15th homework; JAVA, Academic year 2019./2020.; FER

Introduction

For this homework you will integrate the structured web-application which is described in document "`13-uputa.txt`", Section 4, which is available in Ferko's repository with a JPA-based implementation of blog system. Let this new web application's name be *blog* so that it will be available using an URL such as http://localhost:8080/blog. Once you finish the application, you will prepare a ZIP archive of your eclipse project and upload it to Ferko (please note that in the document `13-uputa.txt` this aplication is named *webapp-bazaorm* so do not simply copy everything from this document but instead replace names where needed). Set project's *groupID* to `hr.fer.zemris.java.jmbag0000000000.webapps` (replace 0000000000 with your JMBAG), set *artefactID* to `hw15-0000000000`.

Problem 1.

Starting point for this homework is fully functional web-application described in `13-uputa.txt`, (but without 2nd-level caching – as I mentioned in lectures). You do not have to upload this application; your official homework starts with problem 2.

Problem 2.

As part of this problem you will implement a simple user-management functionality for your blog website. Start by creating a new web-application `/blog` which is based on application from problem 1, as described in Introduction.

Add new domain class `BlogUser` modeling a single user (place it into the same package as all other domain classes). For each blog user you should track the following properties: `id`, `firstName`, `lastName`, `nick`, `email` and `passwordHash`.

For example, some user can have `firstName="Pero"`, `lastName="Perić"`, `nick="perica"`, `email="pp@some.com"` and `passwordHash="22ffc727b1648e4ac073589d2659dec991918ec8"`. Property `passwordHash` is used for storing storing a hex-encoded hash value (calculated as SHA-1 hash) obtained from users password (you have already created a code for hashing binary data in one of your previous homeworks – search for `MessageDigest`). You are not allowed to store users' passwords in plain text into database since this would allow a database admin (and anyone who obtain the access to database) to easily see and steal users' passwords. Instead, during a user registration process you will:

1. ask a user to provide a `nick` and `password`,
2. `ep = calculate hexEncode(calcHash(password))`
3. store `ep` in database as `passwordHash`.

Also treat `nick` property as unique: no two users are allowed to have same nicks (set appropriate domain constraint; also check during the registration if user with given nickname already exists, and if it does, show user appropriate message and ask him to choose different nickname).

