Jobs Management Application

Analysis and Design Document

Student: Galeata Bianca-Floriana

**Group:30235**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 15.03.2018 | 1.0 | First version | Galeata Bianca-Floriana |
| 24.04.2018 | 1.1 | Iteration 1.2 | Galeata Bianca-Floriana |
|  |  |  |  |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 5

2.1 Conceptual Architecture 5

2.2 Package Design 6

2.3 Component and Deployment Diagrams 7

III. Elaboration – Iteration 1.2 7

1. Design Model 7

1.1 Dynamic Behavior 7

1.2 Class Design 7

2. Data Model 7

3. Unit Testing 7

IV. Elaboration – Iteration 2 8

1. Architectural Design Refinement 8

2. Design Model Refinement 8

V. Construction and Transition 8

1. System Testing 8

2. Future improvements 8

VI. Bibliography 9

# Project Specification

The project is an application designed to make an easier communication between companies and persons who are looking for a job. The objective is to offer a simple way for a person to find a suitable job and for companies to find potential employee.

Each user has a username and a password in order to login. Depending on the type that the username is register, the user interface will display specific utilities.

An *unemployed* person can do:

* Login/logout
* Create account
* Search for a job using different criteria
* Apply to a job
* Save jobs to view later

An *agent* can do:

* Login/logout
* CRUD operations on its jobs
* Search for potential employees using different criteria

There may appear another type of user, namely an *administrator* which can do:

* CRUD operations on user types from above

# Elaboration – Iteration 1.1

# Domain Model

The models that we will work with are:

* Client (the unemployed user)
* Agent (the agent from a company)
* Job (what an agent offer and what a client wants)

About a client we will know:

* Name
* Areas of interest
* Username and password
* Jobs on which he applied
* jobs he is interested in

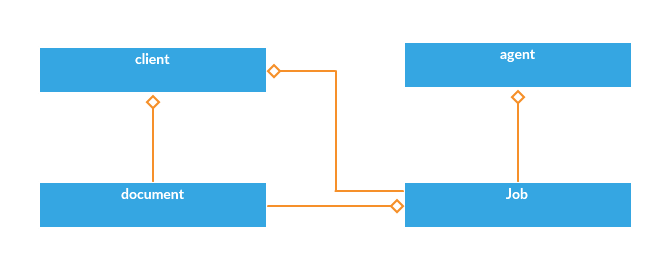
About an agent we will know:

* Name
* Username and password
* Contacts
* Announcement about jobs that he has made

About a job we will know:

* Name
* Level of experience needed
* Type (part time / full time)
* Domain (IT, Finance)
* Company that offers
* Location (country, city etc.)

Next we will present the conceptual diagram:



# Architectural Design

## Conceptual Architecture

For this project the chosen architectural pattern is client -server. It has been chosen because it is a particular case of Layer architecture and brings many of the advantages it has, besides others that are specific to client-server architecture.

One of the reason of this decision is the simple and logical division between client and server. Each of them are clearly defined and also each of them knows what have to do.

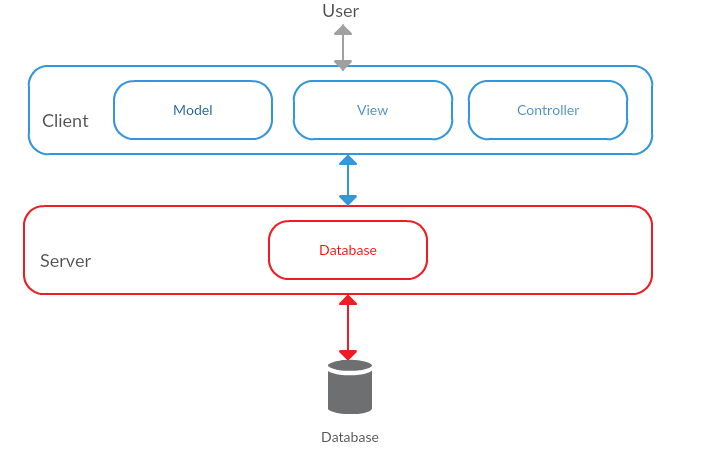
The server is that part of the architecture that provides services. In this idea all the services needed are grouped here. The server in turn will have a layer architecture that will help to have organized services.

Another thing to consider is that a single server will serve many clients, and this project being a web application this feature help when many clients are connected at once and also will help to provide personified interaction for each user.

