So, we created an IValidator interface having an abstract validator method. We then wrote different business layer classes implementing that interface and described the body of the methods accordingly. Then we created a FactoryPatten class which builds all these implemented classes based on certain conditions. Finally, we pass conditions from main method to the FactoryPattern class and implement the validations. Below diagram shows the UML of our Factory Pattern.



Factory Pattern

- Template Pattern – The template pattern is the behavioral design pattern where our program algorithm is initially defined. Here the interface IPostStatus defines the two methods which are later implemented by 2 other subclasses.



<<interface>>
IPostStatus
def insertUserStatus(Self,email,status): raise NotImplementedError
def getUserStatuses(Self): raise NotImplementedError
PostMyStatus
Sub class
def insertUserStatus(Self,email,status): raise NotImplementedError

Template Pattern

Decorator Pattern - Employees and Employers can have different plans to access the application. The basic plan does not allow the users to message and have a apply count of 6 while the premium account allows the users to contact the employers/employee and have an apply count for jobs and posting ads upto 30. The Decorator pattern is used to extend the functionality or in some cases add a new functionality. The decorator class generally adds a component as a field. For our scenario, we have:

For Employer:
Component interface/Abstract class:
Employer

Concrete component:
NormalEmployer

Decorator classes:
EmployerDecorator

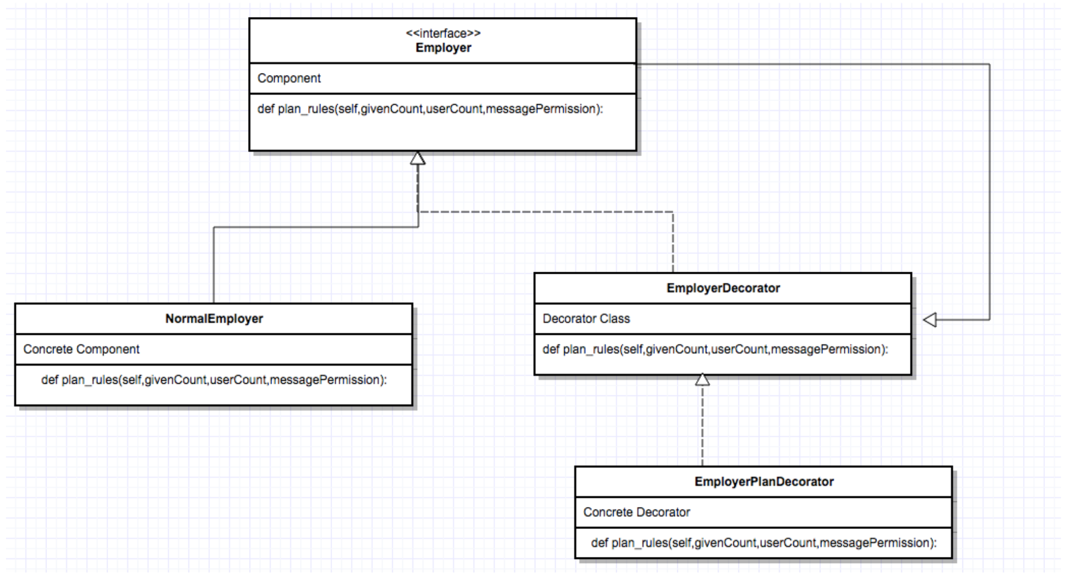
Concrete decorator:
Employer_Plan_decorator

