CS142 Project #5: Forms, Sessions, and Validation

In this project you will extend your work on Project #4 by adding forms for logging in, commenting on existing photos, and uploading new photos. In order to implement these new features you will need to use the Rails facilities for sessions and validation. To get started, copy the directory tree for Project #4 to a new directory named `project5`. Do all of your work for this project in the new directory.

Problem 1: Simple Login (10 points)

Write a database migration that will add a new attribute `login` to the `User` model. This attribute is a string containing the identifier the user will type when logging in (their "login name"). Include code in your migration to initialize the `login` attribute for each existing user to the users's last name, converted to lower case.

Modify the application to support 3 new URLs:

- /users/login: displays a simple login form where the user can enter their login name (no passwords for now).
- `/users/post_login`: the login form posts to this URL. Its action method checks to ensure that there exists a user with the given login. If so, it stores the user id in the session where it can be checked by other code that needs to know whether a user is logged in. If there is no such user then the action must redisplay the login form with an appropriate error message. After a successful login you should redirect to the page displaying the user's photos.
- /users/logout: logs the user out by clearing the information stored in the session. This URL should redirect back to the login page.

In addition to implementing these URLs, include support for login/logout in the standard layout used for all of your application's pages. If there is no user logged in, the banner at the top of each page should include a small "Login" link to the login page. If there is a user logged in, the banner should include a small message "Hi Alice! Logout", where "Alice" is the first name of the logged-in user and "Logout" is a link to the logout page.

Problem 2: New Comments (10 points)

Once you have implemented user login, the next step is to implement a form for adding comments to existing photos. Implement a URL `/photos/`*`photo_id`*`/comments/new` that displays a form where a user can add a comment for the photo whose primary key is *photo_id*. You should also display the photo on this page so the user can see it while he/she is typing the comment. The form should post to the URL `/photos/`*`photo_id`*`/comments`; your implementation for this URL should create a new comment in the database using the Rails models. The comment must include the identifier of the logged in user and the time when the comment was created. Make sure that new comments can be viewed in the same way as pre-existing comments.

Once you've implemented the form for new comments, modify the page `/users/`*`id`* to display a "New Comment" link next to each photo, which will go to the new-comment form for that photo.

Your implementation must handle the following errors:

- If there is no user logged in then it should not be possible to add comments.
- Do not allow empty comments: return to the new-comment form and display an error message. Use the Rails validation mechanism to implement this.

Problem 3: Photo Uploading (10 points)

Allow users to add new photos. To do this, implement a URL `/photos/new`, which displays a form where the user can select a photo file for upload. The form should post to the URL `/photos`, which copies the incoming photo data to a file in the directory `project5/app/assets/images` and creates a new record in the database containing the name of the photo file, the creation time, and the identifier of the user. Also, add a "New Photo" link at an appropriate place in one of your existing pages, which users can click to go to the photo upload form.

Your implementation should check to make sure that a user is logged in and prevent photo uploading if not.

Problem 4: Registration and Passwords (15 points)

Enhance the login mechanism with support for new-user registration and passwords.

Your password mechanism must implement *salting*, which is a way of storing passwords securely. The salting mechanism is described in the next few paragraphs. One not-very-secure approach would be to store passwords directly in the database. However, if someone is able to read the database (for example, a rogue system administrator) they can easily retrieve all of the passwords for all users.

A better approach is to apply a *message digest* function such as SHA-2 (Standard Hash Algorithm 2) to each password, and store only the message digest in the database. SHA-2 takes a string such as a password as input and produces a string of hex digits (called a message digest, or cryptographic hash) as output. You can adjust how long this string is, either 256, 384, or 512 bits. You should use a 512 bit string.

The output has a two very useful and interesting properties. First, it is *collision resistant.* This means that, given an input I that generates a digest D, it's effectively impossible to come up with a different input, I2, that generates the same D. This is true whether you know the original input I or not. Second, it's a *one-way function*. Because a huge number of inputs can produce the same digest, you can't rewind the computation and produce an input that generates a digest. From a practical standpoint, this means that if you store the digest of someone's password in the database, then that doesn't help someone guess the password or come up with a new password that has the same digest.

When a user sets their password, you must use `Digest::SHA2` to generate the SHA-2 digest corresponding to that password, and store only the digest in the database; once this is done you can discard the password. When a user enters a password to login, use `Digest::SHA2` to compute the digest, and compare that digest to what is stored in the database. With this approach, you can make sure that a user types the correct password when logging in, but if someone reads the digests from the database they cannot use that information to log in.