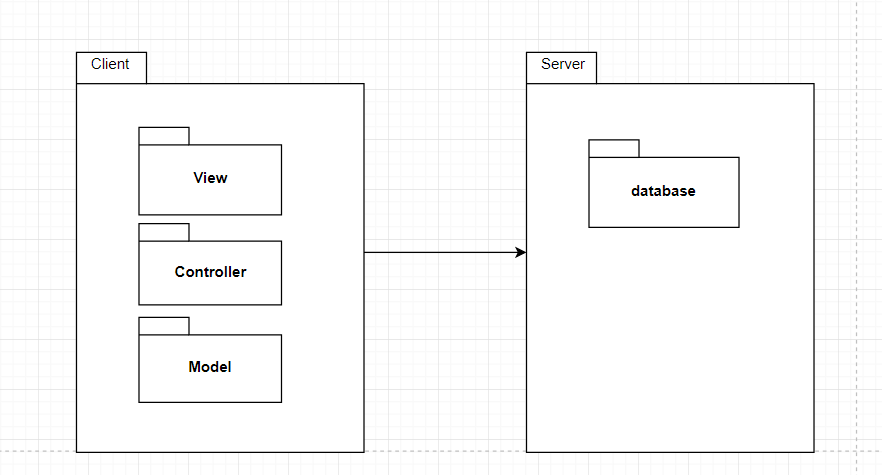
The client is that part of the architecture that uses the services from the server in order to assure a good interaction between user and application.

This separation of logic and features between client and server make it easy to change the user interface when needed without the changes to propagate in the server too. In addition, this assure the security of the application, the server only has some exposed services, the rest of the logic being encapsulated.

The architectural diagram is:



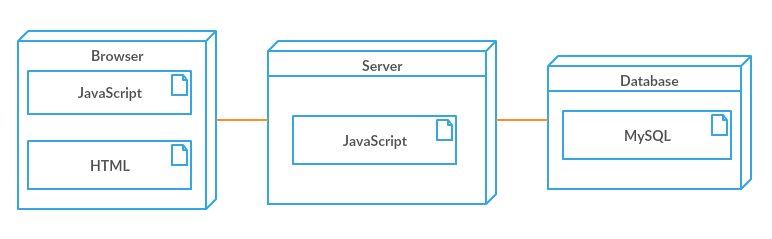
## Package Design



As the architecture shows, the main packages are client and server. In the first one, there are included 2 smaller packages that represent the user interface (View) and the processing logic from client (controller). In the server package: database package (manipulate and access the database), the model package (object that are worked with like job etc.) and the domain package (where is the logic for processing data).

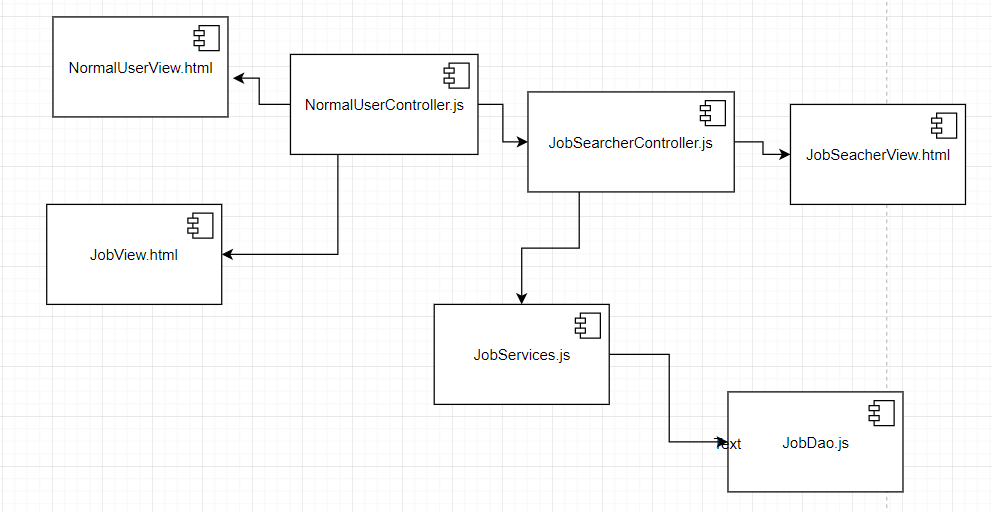
## Component and Deployment Diagrams

* *Deployment diagram*



In this diagram is presented how the application is distributed over some physic components. The application is divided between three machines: one where the browser runs, one where is the server and one for the database.

* *Component diagram*



This component diagram shows how classes communicate in order to accomplish a search action. As it can be seen the view communicates with its controller and from there the controller communicates with the other controllers and with services. The services have a connection to the class that operates with the database. This diagram shows the general communication between classes, in a particular case there can be more than one type of class involved.

