Table of Contents

Job Find System – Java Application

Part A – Problem Analysis

User/System Requirements (Use cases)

The first part of the project's research concerns the thorough understanding of the basic requirements of the system. Through this process, the specifications and expectations to be achieved during the development of the system are clarified.

Specifically, the main functions are:

1. Candidate registration: Users (job seekers) should be able to register as candidates by providing their resumes, demographics and contact information, as well as their preferences regarding the desired job characteristics.
2. Job registration: Users (employers) should be able to add new jobs to the system’s list that is viewable by the candidate. The job registration should include basic details about the company, job position, and comparison candidates’ criteria.
3. Register of interest in jobs: Candidates should be able to express interest in one or more available jobs.
4. Deactivate a job post: Users (employers) should be able to deactivate a job post when it is no longer available.
5. Job recommendations: The system should recommend suitable jobs to candidates based on their preferences and job characteristics.
6. View job suggestions: Candidates should be able to see a list of job suggestions that matching their preferences.

Data Collection

The second part focuses on the collection and classification/categorization of the data that the system must store in its database. This process is critical because it helps to understand the type and structure of data that will be used to implement the functionalities mentioned above.

Specifically:

- Job characteristics (city of work, years of experience, profession category, driving license possession, business trips, etc.).
- Job description, responsibilities, minimum requirements, posting date, etc.
- Company details
- Job characteristics/candidate criteria for comparison (city of work preference, years of experience, profession category, driver's license requirement, willing for business travel).

User:

- Account name (username)
- Email address
- Password
- Address (residential/company)
- Contact Phone
- Resume (not for employer)

Job Application:

- Date of application
- *Job Details*
- *User Details*
- *Job Comparison Elements*

Company Details:

- Company name
- Company address
- Company Contact Details

Profession Category:

- Description (name)

Job Comparison Elements:

- City of desire for work
- City of residence of the candidate
- Years of experience
- Category of Profession of Choice
- Possession of a driver's license? (boolean)
- Willing for business trips? (boolean)

Job Position:

- City of work
- Minimum years of requested experience
- Further job brief
- Responsibilities
- Minimum requirements
- Post date
- Active/Inactive? (boolean)
- Profession Category
- Does it require a driver's license? (boolean)
- Does it include business trips? (boolean)

Part B – Diagrams (UML ERD and Class Diagram)

Once the system requirements and data have been collected, the research of the project continues with the diagrams design.

Entity Relationship Diagram (ERD)

To make it easier to understand how the program will develop, it is important to move the design process step by step, incorporating complexity gradually.

For this reason, I found it necessary to create an ERD diagram before the Class diagram. This provides a first picture of the basic models (Entities) and attributes/properties stored in the system, as well as their connections, without yet involving methods and interfaces.

The way the system database is designed depends on several criteria.
In this particular case, I would like to focus on three of them:

1. Speed and Maintenance:
   For programs (like this one) that can handle tens of thousands of records, there is always the concern about how many columns should be in a table and how many of them should be broken into more tables (normalization). It's a trade-off between the speed and easy maintenance that a database can have.

   Specifically, if we put all the data in one table, the use of joins is reduced (thus faster access to the data). But a huge table is difficult to manage, understand and maintain. On the other hand, using more tables leads to a more structured code (easy management and maintenance) but too many unnecessary tables lead to the use of many joins that will affect performance (speed reduction).

2. <u>Memory</u>: another criterion is the correct management of system memory capacity. For example, placing too many columns in a table increases the risk of having too many NULL values. For tens of thousands of records it would create a huge problem because we would have a large piece of memory reserved which would be unused.

3. <u>Links between tables</u>: Properly designed relationships between tables allow users to search and retrieve data more efficiently.

