Chapter 9 Advanced login

The basic login system developed in Chapter 8 is fully functional, but most modern websites include the ability to "remember" users when they visit the site again even if they've closed their browsers in the interim. In this chapter, we use *permanent cookies* to implement this behavior. We'll start by automatically remembering users when they log in (Section 9.1), a common model used by sites such as Bitbucket and GitHub. We'll then add the ability to *optionally* remember users using a "remember me" checkbox, a model used by sites such as Twitter and Facebook.

Because the Chapter 8 login system is complete by itself, the core of the sample application will work fine without it, and if desired you can skip right to Chapter 10 (and from there to Chapter 13). On the other hand, learning how to implement the "remember me" feature is both highly instructive by itself and lays an essential foundation for account activation (Chapter 11) and password reset (Chapter 12). Moreover, the result is an outstanding example of computer magic: You've seen a billion of these "remember me" login forms on the Web, and now's your chance to learn how to make one.

9.1 Remember me

In this section, we'll add the ability to remember our users' login state even after they close and reopen their browsers. This "remember me" behavior will happen automatically, and users will automatically stay logged in until they explicitly log out. As we'll see, the resulting machinery will make it easy to add an optional "remember me" checkbox as well (Section 9.2).

As usual, I suggest switching to a topic branch before proceeding:

```
$ git checkout -b advanced-login
```

9.1.1 Remember token and digest

In Section 8.2, we used the Rails `session` method to store the user's id, but this information disappears when the user closes their browser. In this section, we'll take the first step toward persistent sessions by generating a *remember token* appropriate for creating permanent cookies using the `cookies` method, together with a secure *remember digest* for authenticating those tokens.

As noted in Section 8.2.1, information stored using `session` is automatically secure, but this is not the case with information stored using `cookies`. In particular, persistent cookies are vulnerable to session hijacking, in which an attacker uses a stolen remember token to log in as a particular user. There are four main ways to steal cookies: (1) using a packet sniffer to detect cookies being passed over insecure networks,[1] (2) compromising a database containing remember tokens, (3) using cross-site scripting (XSS), and (4) gaining physical access to a machine with a logged-in user. We prevented the first problem in Section 7.5 by using Secure Sockets Layer (SSL) site-wide, which protects network data from packet sniffers. We'll prevent the second problem by storing a hash digest of the remember tokens instead of the token itself, in much the same way that we stored password digests instead of raw passwords in Section 6.3.[2] Rails automatically prevents the third problem by escaping any content inserted into view templates. Finally, although there's no iron-clad way to stop attackers who have physical access to a logged-in computer, we'll minimize the fourth problem by changing tokens every time a user logs out and by taking care to *cryptographically sign* any potentially sensitive information we place on the browser.

With these design and security considerations in mind, our plan for creating persistent sessions appears as follows:

1. Create a random string of digits for use as a remember token.
2. Place the token in the browser cookies with an expiration date far in the future.
3. Save the hash digest of the token to the database.
4. Place an encrypted version of the user's id in the browser cookies.
5. When presented with a cookie containing a persistent user id, find the user in the database using the given id, and verify that the remember token cookie matches the associated hash digest from the database.

Note how similar the final step is to logging a user in, where we retrieve the user by email address and then verify (using the `authenticate` method) that the submitted password matches the password digest (Listing 8.7). As a result, our implementation will parallel aspects of `has_secure_password`.

We'll start by adding the required `remember_digest` attribute to the User model, as shown in Figure 9.1.

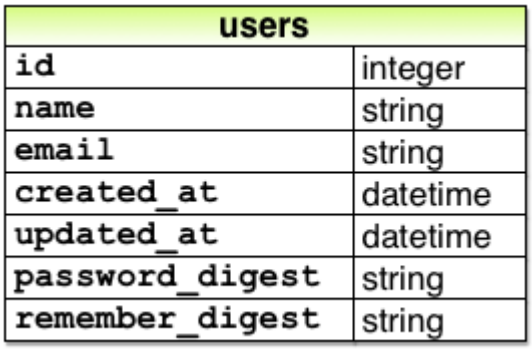

Figure 9.1: The User model with an added `remember_digest` attribute.

To add the data model from Figure 9.1 to our application, we'll generate a migration:

```
$ rails generate migration add_remember_digest_to_users remember_digest:string
```

(Compare to the password digest migration in Section 6.3.1.) As in previous migrations, we've used a migration name that ends in `_to_users` to tell Rails that the migration is designed to alter the `users` table in the database. Because we also included the attribute (`remember_digest`) and type (`string`), Rails generates a default migration for us, as shown in Listing 9.1.

Listing 9.1: The generated migration for the remember digest.
`db/migrate/[timestamp]_add_remember_digest_to_users.rb`

```
class AddRememberDigestToUsers < ActiveRecord::Migration[5.0]
                          def change
          add_column :users, :remember_digest, :string
                           end
                         end
```

Because we don't expect to retrieve users by remember digest, there's no need to put an index on the `remember_digest` column, and we can use the default migration as generated above:

```
$ rails db:migrate
```

Now we have to decide what to use as a remember token. There are many mostly equivalent possibilities—essentially, any long random string will do. The `urlsafe_base64` method from the `SecureRandom` module in the Ruby standard library fits the bill:[3] it returns a random string of length 22 composed of the characters A–Z, a–z, 0–9, "-", and "_" (for a total of 64 possibilities, thus "base64"). A typical base64 string appears as follows:

```
$ rails console
>> SecureRandom.urlsafe_base64
=> "q5lt38hQDc_959PVoo6b7A"
```

Just as it's perfectly fine if two users have the same password,[4] there's no need for remember tokens to be unique, but it's more secure if they are.[5] In the case of the base64 string above, each of the 22 characters has 64 possibilities, so the probability of two remember tokens colliding is a negligibly small $1/64^{22} = 2^{-132} \approx 10^{-40}$ 1/6422=2−132≈10−40.[6] As a bonus, by using base64 strings specifically designed to be safe in URLs (as indicated by the name `urlsafe_base64`), we'll be able to use the same token generator to make account activation and password reset links in Chapter 12.