However, the approach of the previous paragraph has one remaining flaw. Suppose an attacker gets a copy of the database containing the digests. Since the SHA-2 function is public, they can employ a fast *dictionary attack* to guess common passwords. To do this, the attacker takes each word from a dictionary of possible passwords and computes its digest using SHA-2. Then the attacker checks each digest in the database against the digests in the dictionary (this can be done very quickly by putting all of the dictionary digests in a hash table). If any user has chosen a simple dictionary word as their password, or even a short sequence of random characters, the attacker can guess it quickly.

In order to make dictionary attacks more difficult, you must use password salting. When a user sets their password, compute a large (more than 40 character) random string using `SecureRandom.hex` and concatenate it with the password before computing the SHA-2 digest (`SecureRandom.hex` will return a suitable random number). The random string is called a salt. Then store both the salt and the digest in the database. When checking passwords during login, retrieve the salt from the database, concatenate it to the password typed by the user, and compute the digest of this string for comparison with the digest in the database. With this approach an attacker who has gained access to the login database cannot use the simple dictionary attack described above; the digest of a dictionary word would need to include the salt for a particular account, which means that the attacker would need to recompute all of the dictionary digests for every distinct account in the database. This makes dictionary attacks more expensive.

This salting approach requires one small twist in your code. The password value that is posted in forms never gets stored in the database (so it is not specified in migrations, for example). In order to handle the password correctly, you must define it as a *virtual attribute* of the `User` model. This simply means that you must define by hand the two standard accessor methods `password` and `password=` in the user model (if you used a migration to create those accessors, they would automatically read and write from the database record; your methods will not do that). When `password=` is invoked, it must compute the salt and the password digest and set the corresponding attributes of the `User` model (which are part of the database).

- Add two new attributes to each user: `password_digest` and `salt`.
- Add a new method `password_valid?` to the `User` model. This method takes a candidate password as argument and returns a boolean value indicating whether the password is the correct one for the user corresponding to this model object.
- Add support for two new URLs: `/users/new` displays a form to register a new user, and it posts to `/users`. The registration page provides fields for the new user's first and last names, their login, plus two fields in which identical copies of the password must be typed. The post action must make sure that the new login doesn't already exist and that the two copies of the password are identical. If the information is

valid, then a new user gets created in the database and the action redirects to the login page. If there is an error then the registration form gets redisplayed along with appropriate error messages. Be sure to use the Rails validation mechanism.

- Add a password field to the login form and check it as part of the post action for the form.
- On the login form, add a link to the registration page.
- Initialize existing users to have a password that matches the value of their login attribute.

Style Points (5 points)

These points will be awarded if your problem solutions have proper MVC decomposition, follow the Rails conventions, and generate valid XHTML. In addition, your code and templates must be clean and readable, and your Web pages must be at least "reasonably nice" in appearance and convenience.

Hints

- If you add a new column to a table in a migration and then attempt to write information in that column in the same migration, Rails will ignore data in that column (it built its view of the table before the column was added). To get around this problem, you can invoke the `reset_column_information` method on the model class: this will force Rails to reexamine the database schema so that it notices any changes to the table structure.
- In order to upload files in a Rails form, you will need to do the following things (see the lecture notes for more information):
  - Use `form.file_field` to generate the form element.
  - When the form is posted, you can reference the uploaded file with the `params` hash: if the first argument to `form_for` was `:xyz` and the first argument to `form.file_field` was `:abc`, then the uploaded file will be available as `params[:xyz][:abc]`. This is an object of class ActionDispatch::Http::UploadedFile, which supports IO methods such as `read`. The object also provides a method `original_filename`, which returns the name of the file that was selected by the user in their browser.
- Do not store uploaded photos in the database; store them on disk in the directory `project5/app/assets/images`.
- The Ruby method `DateTime.now` returns a string containing the current date and time in the right format for storing in the database.
- You may find the Ruby method `getlocal` useful. When applied to a time value, it produces a corresponding time in the local timezone (by default, times are stored in Greenwich Mean Time). Note: the best approach is to store times in GMT and only convert them to local time when displaying for the user.
- If your database gets corrupted because of bugs, or if you'd just like to clear out the new comments and photos you have added, you can reset your database with the following steps:
  - Invoke the following shell command:

    ```
    rake db:migrate:reset
    ```

    This will delete the database for the project, re-create it , and rerun all migrations to bring the database up to date.
  - Go to the directory `project5/app/assets/images` and delete all of the new image files you have created, leaving only the original ones.

Deliverables

Use the standard class submission mechanism to submit the entire application (everything in the `project5` directory). Please indicate in a README file whether you developed on Windows or a Macintosh (we may need this information in order to test your solution). Please clean up your project directory before submitting, as described in the submission instructions.