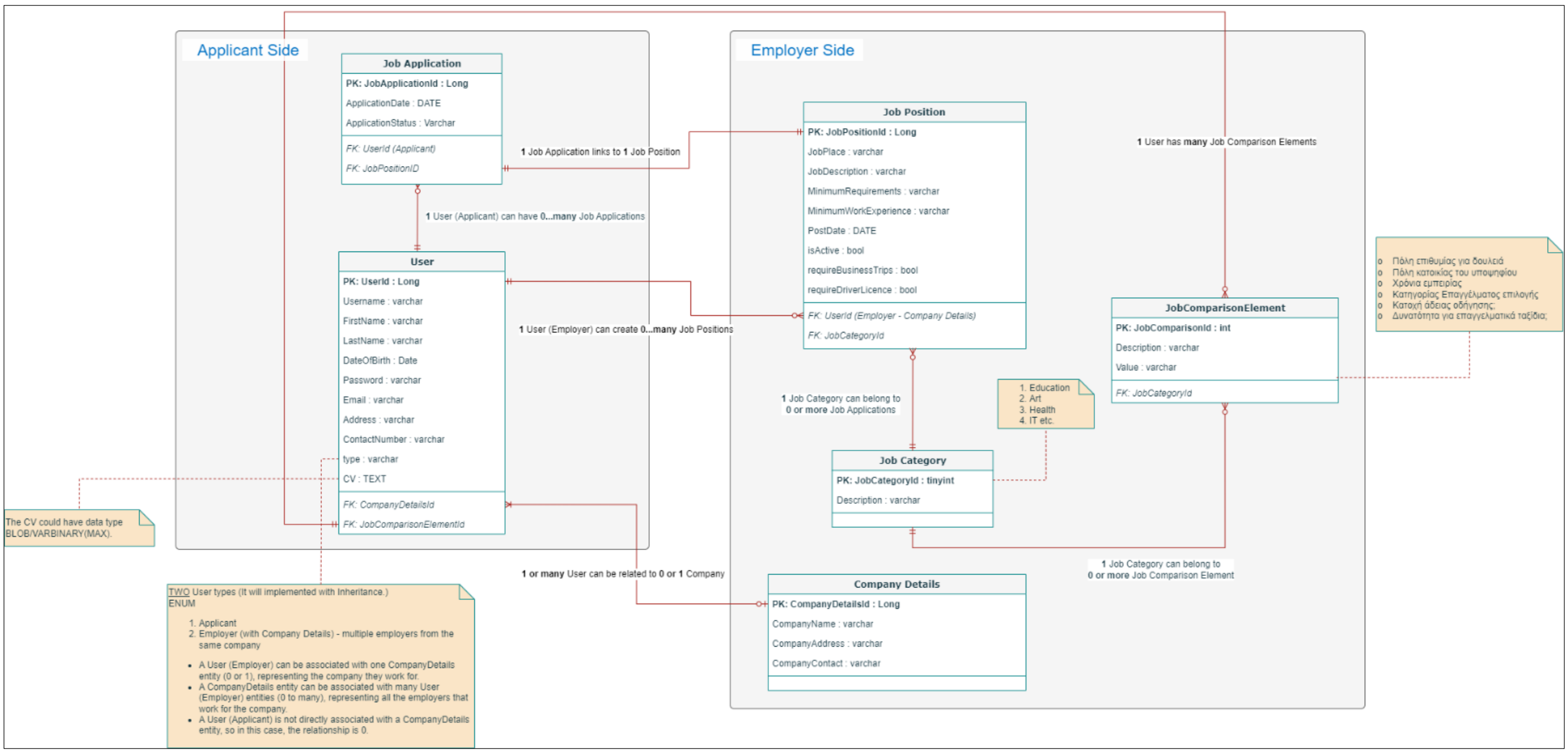
The suggestions for addressing the above concerns are represented by the table below. Especially:

1. <u>Speed and Maintenance</u>: The decision to create a new table is based on whether the user needs to manage this data as an independent entity through the program (through some API calls). For example, the columns from the *JobComparisonElement* table could be part of the *User* table. So each user would have their own set of comparison elements.

   However, for the reason that the *JobComparisonElement* table may be changed frequently, it would be more flexible to allow the competent user (who does not have access to the code) to update these data through the application. Similarly for the table *JobCategory*. Following this, we achieve the trade-off to have a well-structured and fast DB as the number of tables is necessary and the use of joins specific.

2. <u>Memory</u>: in combination with suggestion 1 (mentioned-above), the information was divided into more than one table. For example, all job-related information was not placed in one table but in three (*JobComparisonElement, Job Application* and *Job Position*). Also, columns were included that are compatible with the collected data and do not deviate from the logic of the system.