During a user's login process (handled by `/servleti/main` servlet, see diagram on the next page), you will:
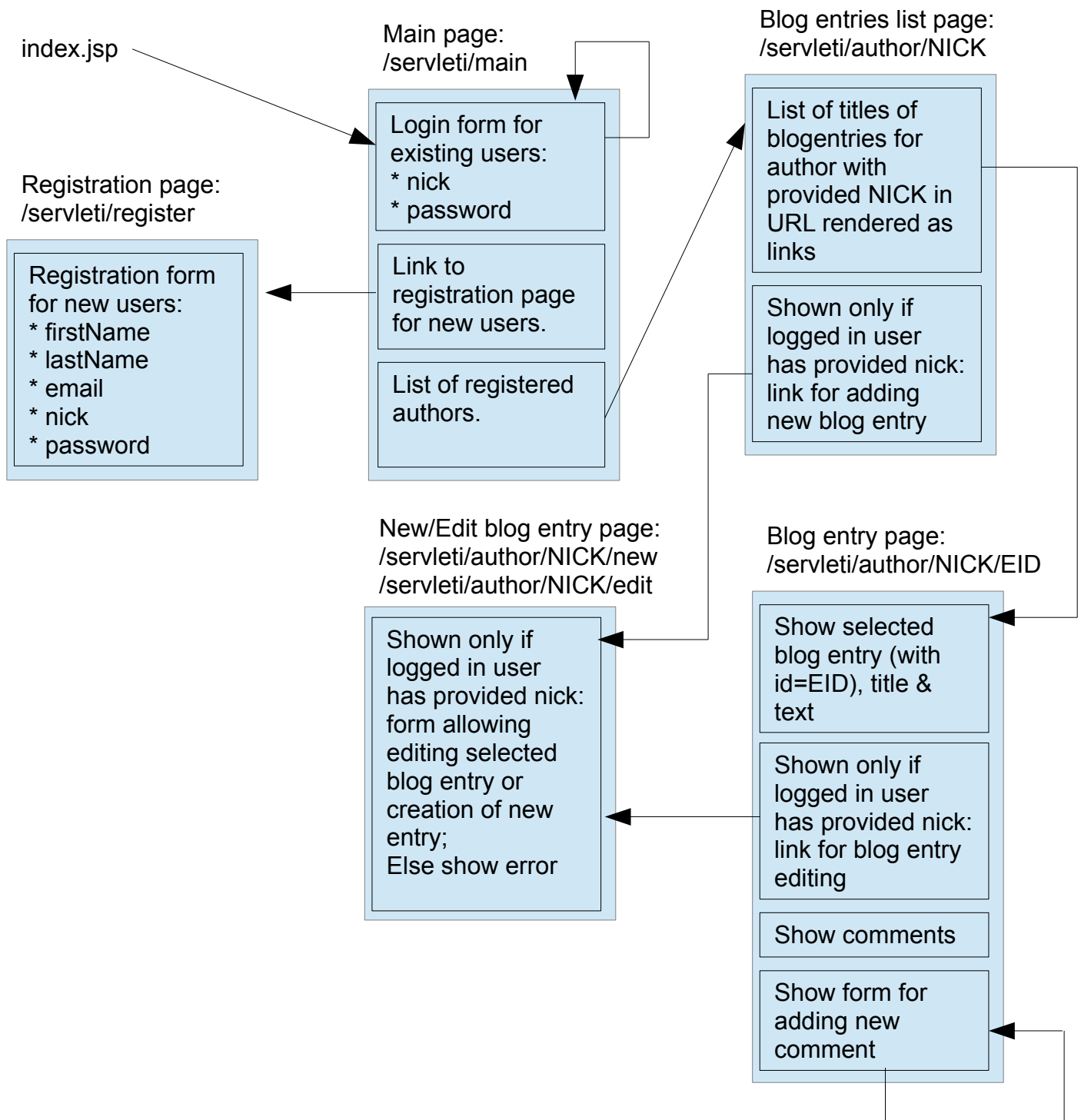
1. ask user to provide `nick` and `password` **and send it via POST method**,
2. calculate `ep = calculate hexEncode(calcHash(password))`,
3. lookup user in database with provided `nick`,
4. compare stored `passwordHash` and calculated `ep` for match.

If comparison does not match, display appropriate error message, and render login form again but **without** provided password (username which the user provided should be filled in the form automatically). **If password matches existing record in database, <u>redirect</u> user to appropriate page so that F5 (or refresh) in browser will not resubmit authentication request - try it.**
Modify domain class `BlogEntry`: add property `creator` which will reference the `BlogUser` that created the entry. Make that relation bidirectional.

Problem 3.

You will adjust existing code and implement whats missing to obtain a web application with page-flow as given on following diagram.

index.jsp

Registration page:
/servleti/register

Registration form
for new users:
* firstName
* lastName
* email
* nick
* password

Main page:
/servleti/main

Login form for
existing users:
* nick
* password

Link to
registration page
for new users.

List of registered
authors.

