Authentication I: Creating Users

In this project, we will refer to the AuthVideoDemo project.

Phase I: bcrypt

Don't store passwords in the DB

Never store passwords in the database. This is bad, because if someone hacks your server and gains access to the database, they will be able to steal all of your user's passwords. Since your users probably use the same password across services, the hacker will be able to login to the user's gmail, facebook, etc.

What is a hash function?

The solution to this problem is to use a hash function. A hash function is a "one-way" function; it is easy to compute in one direction, but difficult to "invert". That means that, given a password, you can easily calculate the hash value of the password, but there is no easy way to go back and recover the password from the hash.

Basically, a hash is sort of like scrambling; scrambling is easier than unscrambling. The fancy mathematical trick (which we'll leave to the mathematicians), is to design a function where:

- You scramble things the same way every time: the same password will hash to the same value every time.
- That means the scrambling isn't really random; if the scrambling were totally random, you'd calculate a new, different hash every time.
- Even though the scrambling isn't truly random, it should still be difficult (that is, near impossible) to "unscramble".

Using a library

Let's use a library called `bcrypt` to do the hashing for us. First `gem install bcrypt`. Now let's play in the console:

```
require 'bcrypt'
# => true
password_hash = BCrypt::Password.create("my_secret_password")
# => "$2a$10$sl.3R32Paf64TqYRU3DBduNJkZMNBsbjyr8WIOdUi/s9TM4VmHNHO"
password_hash.is_password?("my_secret_password")
# => true
password_hash.is_password?("not_my_secret_password")
# => false
```

We use the `BCrypt::Password` class here; the `::create` factory method takes a password and hashes it. We store the resulting `BCrypt::Password` object in `password_hash`.

The `BCrypt::Password` class has an instance method, `#is_password?`. The method takes a string, hashes the string, and compares it with the `BCrypt::Password` contents.

If the argument is the same string as before, the two hashes will be equal and `#is_password?` returns true; otherwise false is returned.

Requiring `bcrypt`

Edit your `Gemfile`; add `gem 'bcrypt'`.

Phase II: `User` model

Initial columns and indices

Let's start with `rails g model user`. Add string columns to the `users` table: `username` and `password_digest`. Digest is another name for a hash. Toss on an index for `username` too (make it unique).

Note that we will not store the password itself in the DB; we're going to store the hashed version.

Storing/verifying a password

Let's create a `User` object:

```
u = User.new
# => #<User id: nil, username: nil, password_digest: nil, created_at: nil, updated_at: nil>
u.username = "earl"
# => "earl"
u.password_digest = BCrypt::Password.create("i_love_breakfast")
# => "$2a$10$oO6LUi.ikUl7rloGcZ.NFeURc0pNQhQA9MTaB89XX/kDNm.3vQVVu"
u.save
#   (0.3ms)  BEGIN
#  SQL (108.5ms)  INSERT INTO "users" ("created_at", "password_digest", "updated_at", "username") VALUES ($
#   (0.5ms)  COMMIT
# => true
u.password_digest
# => "$2a$10$oO6LUi.ikUl7rloGcZ.NFeURc0pNQhQA9MTaB89XX/kDNm.3vQVVu"
```

Let's see how to verify a password later:

```
u = User.first
#  User Load (0.7ms)  SELECT "users".* FROM "users" LIMIT 1
# => #<User id: 1, username: "earl", password_digest: "$2a$10$oO6LUi.ikUl7rloGcZ.NFeURc0pNQhQA9MTaB89XX/kD.
BCrypt::Password.new(u.password_digest).is_password?("i_love_breakfast")
# => true
```

Okay, cool. Notice that when we pull down the `User`, `password_digest` is set to a string, since that is what the DB stores. We want to get a `BCrypt::Password` object back from the digest. Because the digest is already hashed, we use the `new` constructor rather than the `create` factory method; `create` creates a `Password` object by hashing the input, while `new` builds a `Password` object from an existing, stringified hash.

Anyway all this code is not very convenient. Let's add some helper-methods to `User`.

Write `User#password=` and `User#is_password?`

Let's first write a method that will make it easier to set the `password_digest` column; right now, the programmer is required to hash the password themselves. Let's do it for them:

```
class User < ActiveRecord::Base
  def password=(password)
    self.password_digest = BCrypt::Password.create(password)
  end
end
```

The fact that we don't have a `password` column in the DB does not prevent us from writing a setter method named `password=`. This is totally cool. In this case, instead of saving the actual password, we're going to use BCrypt to hash it and then save it to the `password_digest` column.

