# Naming Convention class

At the beginning of our work, we decided that we are going to need some kind of a naming convention interface that would be implemented by most of the classes in the project to prevent bugs, caused by spelling mistakes. Upon further consideration, however, we discovered that this interface was mostly going to be used in the top layer of our program to coordinate the interaction between the Servlet and the .jsp files, and that was a problem, because .jsp files cannot implement interfaces. They can, however, use the import statement to import classes. That is why we agreed that instead of an interface, we should use a plain old class for this purpose. A class, which serves as something like a pseudointerface. The only reason it remains placed in the ***layer2.domain.interfaces*** package, instead of being moved to the first layer is because, as I previously mentioned, it is heavily referenced in the majority of the .jsp files, and even if we use NetBeans’ refactor move option, we would still have to change all of the import statements in the .jsps by hand, and that is something that we cannot afford to do, due to time constraints.

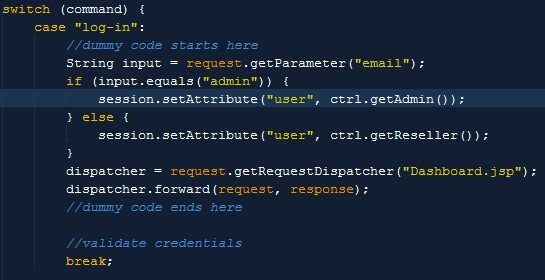
# Log-In and Security

Since we were going to have at least two different types of users, using the product, and since developing a log-in system that actually checks for someone's username and password seemed like a rather lengthy process, we decided that, at least until we're done with the more important user stories first, we would need some kind of a dummy log-in system that would easily allow us to log-in and test the system.

The idea I came up with is the following: In order for us to log-in as an admin (dell user), we would have to type "admin" in the email field *[\*Note: At this point, we were still using emails for logging-in, which we later replaced with a username]* and just some random characters in the password field. And in order for us to log-in as a partner of dell, we would just type random characters in both input fields and get in.

It works the following way. First, we manually populated the database with one tuple of each of the two different types of users we would have, via INSERT INTO queries. Then, we wrote the code, so that once the log-in button on the index page is pressed, the Servlet would only check for the email parameter. If it was “admin”, it would start the chain calling of the dummy ***getAdmin()*** methods which went through the Controller class all the way down to the dataSource layer. And if the email parameter was not “admin”, it would call the dummy ***getReseller()*** methods. Once in the dataSource layer (the DBFacade, to be more precise), the correct userID’s were hard-coded and passed as arguments for the mappers to use and retrieve the needed user.

\*Note: At this time, we were still using the “reseller” terminology for dell’s partners, which was later dropped.

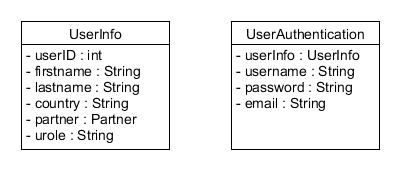


*Dummy Log-In [Note: At the time this screenshot was taken, we were still using the email for logging-in]*

One week before the project deadline, however, we determined that it was time to handle logging-in the proper way, so we dropped this dummy implementation. Now when a user tries to log-in, the following takes place:

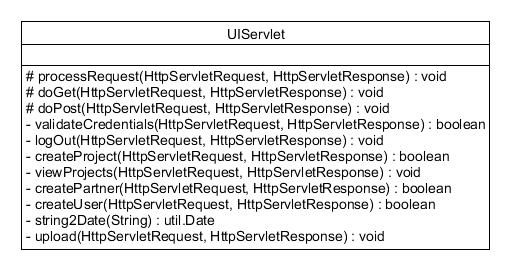
1. The program checks if a user with the entered username exists in the database.
2. If so, it checks if the entered password corresponds to the one stored for this user.
3. Then, if all went well, the log-in has been successful.

We are aware of the fact that this is in no way a secure log-in as we are breaking the first rule of password storing, which is never to store non-hashed values of passwords anywhere and that’s exactly what we’re doing (by storing them in the database). However, time constraints were once more the thing that made us decide not to implement a hashing algorithm for the passwords.

Nevertheless, one of the things we did concerning security was to split the user information into two different tables – one that stores the general information for the user and one that stores his log-in credentials and more sensitive information (e.g. email), called respectively UserInfo and UserAuthentication. That way, when we create an object of the type UserInfo and we use it throughout the program, we don’t carry around all the sensitive information with us.

# Servlet

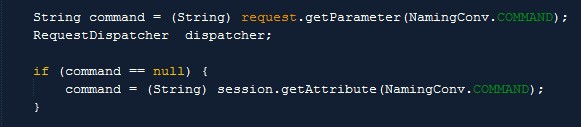
Since our application was not going to be too big, we decided to have one servlet that handles all requests. In this manner, we would have a centralized code-flow, while still keeping the code in the servlet manageable.



For the implementation, we relied on the examples, given to us before the project start. The first thing we do whenever a request is sent to the servlet and into the ***processRequest()*** method is to attempt to retrieve the Controller object from the session. If no such object exists, we create one and set it as an attribute of the session, so we can access it later as long as the same user is still logged-in. Later, we realized that we don’t need to do that because we don’t actually store any user-related information in the Controller, so we could just make a new one every time the ***processRequest()*** is called, but by the time we had this realization, we had already finished coding.



The next thing that happens is we try to retrieve the command attribute from the request object. If it’s not there, then we take it from the session, because we’ve constructed the .jsp pages in such a way that whenever a call to the Servlet is made, there is always a command attribute either in the request, or in the session.



Once we have the String command, we enter into a switch-case