3. <u>Links between tables</u>: the selection of connections was based on system requirements. For example, <u>one</u> candidate will be able to express interest in <u>one or more</u> available jobs. Therefore, the *User* table is linked to the *Job Position* table in a <u>one-to-many</u> relationship.



The CV could have data type BLOB/VARBINARY(MAX).

<u>TWO</u> User types (It will implemented with Inheritance.)
ENUM

1. Applicant
2. Employer (with Company Details) - multiple employers from the same company

- A User (Employer) can be associated with one CompanyDetails entity (0 or 1), representing the company they work for.
- A CompanyDetails entity can be associated with many User (Employer) entities (0 to many), representing all the employers that work for the company.
- A User (Applicant) is not directly associated with a CompanyDetails entity, so in this case, the relationship is 0.

- Πόλη επιθυμίας για δουλειά
- Πόλη κατοικίας του υποψηφίου
- Χρόνια εμπειρίας
- Κατηγορίας Επαγγέλματος επιλογής
- Κατοχή άδειας οδήγησης;
- Δυνατότητα για επαγγελματικά ταξίδια;

Class Diagram

The last step in the system design process is to create a Class diagram. The information in the ERD diagram (entities and columns) clarifies the main Classes/Models and their basic characteristics (attributes/ properties). Also, the Use Cases listed above represent a large part of the functions/responsibilities (methods) that should be included within the relevant Classes.

In addition, the design of a Class diagram is based on how the code should be structured and how the classes should communicate with each other. The project follows the rules of the SOLID Principles which ensures that each class individually has a unique responsibility (High Cohesion – ease of code maintenance) and that classes should be as independent as possible reducing the minimum impact of changes between them (Low coupling – communication through interfaces).

To achieve this, the 'Model-Repository-Service-Controller' and Façade Design Patterns were used:

- Models: the (ERD) *Entity* tables stored in the database. Each of them will have a:
- Repository: it has the duty to update the database. That is, to communicate (through SQL queries) with the corresponding Class of the Model located in the database and apply the corresponding changes (e.g., CRUD operations)
- Service: calls the methods of the *Repository* but also includes extra business logic (e.g., if-statements for whether the specific *Model* can be deleted or not)
- Controller: calls Service methods and includes only logic related to communication with the client (e.g., API Calls with Response messages)

It is important to note that:

- Each class has a single responsibility. For example, the *UserController* class is responsible only for handling HTTP requests related to users.
- The last three Classes communicate via Interfaces. Dependency injection is applied where the Service communicates with the *RepositoryInterface* and the Controller with the *ServiceInterface*.
- High-level modules (such as Services) do not depend on low-level modules (such as Repositories). Both depend on abstractions (such as Interfaces)

Also, the use of Inheritance with Parent *User* Class and child Classes (*Applicant* and *Employer*) allows us to apply the DRY method. This will break down the responsibilities/methods and attributes of each user depending on the role.

Finally, there could be, for example, a Façade class e.g., *JobMatchingFacade* class whose role would be to match candidates to jobs. This class will reduce the complexity of the candidate and job matching process by including methods that call the corresponding *UserService* and *JobPositionService* methods (communication always through Interfaces). Thus, we place a logic of the code in a separate place. Otherwise, *UserService's* methods would be injected inside the *JobPositionService* and vice versa, making the code more difficult to maintain.

In summary, some of the features covered in the Class diagram are:

- The ability to register new job posts : the user logged in as *EmployerUser* will be able to add new job post by calling the method/API call ***JobPositionController → addJobPosition()***. The method receives as input parameter the *JobApplication* object with all the necessary information. Thus, through the 'Repository-Service-Controller' pattern (and Interfaces) the new information will be added to the database.