Likewise, let's save the programmer from having to do the hard work of verifying a password:

```
class User < ActiveRecord::Base
  def password=(password)
    self.password_digest = BCrypt::Password.create(password)
  end

  def is_password?(password)
    BCrypt::Password.new(self.password_digest).is_password?(password)
  end
end
```

Let's test it out!

```
u = User.new
# => #<User id: nil, username: nil, password_digest: nil, created_at: nil, updated_at: nil>
u.username = "Houdini"
# => "Houdini"
u.password = "i_remember_kiki"
# => "i_remember_kiki"
u.save
#   (0.3ms)  BEGIN
#  SQL (4.8ms)  INSERT INTO "users" ("created_at", "password_digest", "updated_at", "username") VALUES ($1,
#   (2.6ms)  COMMIT
# => true
u = User.last
#  User Load (0.8ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" DESC LIMIT 1
# => #<User id: 2, username: "Houdini", password_digest: "$2a$10$.cMnzIMCgh/VUZ1OF3dJUOf1zJSRBw2t6YMAcKeuIb
u.is_password?("i_remember_kiki")
# => true
u.is_password?("random_password_guess")
# => false
```

Whoa, object orientation for the win! Hey, did you know it's totally cool to write methods on your model objects? I grant thee the permission! Go forth, and multiply methods!

Phase III: Creating new users

Let's add a `UsersController` and a `users` resource. Let's add a `new` view so that the user can sign up for the site.

```
# app/controllers/users_controller.rb
class UsersController < ApplicationController
  def create
    @user = User.new(user_params)

    if @user.save
      render json: @user
    else
      render json: @user.errors.full_messages
    end
  end
```

```
  def new
    @user = User.new
  end

  private

  def user_params
    params.require(:user).permit(:username, :password)
  end
end
```

Notice that we are calling `#permit` with `:password`, and *not* with `password_digest`. Is this ok? *Yes*: Just like in your Active Record Lite project, mass-assignment will call the setter method `password=`. It is totally okay that this doesn't set a password column.

Then our views:

```
<!-- app/views/users/new.html.erb -->
<h1>Create User</h1>

<%= render "form", user: @user %>

<!-- app/views/users/_form.html.erb -->
<% action = (user.persisted? ? user_url(user) : users_url) %>
<% method = (user.persisted? ? "patch" : "post") %>
<% message = (user.persisted? ? "Update user" : "Create user") %>

<form action="<%= action %>" method="post">
  <input
     name="_method"
     type="hidden"
     value="<%= method %>">
  <input
     name="authenticity_token"
     type="hidden"
     value="<%= form_authenticity_token %>">

  <label for="user_username">Username</label>
  <input
     id="user_username"
     name="user[username]"
     type="text">
  <br>

  <label for="user_password">Password</label>
  <input
     id="user_password"
     name="user[password]"
     type="password">
  <br>

  <input type="submit" value="<%= message %>">
</form>
```

Validating `User`

Okay, we can now create users through the form. We probably want to toss some validations on:

```
class User < ActiveRecord::Base
  validates :username, presence: true
  validates :password_digest, presence: { message: "Password can't be blank" }

  # ...
end
```

Notice that I set my own message to use if the `password_digest` is blank. The default is "Password digest can't be blank", but I don't like that because the user won't know what a password digest is.

What if we want to validate the length of the password? Right now, we never store the password in the `User` object; for that reason, we won't still have the password to check when we eventually validate the `User`. Does it make sense why we can't just slap on `validates :password, length: { minimum: 6 }`?

Here's one way to accomplish our goal: store the password in an instance variable, but never save this to the DB. First let's modify `#password=`:

```
  def password=(password)
    @password = password
    self.password_digest = BCrypt::Password.create(password)
  end
```

This saves the password in an instance variable. ActiveRecord will not try to save the password to the DB, however. Instead, the `@password` instance variable will be ignored.

We can now validate the password:

```
class User < ActiveRecord::Base
  attr_reader :password

  validates :username, presence: true
  validates :password_digest, presence: { message: "Password can't be blank" }
  validates :password, length: { minimum: 6, allow_nil: true }

  # ...
end
```

Couple of notes. I added the `#password` reader method; the validation will call this to check the attribute. Again, validations do not need to check only database columns; you can apply a validation to any attribute.

I also added `allow_nil: true`. This means the validation will pass if `@password` is `nil`. This is desirable, because the `@password` attribute is only set if we change the password with **`#password=`**.