Blog entries list page:
/servleti/author/NICK

List of titles of
blogentries for
author with
provided NICK in
URL rendered as
links

Shown only if
logged in user
has provided nick:
link for adding
new blog entry

New/Edit blog entry page:
/servleti/author/NICK/new
/servleti/author/NICK/edit

Shown only if
logged in user
has provided nick:
form allowing
editing selected
blog entry or
creation of new
entry;
Else show error

Blog entry page:
/servleti/author/NICK/EID

Show selected
blog entry (with
id=EID), title &
text

Shown only if
logged in user
has provided nick:
link for blog entry
editing

Show comments

Show form for
adding new
comment

You should create a servlet that will be mapped on "`/index.jsp`" and that will send to a client a redirection to page `/servleti/main` (in your web application context, of course). For example, if your application is deployed as *blog*, writing http://localhost:8080/blog should **produce redirection** to http://localhost:8080/blog/servleti/main. For redirection see: https://tomcat.apache.org/tomcat-7.0-doc/servletapi/javax/servlet/http/HttpServletResponse.html#sendRedirect(java.lang.String))

For our demo user `perica`, requesting:

http://localhost:8080/blog/servleti/author/perica

should bring a page with titles (and links) of all of his blog entries, while requesting:

http://localhost:8080/blog/servleti/author/perica/5

should bring a page with blog entry with `id=5` (assuming that the creator of that entry is indeed `perica`) – if not, produce an error.

The general idea of our application is that all users: anonymous and logged-in should see exactly the same page structure. However, logged in users also see additional functionality: adding a new blog entry on his blog page and editing his blog entries.

Anonymous users can obtain an account by filling in registration form – no restrictions should apply beside the fact that two users can not have the same nick.

In previous picture only a rough structure is presented (with some examples of URLs); all that is missing is left to you to implement as you deem appropriate (including parameters, back links, etc).

In a case where you wish to map a servlet to a partial URL (for example, to any URL that starts by `/servleti/author` regardless of which path was provided after that), you can get information on actual URL that triggered the servlet (for example, `/servleti/author/perica`, `/servleti/author/perica/5`, `/servleti/author/perica/new` etc) using `HttpServletRequest` methods `getServletPath()` and `getPathInfo()`. Take a look at these methods and what they return.

http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getServletPath%28%29
http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getPathInfo%28%29

Handling of the login process

Please observe that information on users is now stored in our web applications database. That means that we alone will handle authentication and authorization. This is what you should do.

When user provides nick and password, you will check them and if user is valid you will store `BlogUser.id` into current session (use, for example, key "`current.user.id`"); additionally, store current user nick, first name and last name under keys "`current.user.fn`", "`current.user.ln`" and "`current.user.nick`".

Each action that needs to check if there is logged-in user will simply check if there is "`current.user.id`" in session map. If no, we are working with anonymous user that can only browse all blogs and blog entries and add comments. If there is such key stored, we have logged-in user whose other commonly-used information can also obtained from session map.

Handling of the logout process

You should add to main page also a logout link. Starting associated action should simply invalidate current session (see `HttpServletRequest.getSession().invalidate()`) and **send back redirection** to `/servleti/main` (just as servlet mapped to `/index.jsp` did).

http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html
http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html

Additional note:

In header of each rendered page (not in `<head>...</head>` of HTML itself but in "visual" header – top of rendered page) please write first name and last name of loged-in user or "not loged in", and provide link for logout (if user is loged-in).

Any graphical design (e.g. CSS styles) is optional. Also, you don't have to implement editing of users profile (e.g. allowing user to change fist name, last name, email or password).

Finally, anything that is not strictly prescribed in this document you are free to solve as you deem appropriate. However, please note that you are expected to create a high-quality code and an application that is layered and conceptually clear, just as we explained on lectures and in previous homework.

Also, it is expected that by default, `persistence.xml` is configured to use:
- **Url:** jdbc:derby://localhost:1527/blogBaza
- **Username:** blogDBAdmin
- **Password:** blogDBPassword

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else). You can use Java Collection Framework and other parts of Java covered by lectures; if unsure – e-mail me. Document your code!

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

You are not required to unit test code in this homework.

When your complete homework is done, pack it in zip archive with name `hw15-0000000000.zip` (replace zeros with your JMBAG); make sure there is **NO** `target` directory present. Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is June 29st 2020. at 11:59 PM.