For Employee:
Component interface/Abstract class:
Employee

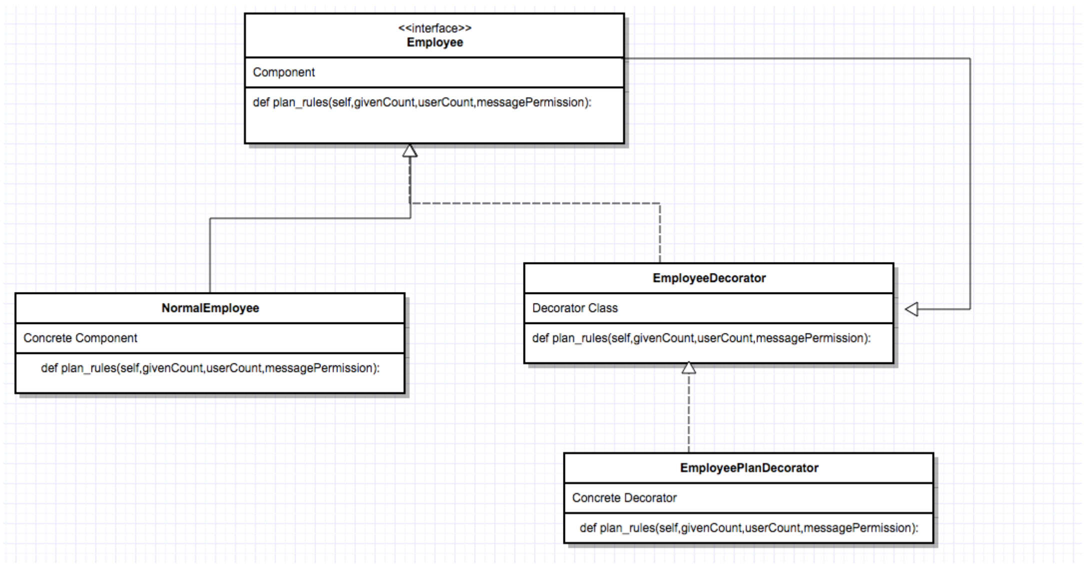
Concrete component:
NormalEmployee

Decorator classes are:
EmployeeDecorator

Concrete decorators:
Employee_Plan_decorator



<<interface>>
Employer
Component
def plan_rules(self,givenCount,userCount,messagePermission):
EmployerDecorator
Decorator Class
def plan_rules(self,givenCount,userCount,messagePermission):
NormalEmployer
Concrete Component
def plan_rules(self,givenCount,userCount,messagePermission):
EmployerPlanDecorator
Concrete Decorator
def plan_rules(self,givenCount,userCount,messagePermission):



<<interface>>
Employee
Component
def plan_rules(self,givenCount,userCount,messagePermission):
EmployeeDecorator
Decorator Class
def plan_rules(self,givenCount,userCount,messagePermission):
NormalEmployee
Concrete Component
def plan_rules(self,givenCount,userCount,messagePermission):
EmployeePlanDecorator
Concrete Decorator
def plan_rules(self,givenCount,userCount,messagePermission):