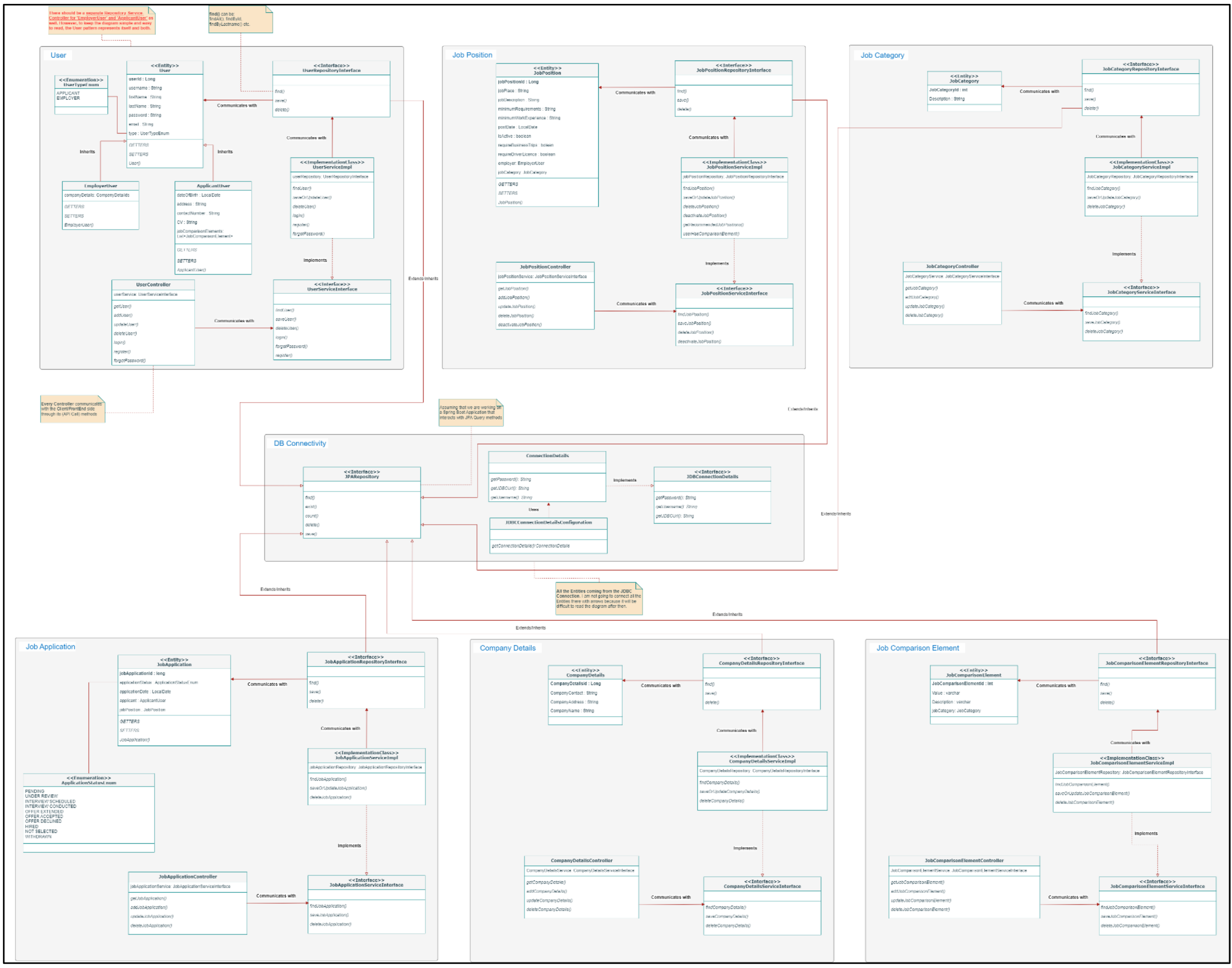
- The ability to deactivate a job post when it is no longer available: the user logged in as *EmployerUser* will be able to search for the job position he wants and deactivate it. This can be done using the call method/API ***JobPositionController→findJobPosition().***
Then call the method/API ***JobPositionController → deactivate()*** which will take *JobPosition* as input parameter object searched for by the user.

- The ability to register candidate profiles: the applicant user will be able to register in the system by selecting as *ApplicantUser using the call/API* method ***ApplicantUserController → register()***. There, he/she will fill in all the basic personal information necessary to create his/her profile. At the same time, since the *ApplicantUser* object has as a Class member a *List<JobComparisonElement>*, when registering, will be asked to fill in this information as well.