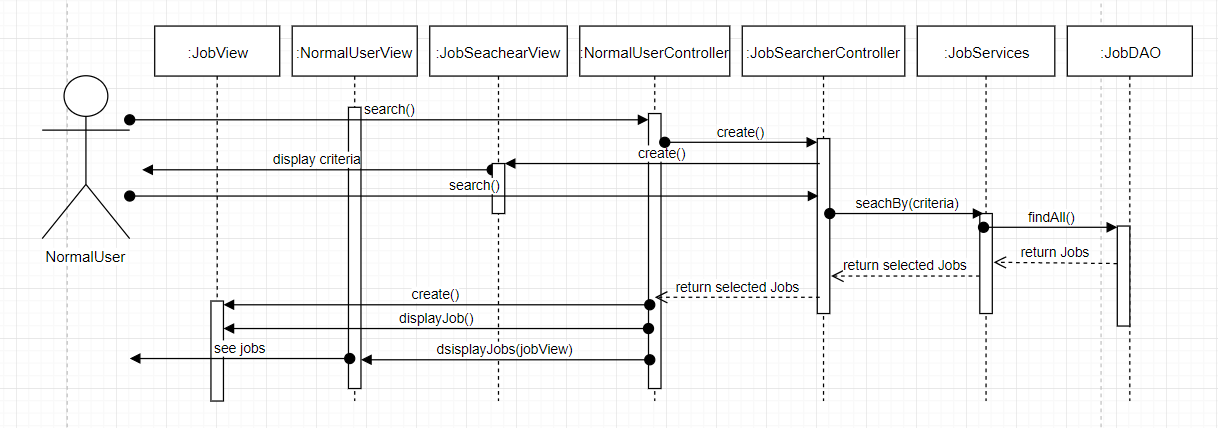
# Elaboration – Iteration 1.2

# Design Model

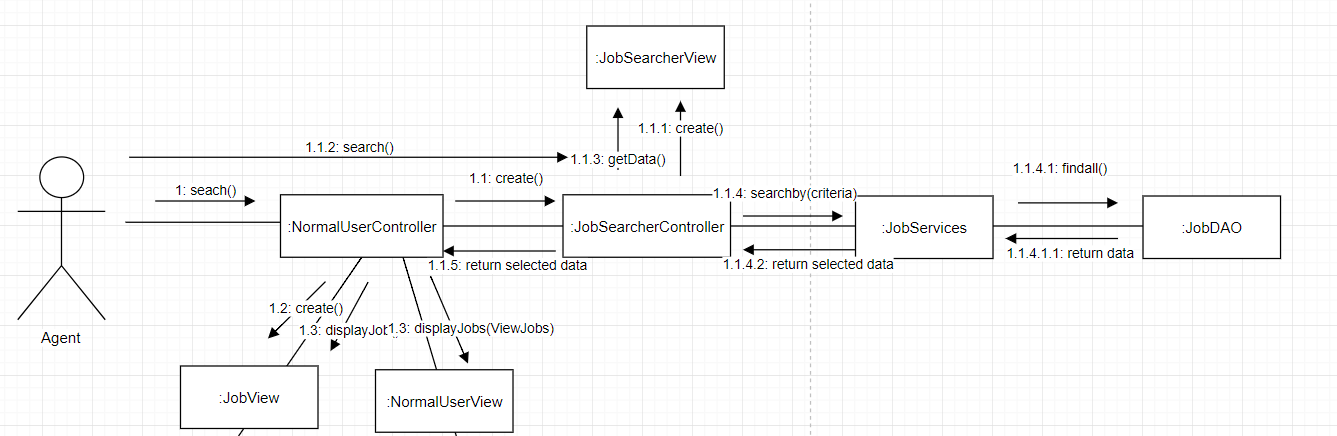
## Dynamic Behavior

For this section 2 important scenario have been chosen in order to present the dynamic behavior. The first one refers to a normal user which wants to search for a job using some filters. The below sequence and communication diagram show how this is made:

* *Sequence diagram*



* *Communication diagram*

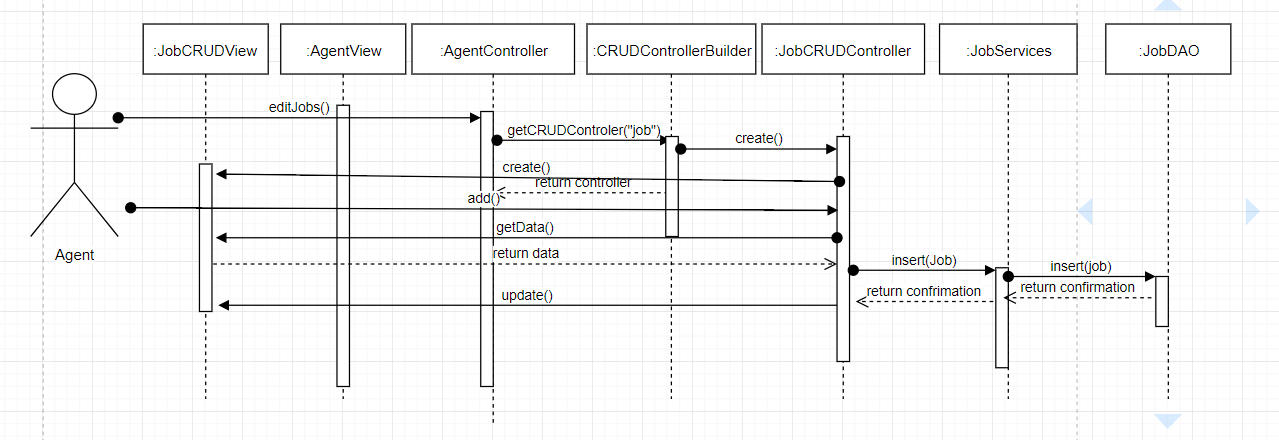


Both diagrams describe the next flow:

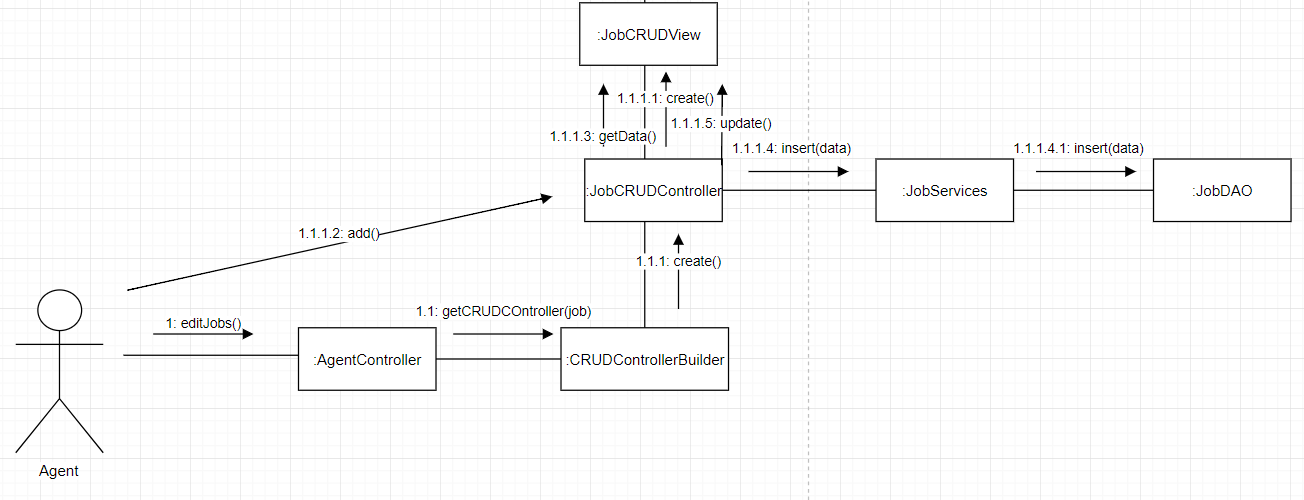
1. The user selects the button search from the view,
2. This make a new controller and view to appear
3. From the new view he selects the criteria and then send to the new controller the command to make the search
4. It uses services to make the search and return the result to the main controller
5. Here small views for jobs are made and then inserted in the principal view

The second scenario refers to an agent who wants to add a new announce about a job.

* Sequence diagram



* Communication diagram



For this scenario the flow is:

1. The agent selects from the view the button for editing jobs
2. This makes a new view and controller for editing
3. From the new controller, after inserting data in the view the user chose to add the job
4. The controller uses services to insert in the database the new job and then it updates the editing view

## Class Design

In the next diagram, we will the classes that will be created. For this project have been used design patterns:

* *Singleton*

The Singleton design pattern have been user for the class that has the connection to the database. It is important and necessary to use this pattern for the connection because it is desired to have a single connection to a database. If there are more, conflicts and inconsistence may appear when more users modify the data from the database.

* *Observer*

This design pattern has been used for the connection between a normal user and an announcement. The idea is that a user has to be notify when an announce in which is interested, is near the expiration date.