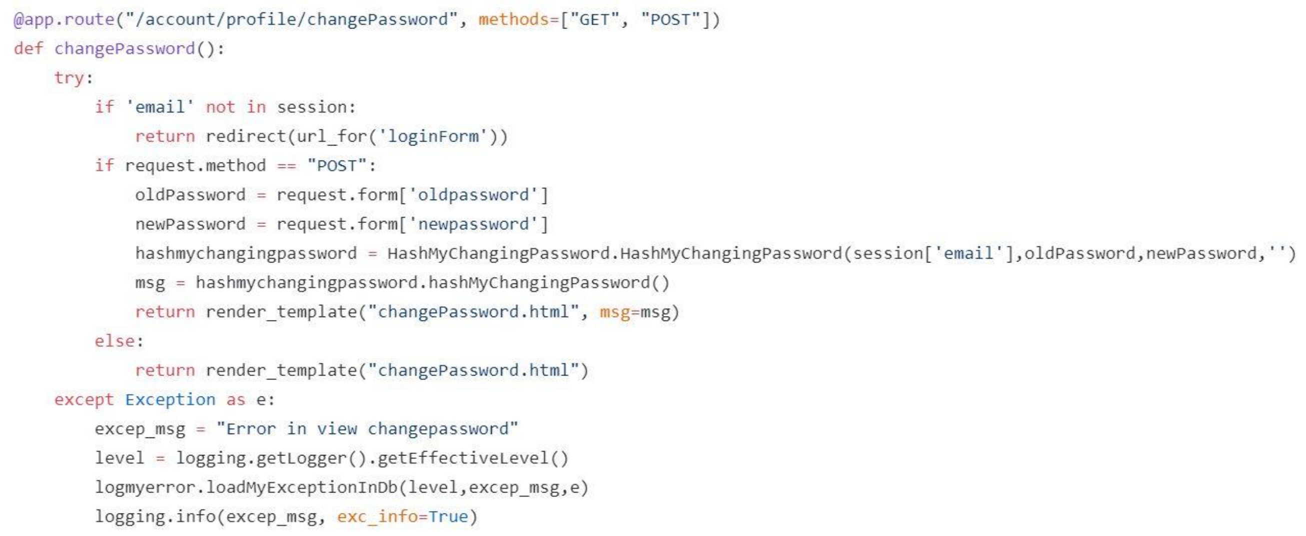
Template Pattern

3-LAYER ARCHITECTURE

For the ease of code maintenance and following industry standards, we have followed 3-layer architecture while building our application. Our application comprises of Presentation Layer, Business Layer and Data Layer. Before writing classes for different layers, we went through the concepts as discussed in class, as to how to segregate our classes into three different layers.

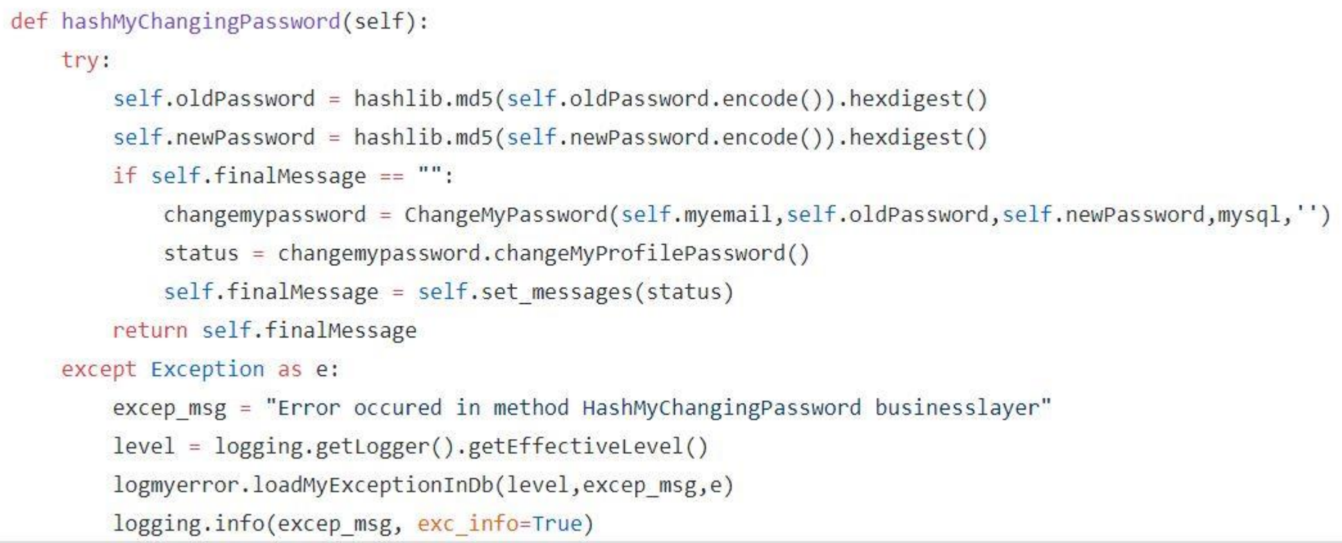
- Presentation Layer – This is the topmost layer which deals with the user interface. Our main.py class represents the presentation layer. In this class we have written different methods which routes the user to different functionalities of the application. There are various forms in our application. For all the POST methods, we have captured the form-data in this layer.
- Business Layer – This is the layer that performs the business logic operations and calculations. We have created a folder called Businesslayer and stored all the Business layer classes there. These classes act as an intermediate between the Presentation Layer and the Data Layer. We wrote separate classes for separate functionalities of the application.
- Data Layer – This is the layer where we do the database operations. Storing to the database or fetching data from the database is done in this layer. We have created a folder called Databaselayer and stored all the Database layer classes there. These classes either store or retrieve data from the database and sends them to the Business Layer.