- The possibility for each candidate to express interest in one or more available positions: the user logged in as *ApplicantUser* will be able to search for the job position he/she wants (one or more) using the call ***JobPositionController →findJobPosition()*** method/API call. He/she will then be able to apply using the method/API call ***JobApplicationController → addJobApplication()*** which sets as a parameter the chosen job position.

- The generate process of a list of job proposals per candidate, which will be done by the system by performing an evaluation method: before the system displays all available job positions, the method ***JobServiceImpl → getRecommendedJobPositions()*** will compare the applicant's comparison data with the corresponding job positions criteria. Then the jobs that will have the most matches, they will have the highest score and will appear at the top of the sorted list.

THE DIAGRAM IS ON THE NEXT PAGE


There should be a separate Repository, Service, Controller for 'EmployerUser' and 'ApplicantUser' as well. However, to keep the diagram simple and easy to read, the User pattern represents itself and both.
find() can be: findAll(), findById, findByLastname() etc.
User
<<Enumeration>> UserTypeEnum
APPLICANT
EMPLOYER
<<Entity>> User
userId : Long
username : String
firstName : String
lastName : String
password : String
email : String
type : UserTypeEnum
GETTERS
SETTERS
User()
<<Interface>> UserRepositoryInterface
find()
save()
delete()
Communicates with
Inherits
Inherits
EmployerUser
companyDetails: CompanyDetails
GETTERS
SETTERS
EmployerUser()
ApplicantUser
dateOfBirth : LocalDate
address : String
contactNumber : String
CV : String
jobComparisonElements: List<JobComparisonElement>
GETTERS
SETTERS
ApplicantUser()
<<ImplementationClass>> UserServiceImpl
userRepository: UserRepositoryInterface
findUser()
saveOrUpdateUser()
deleteUser()
login()
register()
forgotPassword()
Implements
UserController
userService: UserServiceInterface
getUser()
addUser()
updateUser()
deleteUser()
login()
register()
forgotPassword()
<<Interface>> UserServiceInterface
findUser()
saveUser()
deleteUser()
login()
forgotPassword()
register()
Every Controller communicates with the Client/FrontEnd side through its (API Call) methods
Extends/Inherits
Job Position
<<Entity>> JobPosition
jobPositionId : Long
jobPlace : String
jobDescription : String
minimumRequirements : String
minimumWorkExperience : String
postDate : LocalDate
isActive : boolean
requireBusinessTrips : boolean
requireDriverLicence : boolean
employer: EmployerUser
jobCategory : JobCategory
GETTERS
SETTERS
JobPosition()
<<Interface>> JobPositionRepositoryInterface
find()
save()
delete()
<<ImplementationClass>> JobPositionServiceImpl
jobPositionRepository: JobPositionRepositoryInterface
findJobPosition()
saveOrUpdateJobPosition()
deleteJobPosition()
deactivateJobPosition()
getRecommendedJobPositions()
userHasComparisonElement()
JobPositionController
jobPositionService: JobPositionServiceInterface
getJobPosition()
addJobPosition()
updateJobPosition()
deleteJobPosition()
deactivateJobPosition()
<<Interface>> JobPositionServiceInterface
findJobPosition()
saveJobPosition()
deleteJobPosition()
deactivateJobPosition()
Job Category
<<Entity>> JobCategory
JobCategoryId : int
Description : String
<<Interface>> JobCategoryRepositoryInterface
find()
save()
delete()
<<ImplementationClass>> JobCategoryServiceImpl