Let me give an example:

```
User.create!(username: "houdini", password: "password")
#   (0.3ms)  BEGIN
#  SQL (0.6ms)  INSERT INTO "users" ("created_at", "password_digest", "updated_at", "username") VALUES ($1,
#   (2.0ms)  COMMIT
# => #<User id: 6, username: "houdini", password_digest: "$2a$10$88gQuHB0WxPa//tsI6pB4.xwrMWFGdtjnoMfSSfzgp
u = User.last
#  User Load (0.5ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" DESC LIMIT 1
# => #<User id: 6, username: "houdini", password_digest: "$2a$10$88gQuHB0WxPa//tsI6pB4.xwrMWFGdtjnoMfSSfzgp
u.password
# => nil
u.password_digest
# => "$2a$10$88gQuHB0WxPa//tsI6pB4.xwrMWFGdtjnoMfSSfzgpzp5xIiQhM.6"
```

Notice that after we fetch Houdini back from the DB, the password is no longer set. That's because this attribute only lived in an instance variable that was never going to get persisted to the DB.

Let's try one more thing:

```
u.valid?
# => true
```

If we didn't `allow_nil: true`, then Houdini would be marked as invalid, because a `nil` `password` attribute would not meet the length requirement. In reality, we only need to validate the password when it has been changed, which

is exactly when `@password` is not `nil`.

Authentication II: Creating Sessions

In this project, we will refer to the AuthVideoDemo project. We pick up from the previous work.

Overview

We are now successfully creating users. Let's start logging them in, too.

We're going to create a (singular) resource named `session`. When the user logs in, they'll go to `/session/new` to fill in their credentials. POSTing to `/session` (to the `SessionsController#create` action) will sign the user in; we'll store a session token in the cookie so that subsequent requests will know which user is logged in. Lastly, to log out, the client can issue a DELETE request to `/session`.

A couple notes:

- There is no `Session` model class, nor `sessions` table. This doesn't mean we can't write a `session` resource.
- The `session` resource is singular because the user will only use at most one session: their own.
- We'll write a `SessionsController`; even when using a singular resource, controllers are always pluralized.

Phase IV: Logging in: verifying credentials

Adding a `session` resource, `SessionsController`

Add a `session` resource to the routes file. Generate a `SessionsController`. Write a `new` form so that the user can fill out their username/password:

```
<!-- app/views/sessions/new.html.erb -->

<h1>Log in!</h1>

<form action="/session" method="POST">
  <input
    type="hidden"
    name="authenticity_token"
    value="<%= form_authenticity_token %>">

  <label for="user_username">Username</label>
  <input type="text" name="user[username]" id="user_username">
  <br>

  <label for="user_password">Password</label>
  <input type="password" name="user[password]" id="user_password">
```

```
  <br>

  <input type="submit" value="Log in!">
</form>
```

Remember to include the form authenticity token!

Okay; but what about the create action?

Adding `::find_by_credentials`

Okay, we need to write `SessionsController#create` so that it verifies the username/password and then builds/sets a session token. Let's add a `find_by_credentials` method:

```
class User < ActiveRecord::Base
  # ...

  def self.find_by_credentials(username, password)
    user = User.find_by_username(username)
    return nil if user.nil?
    user.is_password?(password) ? user : nil
  end

  # ...
end
```

This will only return a user if the username/password is correct.

Let's try this out in our controller:

```
class SessionsController < ApplicationController
  def create
    user = User.find_by_credentials(
      params[:user][:username],
      params[:user][:password]
    )

    if user.nil?
      render json: "Credentials were wrong"
    else
      render json: "Welcome back #{user.username}!"
    end
  end

  def new
  end
end
```

Cool! But this is useless, because the sessions controller doesn't set any cookie to remember the logged-in user.

Phase V: Logging in: setting the session

We now have the `SessionsController` properly verifying the user's credentials, but it doesn't actually do anything with the successful login. Let's fix that.

Add a `UsersController#show` page

Make a very simple `show` view for your `UsersController`.

```
<!-- app/views/users/show.html.erb -->
<h1><%= @user.username %></h1>
```

```
<p>Hello, dear user!</p>
```

Let's edit the `#create` method in `UsersController` and `SessionsController` to redirect to this page on successful form submission.

Let's begin making the show page private; a user should be the only one allowed to look at their own `users#show` page. But before we start, we need to start setting a session token...