To relate the architecture in our application, let us consider one module for example. Say, Changing Password module. Below are the method screenshots which demonstrates the connectivity between three layers and the functionalities each layer implementing.



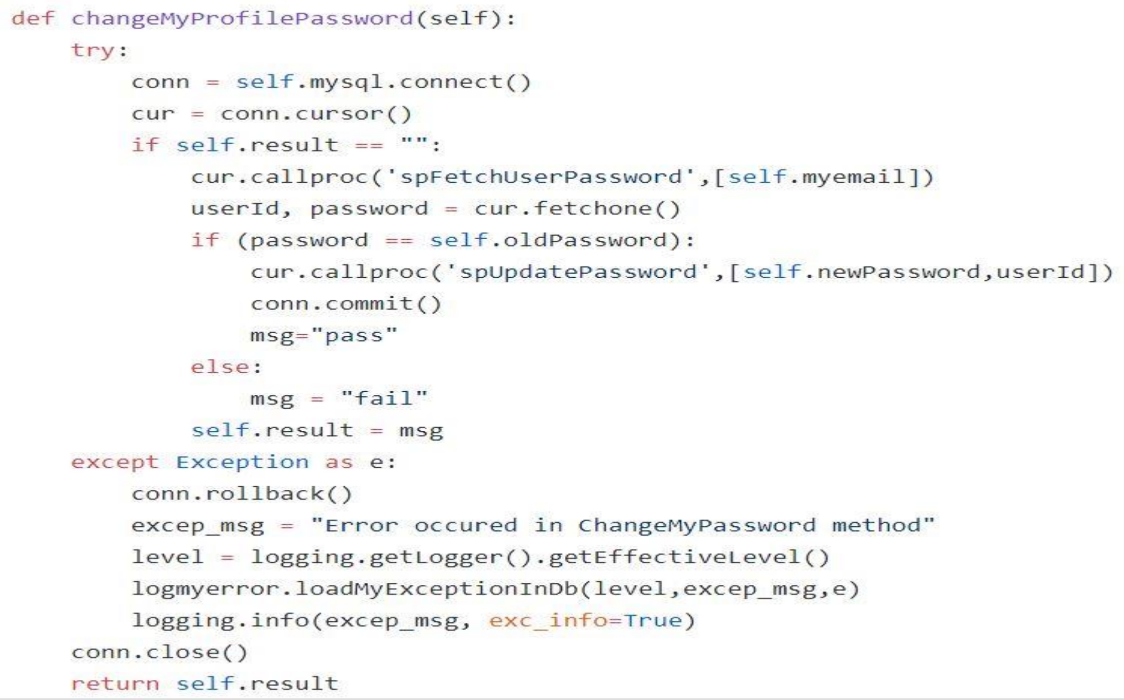
```
@app.route("/account/profile/changePassword", methods=["GET", "POST"])
def changePassword():
    try:
        if 'email' not in session:
            return redirect(url_for('loginForm'))
        if request.method == "POST":
            oldPassword = request.form['oldpassword']
            newPassword = request.form['newpassword']
            hashmychangingpassword = HashMyChangingPassword.HashMyChangingPassword(session['email'],oldPassword,newPassword,'')
            msg = hashmychangingpassword.hashMyChangingPassword()
            return render_template("changePassword.html", msg=msg)
        else:
            return render_template("changePassword.html")
    except Exception as e:
        excep_msg = "Error in view changepassword"
        level = logging.getLogger().getEffectiveLevel()
        logmyerror.loadMyExceptionInDb(level,excep_msg,e)
        logging.info(excep_msg, exc_info=True)
```