JobCategoryRepository: JobCategoryRepositoryInterface
findJobCategory()
saveOrUpdateJobCategory()
deleteJobCategory()
JobCategoryController
JobCategoryService: JobCategoryServiceInterface
getJobCategory()
editJobCategory()
updateJobCategory()
deleteJobCategory()
<<Interface>> JobCategoryServiceInterface
findJobCategory()
saveJobCategory()
deleteJobCategory()
Assuming that we are working on a Spring Boot Application that interacts with JPA Query methods
DB Connectivity
<<Interface>> JPARepository
find()
exist()
count()
delete()
save()
ConnectionDetails
getPassword(): String
getJDBCUrl(): String
getUsername(): String
<<Interface>> JDBCConnectionDetails
getPassword(): String
getUsername(): String
getJDBCUrl(): String
Uses
JDBCConnectionDetailsConfiguration
getConnectionDetails():ConnectionDetails
All the Entities coming from the JDBC Connection. I am not going to connect all the Entities there with arrows because it will be difficult to read the diagram after then.
Job Application
<<Entity>> JobApplication
jobApplicationId : long
applicationStatus : ApplicationStatusEnum
applicationDate : LocalDate
applicant : ApplicantUser
jobPosition : JobPosition
GETTERS
SETTERS
JobApplication()
<<Interface>> JobApplicationRepositoryInterface
find()
save()
delete()
<<Enumeration>> ApplicationStatusEnum
PENDING
UNDER REVIEW
INTERVIEW SCHEDULED
INTERVIEW CONDUCTED
OFFER EXTENDED
OFFER ACCEPTED
OFFER DECLINED
HIRED
NOT SELECTED
WITHDRAWN
<<ImplementationClass>> JobApplicationServiceImpl
jobApplicationRepository: JobApplicationRepositoryInterface
findJobApplication()
saveOrUpdateJobApplication()
deleteJobApplication()
JobApplicationController
jobApplicationService: JobApplicationServiceInterface
getJobApplication()
addJobApplication()
updateJobApplication()
deleteJobApplication()
<<Interface>> JobApplicationServiceInterface
findJobApplication()
saveJobApplication()
deleteJobApplication()
Company Details
<<Entity>> CompanyDetails
CompanyDetailsId : Long
CompanyContact : String
CompanyAddress : String
CompanyName : String
<<Interface>> CompanyDetailsRepositoryInterface
find()
save()
delete()
<<ImplementationClass>> CompanyDetailsServiceImpl
CompanyDetailsRepository: CompanyDetailsRepositoryInterface
findCompanyDetails()
saveOrUpdateCompanyDetails()
deleteCompanyDetails()
CompanyDetailsController
CompanyDetailsService: CompanyDetailsServiceInterface
getCompanyDetails()
editCompanyDetails()
updateCompanyDetails()
deleteCompanyDetails()
<<Interface>> CompanyDetailsServiceInterface
findCompanyDetails()
saveCompanyDetails()
deleteCompanyDetails()
Job Comparison Element
<<Entity>> JobComparisonElement
JobComparisonElementId : int
Value : varchar
Description : varchar
jobCategory: JobCategory
<<Interface>> JobComparisonElementRepositoryInterface
find()
save()
delete()
<<ImplementationClass>> JobComparisonElementServiceImpl
JobComparisonElementRepository: JobComparisonElementRepositoryInterface
findJobComparisonElement()
saveOrUpdateJobComparisonElement()
deleteJobComparisonElement()
JobComparisonElementController
JobComparisonElementService: JobComparisonElementServiceInterface
getJobComparisonElement()
addJobComparisonElement()
updateJobComparisonElement()
deleteJobComparisonElement()
<<Interface>> JobComparisonElementServiceInterface
findJobComparisonElement()
saveJobComparisonElement()
deleteJobComparisonElement()
Communicates with
Implements
Extends/Inherits

