**Time**: 6 hours **Deployed** http://pablo-delhoyo.herokuapp.com/

**CRUD and REST web app** **Repo** https://github.com/pdelho/brain

**How to focus it?**

I will use MVC pattern to map each functionality onto a specific URL. The controller will be triggered according to each case. Thanks to this pattern, the service layer is separated and called using an entity manager, which will call a different service to execute an operation into the database using the model. The result is written or read and added into the model. The view oversees rendering and present this data.

More accurately, sping-mvc is purposed as a solution. Why spring? There are several reasons but the most important are: easy injection of dependencies (lots of functionalities are handy such as validation, deployment in Heroku support, parsers…), easy to configure, intuitive. I will use a bootsrap template to make everything more fancy and intuitive to a final user (but frontend is not the focus of the app).

**Model**

We must define the OBJECT that builds up the concept of Contact. In the class Contact the attributes are defined, as well as the getters and setters. We can easily (only with tags like @NotBlank, @Email…) define some validations to be performed. This object can be mapped onto a table also or an XML file with tags (@Table, @XMLRootElemet)

We define a repository to access the database. The class contains the CRUD functions and the queries if they are not predefined. We have decided to use postgres for a basic reason: MySQL has no free addon in Heroku. It's important to create the table (initTableContact.sql)

**Controllers**

Whenever a URI is acceded, the controller implements the high-level logic. To understand how the logic is performed, the most complex case will be explained.

When creating a Contact, a POST method is called. Firstly, the form is automatically validated to check if the user must be informed about binding errors and return the same view. If not, the id (passed as a param of the URI {id}) is used to get the contact with that id. If no results, the service layer is called to create a contact with that data, otherwise the user is updated with the information of the form. In both cases all contacts are gotten and added to the model because they will be rendered afterwards.

**View**

The controller always finishes returning a path aiming to a JSP file. The objects included in the model as well as the whole content of the JSP are rendered and presented into the browser. It's important to define the commandName (the object, typically a form, will be passed to the controller in the submit)

**REST Service**

In the URI ws/contact an XML file can be consumed to create a contact. The idea is the same that explained above