Presentation Layer



```
def hashMyChangingPassword(self):
    try:
        self.oldPassword = hashlib.md5(self.oldPassword.encode()).hexdigest()
        self.newPassword = hashlib.md5(self.newPassword.encode()).hexdigest()
        if self.finalMessage == "":
            changemypassword = ChangeMyPassword(self.myemail,self.oldPassword,self.newPassword,mysql,'')
            status = changemypassword.changeMyProfilePassword()
            self.finalMessage = self.set_messages(status)
        return self.finalMessage
    except Exception as e:
        excep_msg = "Error occured in method HashMyChangingPassword businesslayer"
        level = logging.getLogger().getEffectiveLevel()
        logmyerror.loadMyExceptionInDb(level,excep_msg,e)
        logging.info(excep_msg, exc_info=True)
```

Business Layer



```
def changeMyProfilePassword(self):
    try:
        conn = self.mysql.connect()
        cur = conn.cursor()
        if self.result == "":
            cur.callproc('spFetchUserPassword',[self.myemail])
            userId, password = cur.fetchone()
            if (password == self.oldPassword):
                cur.callproc('spUpdatePassword',[self.newPassword,userId])
                conn.commit()
                msg="pass"
            else:
                msg = "fail"
            self.result = msg
    except Exception as e:
        conn.rollback()
        excep_msg = "Error occured in ChangeMyPassword method"
        level = logging.getLogger().getEffectiveLevel()
        logmyerror.loadMyExceptionInDb(level,excep_msg,e)
        logging.info(excep_msg, exc_info=True)
    conn.close()
    return self.result
```

Data Layer

From the above code snippets, we can relate to the flow of the program as discussed in the 3-layer architecture. The view layer obtains the form data from the HTML page and passes them as parameters to the Business layer function. The Business layer hashes the password and sends the password to the Database layer. In the Database layer, we see that the hashed password is getting stored in the database through the stored procedure, spUpdatePassword.

NAMING/SPACING CONVENTION

For our development of the application we have used Python. Indentation in Python is a part of the syntax. If the program is not properly indented, it will not compile. So for Python, we didn't have to deal with indentation separately. For whitespaces, we decided to leave one-line gap between two methods. This logically separates the method functionalities and adds to the programming aesthetics. Naming convention is something that will help in the maintenance of the codebase, as well as in further refactoring. We gave meaningful names for all the elements: variable, functions, classes, etc. One can easily understand the purpose of the element just by seeing the names. Moreover, we have followed camelCasing in our naming convention.

REFACTORING

Our functionality development ended a week before our project submission. So, after that we started refactoring our code base. We checked on naming conventions and removed unused files. Some refactoring techniques that we followed are:

- We refactored our code by having model classes
- We followed OOPS concepts and had the class constructor object populated.
- We sent fewer parameters in methods through list
- Bigger methods were broken down to smaller chunks to follow the single design principle.

TECHNICAL DEBT

We implemented all the features and functionalities that we proposed in our proposal. However, there is always a time constraint which leaves scope for improvement. There are some particular things we would like to improve, given the opportunity to work on this project again:

- We could structure our application in better way so that more functionalities when added can be rendered in the application.
- We could implement more model classes to structure the application based on user types and categories

PROJECT CONTRIBUTION

We were a group of two members, so we divided our tasks equally and met bi-weekly for scrum meetings. We discussed our problems in the meeting and gave extra effort to the points where we were lacking. We divided the functionality module and tasks accordingly:

Rohit	Nibir
Login User	Registering User
Edit Profile	Sending Messages

Edit Profile Picture	Search Profile
Configurable Business logic for Jobs	Adding/Applying Jobs
Error Logging	Continuous Integration Setup
Posting Status	Change Password
Design Patterns: Template, Decorator	Design Pattern: Factory Pattern, Singleton

CONCLUSION

This project was a very exhaustive learning experience, which made us understand the importance of test driven development and having to create the application which keeps the quality-code as its prime feature. Every application has a room for improvement and so does ours. We can add much more unit test scenarios as part of our test plan and also more functionalities and features can be added to utilize it as a real industry-ready-application.