Part C – Code Implementation Results

The above diagram could be implemented with Java SpringBoot Framework as it would effectively manage the dependency injection between Classes and Interfaces communication. However, due to limited time, a simpler version (pure BETA Java Application) was created. It follows the SOLID Principles and Design Patterns methodologies mentioned in the above sections. The application covers all the necessary features/functions mentioned in the Class Diagram. The Project code is included in the same folder as this document and the *jar* file is located at the project path *out/artifacts/JobFindSystem_jar/JobFindSystem.jar*.

Note that the *JobFindSystem* Class does not exist in the Class Diagram. It represents the system's database. This is where the random/dummy data are generated.

Due to time constraints, no SQL database was created.

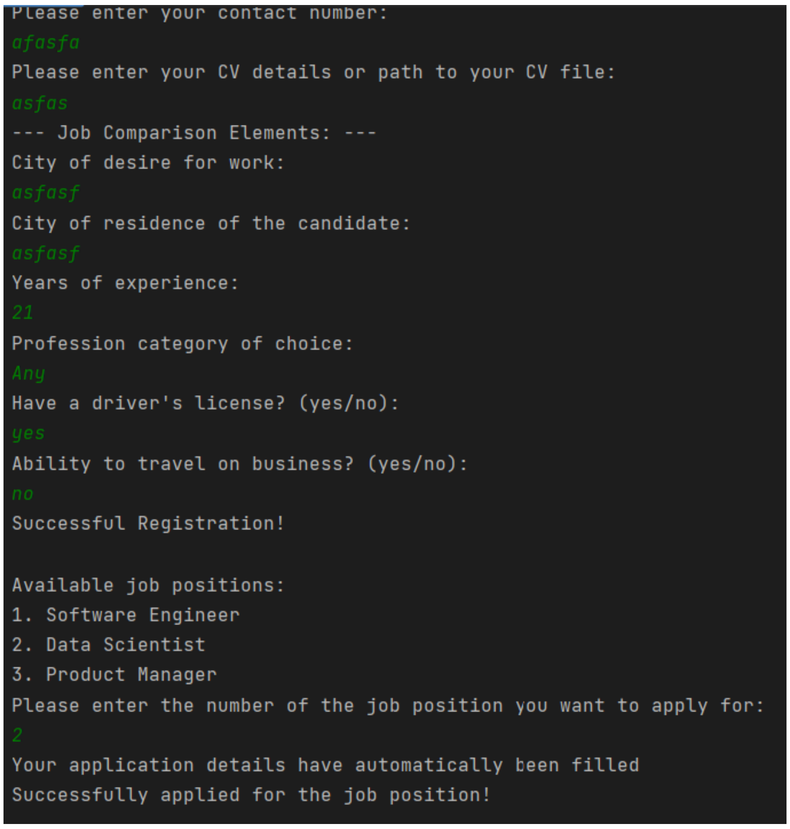
Keep in mind that this is a simplified example and does not include many important aspects of a real application, such as exception handling, database connection etc.

The flow of the application is:

1. The user declares whether he wants to register as an Applicant or Employer
2. If he registers as an Applicant, he will be shown all the available jobs (those that most closely match the data/comparison criteria first). Then he is given the opportunity to apply to the job he wants by selecting from that list.
3. If Employer option is chosen, the employer user can add a new job post or disable a job from the list provided.

Below are images with the results displayed by the console of the application:

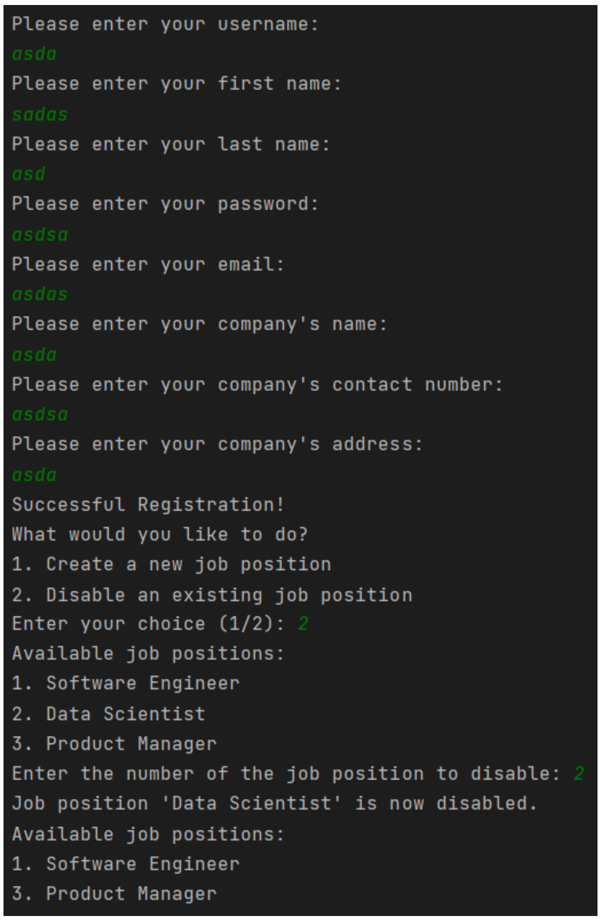
1. Application Process (sorry for *asfas* values)