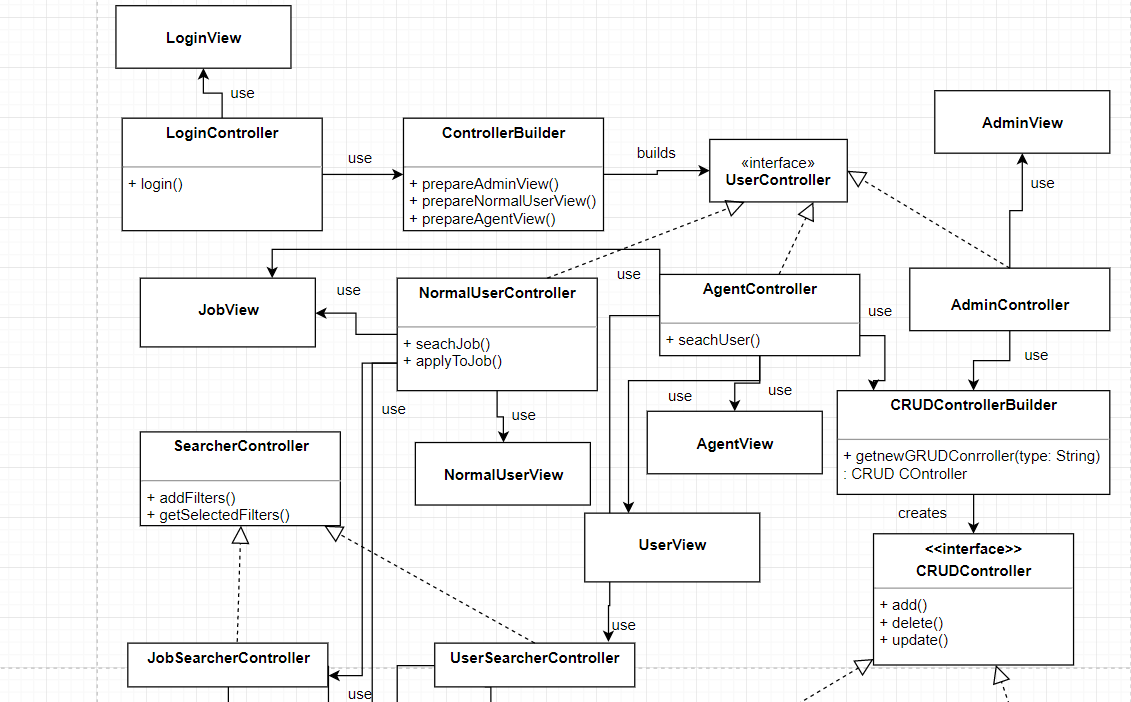
* *Factory*

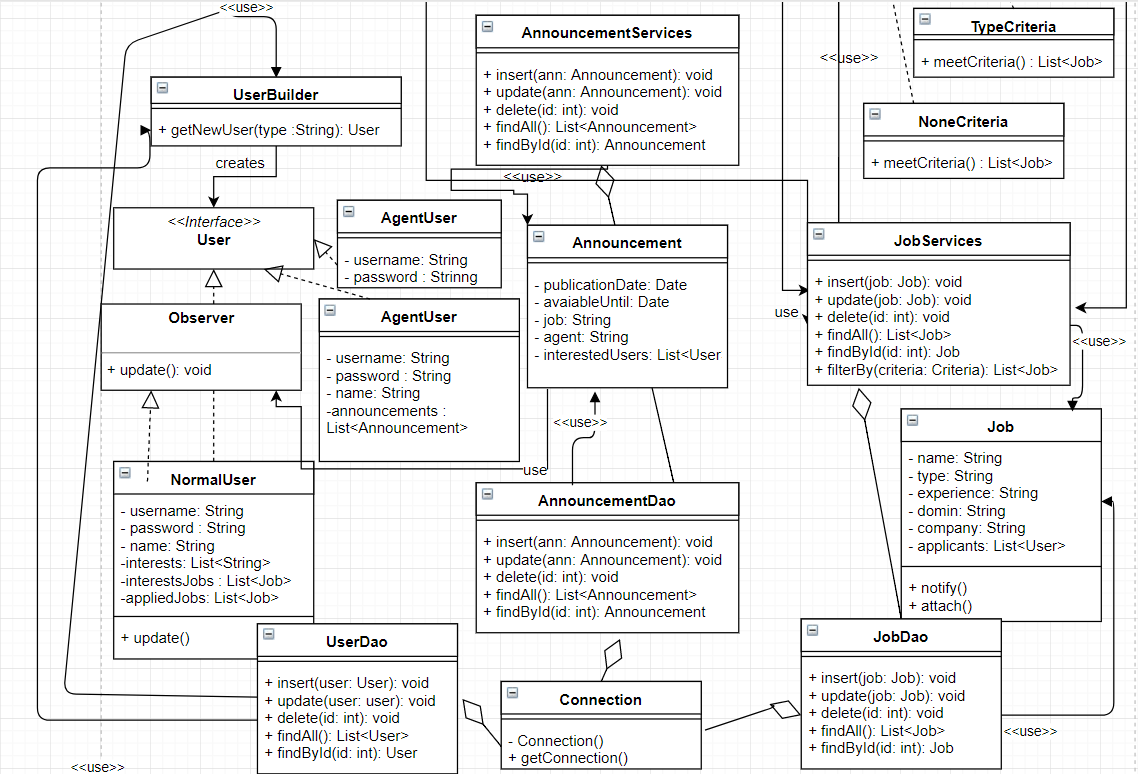
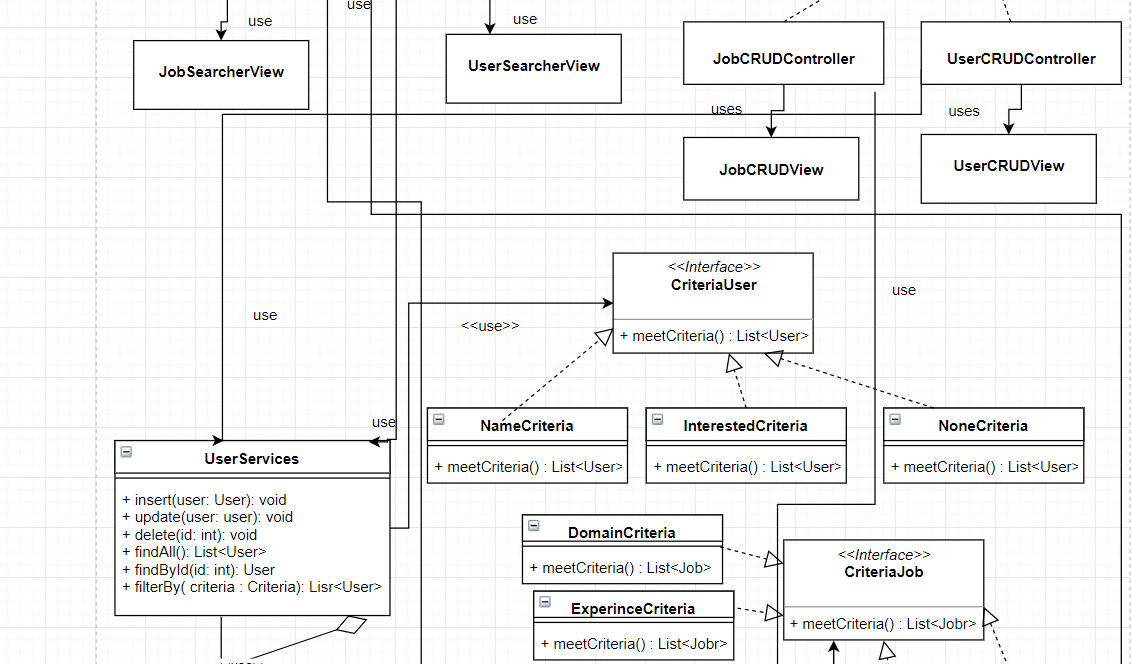
This one have been used for creating a user. There are three types of users: admin, normal user and an agent. We do not want to know explicitly how there are created, we only need that type of user. This design pattern offer encapsulation.