Adding `session_token` to `User`

We're going to need to introduce a `session_token` so that after login we can remember the current user. Let's write a migration:

```
class AddSessionTokenToUsers < ActiveRecord::Migration
  def change
    add_column :users, :session_token, :string, null: false
    add_index :users, :session_token, unique: true
  end
end
```

Next, let's make sure that when we create a `User` it has a `session_token`:

```
class User < ActiveRecord::Base
  # ...

  validates :session_token, presence: true
  after_initialize :ensure_session_token

  def self.generate_session_token
    SecureRandom::urlsafe_base64(16)
  end

  # ...

  def reset_session_token!
    self.session_token = self.class.generate_session_token
    self.save!
    self.session_token
  end

  private
  def ensure_session_token
    # we must be sure to use the ||= operator instead of = or ||, otherwise
    # we will end up with a new session token every time we create
    # a new instance of the User class. This includes finding it in the DB!
    self.session_token ||= self.class.generate_session_token
  end
end
```

Note that the `User` now validates the presence of the `session_token`. Note that we'll conveniently generate a session token for the `User` *if one isn't already set* (that's the `after_initialize` callback bit).

Setting the session

We're now storing the session token in the `User`, but we need to also store it in the `session`. Let's write helper methods in the `ApplicationController`.

```
class ApplicationController < ActionController::Base
  def login!(user)
    @current_user = user
    session[:session_token] = user.session_token
  end
```

```
  def current_user
    return nil if session[:session_token].nil?
    @current_user ||= User.find_by_session_token(session[:session_token])
  end
end
```

To be able to use these helper methods throughout any view, we should use `helper_method`.

```
class ApplicationController < ActionController::Base
  helper_method :current_user
  # ...
end
```

This will make `current_user` available in all views.

And let's update our `SessionsController` and `UsersController` create methods:

```
class SessionsController < ApplicationController
  def create
    user = User.find_by_credentials(
      params[:user][:username],
      params[:user][:password]
    )

    if user.nil?
      render json: "Credentials were wrong"
    else
      login!(user)
      redirect_to user_url(user)
    end
  end

  # ...
end

class UsersController < ApplicationController
  def create
    @user = User.new(params[:user])

    if @user.save
      login!(@user)
      redirect_to user_url(@user)
    else
      render json: @user.errors.full_messages
    end
  end

  # ...
end
```

Phase VI: Using the `current_user`

Adding a logout button

We haven't written any logout functionality, nor do we ever tell the user who they are logged in as. Let's fix that by editing the application's layout:

```
<!-- app/views/layouts/application.html.erb -->

<!-- ... -->
<% if !current_user.nil? %>
  <ul>
```

```
    <li>Logged in as: <%= current_user.username %></li>
    <li>
      <form action="<%= session_url %>" method="POST">
        <input type="hidden"
               name="authenticity_token"
               value="<%= form_authenticity_token %>">
        <input type="hidden" value="delete" name="_method" />
        <input type="submit" value="Logout" />
      </form>
    </li>
  </ul>
<% end %>

<%= yield %>

</body>
</html>
```

Now, after we log in, we should be able to see the logout button. But we don't have a destroy action yet. Let's write one:

```
class ApplicationController < ActionController::Base
  # ...

  def logout!
    current_user.try(:reset_session_token!)
    session[:session_token] = nil
  end
end

class SessionsController < ApplicationController
  # ...

  def destroy
    logout!
    redirect_to new_session_url
  end

  # ...
end
```

Now sign out should be working. Notice that in `logout!` we reset the session token. This will invalidate the old session token. We want to do that in case anyone has managed to steal the token; this will deny the thieves further access to the account.

Adding a `before_action` callback

Let's finally finish what we started: let's protect the `users#show` page so that only the user themselves can view their own show.

To do that, let's write a `require_current_user!` helper:

```
class ApplicationController < ActionController::Base
  # ...

  def require_current_user!
    redirect_to new_session_url if current_user.nil?
  end
end
```

We can then add this as a `before_action` callback in our controllers:

```
class UsersController < ApplicationController
  before_action :require_current_user!, except: [:create, :new]
```

```
  # ...
end
```

Notice that I except the `create` and `new` actions; those are needed before a user is signed up, so there's no way for them to be logged in at that point.

A rule of `before_action` is that, if in the process of running a filter `redirect_to` is called, the action method will not be called. For instance, if a user tries to visit the `users#show` page without having logged in first, the callback will issue a redirect, and Rails will forgo calling the `UsersController#show` method.

Wait one more second

The callback we defined requires that a user be logged in to look at a `users#show` page. However, it does not enforce that user A may not look at user B's show page. Write a new filter in the `UsersController` to do this.