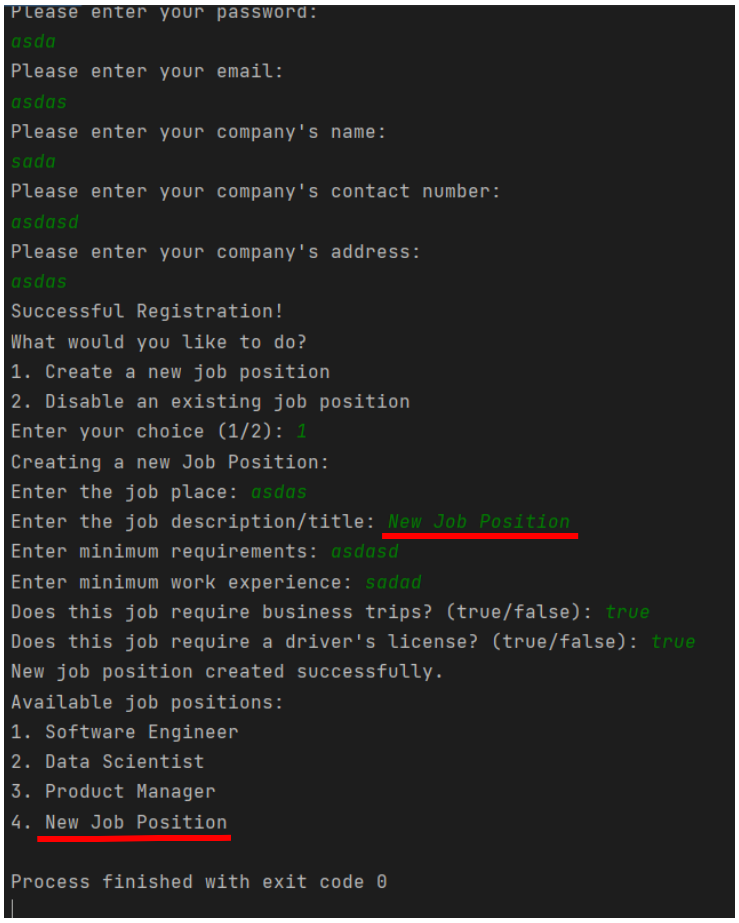
```
Please enter your contact number:
afasfa
Please enter your CV details or path to your CV file:
asfas
--- Job Comparison Elements: ---
City of desire for work:
asfasf
City of residence of the candidate:
asfasf
Years of experience:
21
Profession category of choice:
Any
Have a driver's license? (yes/no):
yes
Ability to travel on business? (yes/no):
no
Successful Registration!

Available job positions:
1. Software Engineer
2. Data Scientist
3. Product Manager
Please enter the number of the job position you want to apply for:
2
Your application details have automatically been filled
Successfully applied for the job position!
```

2. Employer Registration and disable JobPosition Process


```
Please enter your username:
asda
Please enter your first name:
sadas
Please enter your last name:
asd
Please enter your password:
asdsa
Please enter your email:
asdas
Please enter your company's name:
asda
Please enter your company's contact number:
asdsa
Please enter your company's address:
asda
Successful Registration!
What would you like to do?
1. Create a new job position
2. Disable an existing job position
Enter your choice (1/2): 2
Available job positions:
1. Software Engineer
2. Data Scientist
3. Product Manager
Enter the number of the job position to disable: 2
Job position 'Data Scientist' is now disabled.
Available job positions:
1. Software Engineer
3. Product Manager
```

3. Add a new Job Position (sorry for *asdasd* values)


```
Please enter your password:
asda
Please enter your email:
asdas
Please enter your company's name:
sada
Please enter your company's contact number:
asdasd
Please enter your company's address:
asdas
Successful Registration!
What would you like to do?
1. Create a new job position
2. Disable an existing job position
Enter your choice (1/2): 1
Creating a new Job Position:
Enter the job place: asdas
Enter the job description/title: New Job Position
Enter minimum requirements: asdasd
Enter minimum work experience: sadad
Does this job require business trips? (true/false): true
Does this job require a driver's license? (true/false): true
New job position created successfully.
Available job positions:
1. Software Engineer
2. Data Scientist
3. Product Manager
4. New Job Position

Process finished with exit code 0
```