It also has been used to create a CRUD controller. There are two types of this controller (for users and for jobs), but we do not want to know how them are made.

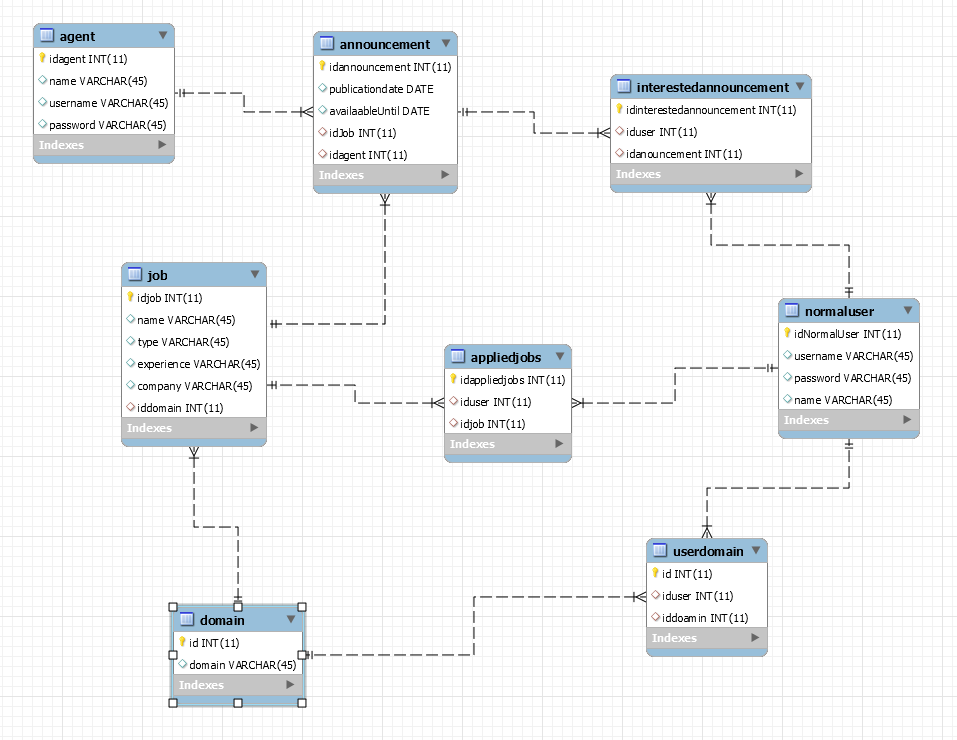
* *Filter*

A basic user can perform searching after different criteria, so an easy way to add filters is to use this design pattern. It also is used to make the search for agents.





# Data Model



# Unit Testing

Each class has been tasted once implemented. Tests have been made for the server and the client part and theirs classes.

For the client part each function for getting data from the user input have been tested in order to assure that the data received is what it is wanted to be received. For example, let’s suppose that a user inserts the username. To verify if that string is received correct using a method, the return string by that method have been printed, manually verified and asserts with the expected results have been made. This test is applied to any data that is received from the user (name, job type etc.)

The function for processing data have been testing by comparing the received result with the correct one. In this manner we are sure that after processing data the result is consistent. For instance, the validation of age can be considered one method of this type. It has been testing by inserting values that are right and ones that are violating the rule. The results of the functions have been verified if it corresponds with the input type. This tests have been applied to others function of this type.

The results have been printed and manually verified. Another method that has been used is the testing using assertions.

In the server part have been tested methods for the database utilization. In this category the tests have been made making inserts, deletes and updates into the database using a test class for different tables and after verifying the database to see if the operations have been successful made. The verifications have been made through application using “select” statements and from workbench, manually. For example, a job has been inserted in the database. To verify if it has really been introduced, it has been searched in the corresponding table. This type of test can be applied to any operations with the database.

Others methods for processing data have been tasted manually or using assertions. The excepted results have been compared with the actual results.

