Cybersecurity: Authentication
CSC340

The next few worksheets will consider cybersecurity in the context of the Microposter application. First, we need to add login functionality so that legitimate users can log in and work with the site. In this worksheet, we'll develop the authentication mechanism behind this functionality.

1. Before we start:
   ```
   cp –r microposter microposter2
   cd microposter2
   ```

2. We already have a User resource that we made with scaffolding previously. Recall that a user has name and email attributes. We'll need to make some changes to the User resource now to support authentication.

3. First, let's make sure that when a user is saved, the email address saved with the user is all lower case. That way we won't have any issues with case mixing up the authentication process. You can do this by putting the following in your User model:
   ```
   before_save { |user| user.email = user.email.downcase }
   ```

   a. `downcase` is a method defined in the string class that returns a copy of the string with all lower-case letters.

4. While we're here, let's also add some validations to the User model:
   a. Make sure the name is present and is no longer than 50 characters long.
   b. Make sure the email is present, is unique among users (ignoring case), and is of the proper *name@place.domain* form. Study and use the following validation code that accomplishes this:
      ```
      email_regex = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
      validates :email, :presence => true, :format => {:with => email_regex},
      :uniqueness => {:case_sensitive => false}
      ```
      We won't worry much for now about how to write regular expressions, but you can see the big picture of the code above.

5. It turns out, specifying uniqueness as a validation requirement as we did above is not sufficient to guarantee uniqueness of the email. So we'll need to enforce the uniqueness requirement at the database level as well. One way to do this is to create a database *index* on the email column, which can then be required to be unique. A database index is a column that can be efficiently searched on, as opposed to a "brute force" linear search.
   a. You can make the email column an index with uniqueness via a migration:

rails generate migration add_email_uniqueness_index

b. Open the migration file and modify it to look as follows:

```
class AddEmailUniquenessIndex < ActiveRecord::Migration
  def change
        add_index :users, :email, unique: true
  end
end
```

c. Apply the migration using rake.

d. With the application, create new users with invalid values and make sure the validation mechanisms stop you.

6. Here's the big picture plan for the rest of this worksheet: we'll add both a password and a password_confirmation attribute to the user model. Before a user can be saved to the database, password and password_confirmation must match (to reduce the possibility of typos). But these will be *virtual* attributes, meaning they're not stored in the database itself.

   In the database, we'll instead store password_digest. This is so that if the database is hacked, the hacker will only have access to the passwords in encrypted form, rather than plaintext, so the passwords will still be safe.

   To authenticate a user, the user will have to enter an email and password. The password will be encrypted using the SHA2 hash (a "one-way" function) and checked against the password_digest stored in the database.

7. First, reset the database to remove any users previously created:
   rake db:reset
   This step is needed since we're adding a new attribute to the model. You'll probably need to stop and restart the server as well.

8. Add the password_digest to the User table:
   rails generate migration add_password_digest_to_users password_digest:string

9. Check the generated migration and then apply it.

10. Add validations for password and password_confirmation that make sure they are present, and that their lengths are between 6 and 40 characters (using :length => {:within => 6..40}).

11. New as of Rails 3.2 is a has_secure_password method. It takes no arguments. Simply call it in the User model. It does what was specified as the "big picture plan" above, mostly automatically for us!

    a. To use has_secure_password, we first need to install the gem bcrypt, which provides encryption methods needed by has_secure_password. Edit your Gemfile accordingly, run bundle install, and restart the server.

b. has_secure_password also requires the presence of the password_digest attribute (which we already have) in order to work properly.
c. has_secure_password will automatically create validations for both the presence of the password, and that it matches password_confirmation.
d. has_secure_password also creates an `authenticate` method to compare the password_digest to the digested form of the password entered by the user.

12. We haven't updated our user creation process in the app itself yet, so to test all this so far, open up the console, make a user:

```
u1 = User.new(name: "Person Withapass",
     email:"pw@hmail.com", password: "123456",
     password_confirmation: "123456")
```

Then use the authenticate method on u1 twice: once passing it "hello", and once passing it "123456". Observe the results.

13. We have now implemented the backend for our authentication mechanism! We just need to update the website to make use of it to protect the website while allowing legitimate access. This is the topic of the next couple cybersecurity worksheets.