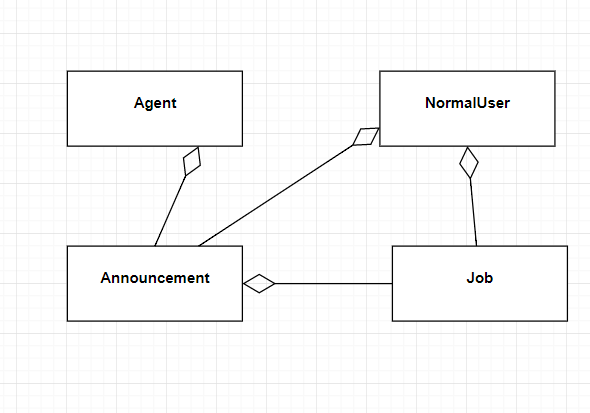
# Elaboration – Iteration 2

# Architectural Design Refinement

The conceptual diagram has been changed. The logic of the project and the concepts that are used have been change. We decided to remove the documents from the concepts and to add an announcement. The idea about this change is that:

* the documents will be send to the agent, using a mail after the agent consider the user application, so we do not need to cover this aspect.
* An announcement is important because it attach to a job an available period and the publication date. These concepts cannot be added to the job directly, because are not characteristics.

The new conceptual diagram:

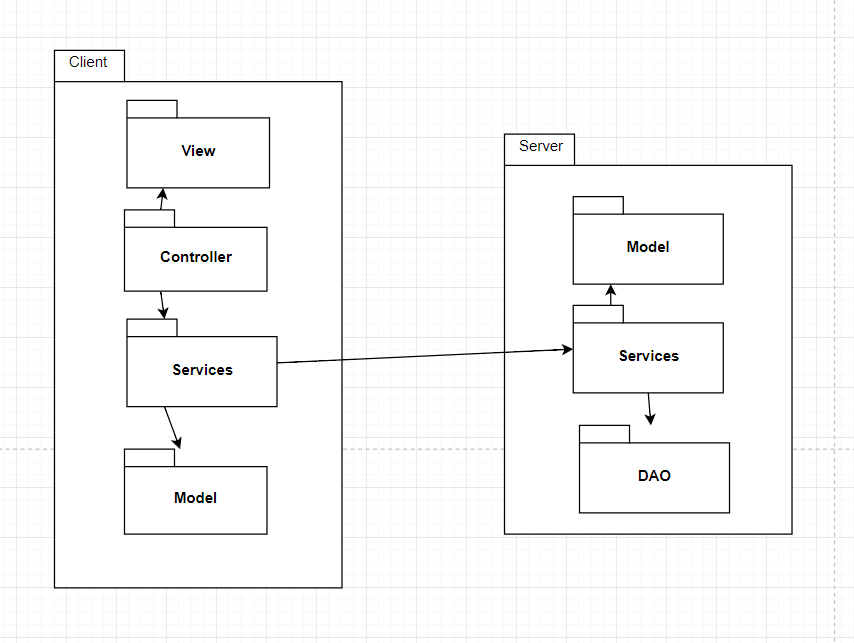


The package diagram has been refined, and new packages in the server have been added. The new packages are:

* Services - this package offers services to the client. The services are especially related to the database operation. This have been added to offer a grouped set of services to the client.
* Model – the concepts that are used in the client (like job etc.) have been added in the server to. The idea is to facilitate the communication between them. If the concepts are in both server and client, then conversions between data is no more needed.

Also a new package has been added in the client too. The package is named Services and contains logic needed when we do not use the database and the methods that makes request to the server. The packages respect the design principle.

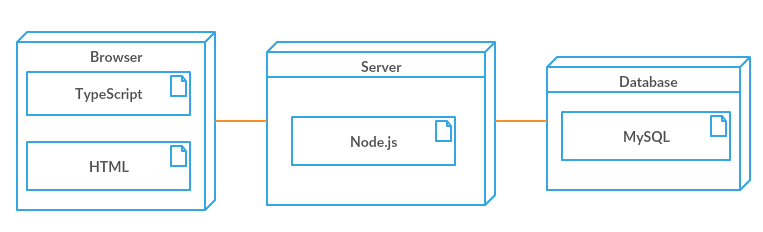
The new package diagram:



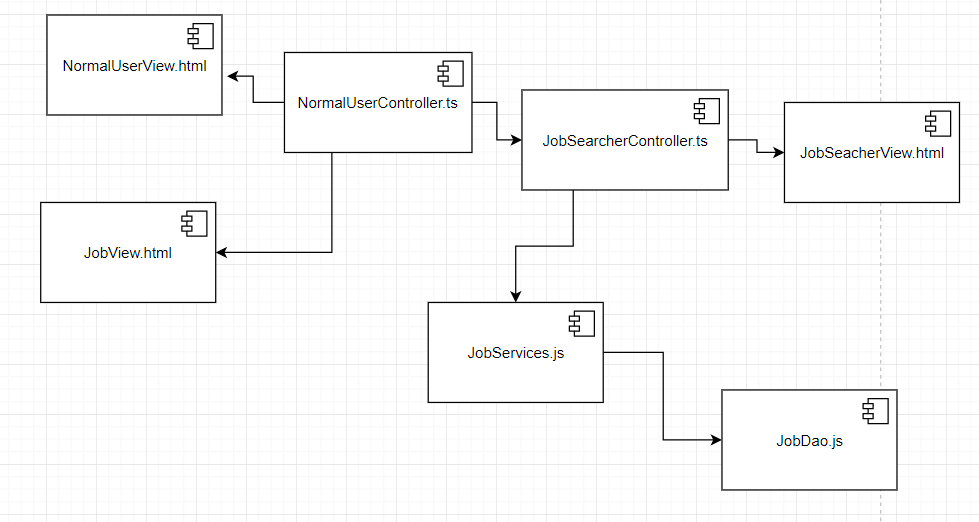
In the deployment diagram what has been change is the used technologies. In the server we will use Node.js and for the client Typescript will be used. The change has been made because using simple JavaScript, the design principle could not be used. Also, the component diagram has change because the same reason.

The new diagrams:

* Deployment diagram



* Component diagram



# Design Model Refinement

GRASP principles refers to assigning responsibility to classes and objects in [object-oriented design](https://en.wikipedia.org/wiki/Object-oriented_design).

We guide after the next principles:

* Controller

The controller pattern assigns the responsibility of dealing with system events to a non-[UI](https://en.wikipedia.org/wiki/User_Interface) class that represents the overall system or a [use case](https://en.wikipedia.org/wiki/Use_case) scenario. A controller object is a non-user interface object responsible for receiving or handling a system event.

In this project, the controllers are divided in use-case driven. We have controllers for:

* Login
* Searching for user
* Searching for job
* CRUD operations on users
* CRUD operations on jobs
* Agent operations
* Normal user operations
* Admin operations

The principle is respected, so no change has been made.

* Creator

This principle refers to which class should create another. In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:

* Instances of B contain or compositely aggregate instances of A
* Instances of B record instances of A
* Instances of B closely use instances of A
* Instances of B have the initializing information for instances of A and pass it on creation.

In this project, this responsibility has been distributed like:

* Controllers creates others controllers because the functionality of the created one is integrated in the main controller.
* Controllers creates views because they use them to display data.
* Controllers use services to complete some operations, so them can create services. (but just the one that are user; for example, an admin controller will not create a services for jobs)
* Services used DAO so access the database, so the creation is needed
* Both services and DAO use concepts from the model (like jobs) and can create them (but just the one used; for example, DAO and Services for a job will not create a user)

This principle is respected so no change is made.

* High cohesion

High cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion means that the responsibilities of a given element are strongly related and highly focused.

In the project, this principle has been respected because:

* The classes are smaller and specialized (services are grouped by the concept that they use; for example, services for jobs, user etc.)
* The controllers are made base on the use-case (controller for searching)
* Classes for different concepts have been made (user, job etc.)
* Database operations are grouped by the concept that they use (DAO for jobs etc.)

This principle is respected so no change is made.

* Indirection

The indirection pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object.

In this project the principle has been used in the situations:

* The user interface and the model is separated by the controller
* The user interface and the database access is separated by controller and services
* The controller and the database access is separated by the services

The principle is respected and no change has been made.

* Information expert

Information expert (also expert or the expert principle) is a principle used to determine where to delegate responsibilities.

To respect the principle is found in situations like:

* Controller delegate the responsibility to modify data to the services
* Services has the necessary information (received from the database or from the user to operate on data)

Just classes that have information needed in order to complete the operation have the operation as a method.

The principle is respected and no change has been made.

* Low coupling

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

In this project, low coupling is present because:

* The project is high cohesive
* The project use indirection

The dependency between classes are reduced, there are just the minimum level of dependency needed for the project to work.

* Polymorphism

According to polymorphism principle, responsibility of defining the variation of behaviors based on type is assigned to the type for which this variation happens.

The polymorphism is used in more cases:

* For creating users (a user can be normal, admin or agent; the user services can use any of those without knowing which one is)
* For creating filters (the searching method use the Interface Criteria, so it uses all the classes that implements it)
* For creating CRUD Controllers
* For creating searching Controller

The principle is respected, so no modification is needed.

* Protected variations

The protected variations pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an [interface](https://en.wikipedia.org/wiki/Interface_(computer_science)) and using [polymorphism](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) to create various implementations of this interface.

In this project anywhere the polymorphism is used, this principle is also use (for exempla when creation a CRUD controller it can have 2 implementations: one of users and one for jobs).

The principle is respected, so no modification is needed.

# Construction and Transition

# System Testing

The system testing has been made with the tools Jasimine and Karma, provided by Angular. This tools helps with testing by making automation testing.

Tests are created for each component and the tool execute them. The purpose of this test are to see if a component is created succefully.

# Future improvements

The application can be become more complex. For example, an email can be send when a job changes its state or one when a user applies to a job.

Also new features can be added, like searching by for filters.

The idea of the application can be successful if more complex features are added.

# Bibliography