Remembering users involves creating a remember token and saving the digest of the token to the database. We've already defined a `digest` method for use in the test fixtures (Listing 8.21), and we can use the results of the discussion above to create a `new_token` method to create a new token. As with `digest`, the new token method doesn't need a user object, so we'll make it a class method.[7] The result is the User model shown in Listing 9.2.

Listing 9.2: Adding a method for generating tokens. `app/models/user.rb`

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
```

```
                                has_secure_password
      validates :password, presence: true, length: { minimum: 6 }

                    # Returns the hash digest of the given string.
                                def User.digest(string)
      cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                    BCrypt::Engine.cost
                        BCrypt::Password.create(string, cost: cost)
                                end

                               # Returns a random token.
                                   def User.new_token
                                SecureRandom.urlsafe_base64
                                end
                               end
```

Our plan for the implementation is to make a `user.remember` method that associates a remember token with the user and saves the corresponding remember digest to the database. Because of the migration in Listing 9.1, the User model already has a `remember_digest` attribute, but it doesn't yet have a `remember_token` attribute. We need a way to make a token available via `user.remember_token` (for storage in the cookies) *without* storing it in the database. We solved a similar issue with secure passwords in Section 6.3, which paired a virtual `password` attribute with a secure `password_digest` attribute in the database. In that case, the virtual `password` attribute was created automatically by `has_secure_password`, but we'll have to write the code for a `remember_token` ourselves. The way to do this is to use `attr_accessor` to create an accessible attribute, which we saw before in Section 4.4.5:

```
                          class User < ApplicationRecord
                            attr_accessor :remember_token
                                        .
                                        .
                                        .
                                  def remember
                               self.remember_token = ...
                          update_attribute(:remember_digest, ...)
                                end
                               end
```

Note the form of the assignment in the first line of the `remember` method. Because of the way Ruby handles assignments inside objects, without `self` the assignment would create a *local* variable called `remember_token`, which isn't what we want. Using `self` ensures that assignment sets the user's `remember_token` attribute. (Now you know why the `before_save` callback from Listing 6.32 uses `self.email` instead of just `email`.) Meanwhile, the second line of `remember` uses the `update_attribute` method to update the remember digest. (As noted in Section 6.1.5, this method bypasses the validations, which is necessary in this case because we don't have access to the user's password or confirmation.)

With these considerations in mind, we can create a valid token and associated digest by first making a new remember token using `User.new_token`, and then updating the remember digest with the result of applying `User.digest`. This procedure gives the `remember` method shown in Listing 9.3.

Listing 9.3: Adding a `remember` method to the User model. **green** `app/models/user.rb`

```
                            class User < ApplicationRecord
                              attr_accessor :remember_token
                      before_save { self.email = email.downcase }
               validates :name,  presence: true, length: { maximum: 50 }
                VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
              validates :email, presence: true, length: { maximum: 255 },
                                  format: { with: VALID_EMAIL_REGEX },
                                  uniqueness: { case_sensitive: false }
                                  has_secure_password
      validates :password, presence: true, length: { minimum: 6 }

                    # Returns the hash digest of the given string.
                                def User.digest(string)
      cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                    BCrypt::Engine.cost
                        BCrypt::Password.create(string, cost: cost)
                                end

                               # Returns a random token.
                                   def User.new_token
```

```
    SecureRandom.urlsafe_base64
  end

  # Remembers a user in the database for use in persistent sessions.
  def remember
    self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
  end
end
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. In the console, assign `user` to the first user in the database, and verify by calling it directly that the `remember` method works. How do `remember_token` and `remember_digest` compare?
2. In Listing 9.3, we defined the new token and digest class methods by explicitly prefixing them with `User`. This works fine and, because they are actually *called* using `User.new_token` and `User.digest`, it is probably the clearest way to define them. But there are two perhaps more idiomatically correct ways to define class methods, one slightly confusing and one extremely confusing. By running the test suite, verify that the implementations in Listing 9.4 (slightly confusing) and Listing 9.5 (extremely confusing) are correct. (Note that, in the context of Listing 9.4 and Listing 9.5, `self` is the `User` class, whereas the other uses of `self` in the User model refer to a user object *instance*. This is part of what makes them confusing.)

Listing 9.4: Defining the new token and digest methods using `self`. **green** `app/models/user.rb`

```
class User < ApplicationRecord
  .
  .
  .
  # Returns the hash digest of the given string.
  def self.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def self.new_token
    SecureRandom.urlsafe_base64
  end
  .
  .
  .
end
```

Listing 9.5: Defining the new token and digest methods using `class << self`. **green** `app/models/user.rb`

```
class User < ApplicationRecord
  .
  .
  .
  class << self
    # Returns the hash digest of the given string.
    def digest(string)
      cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                    BCrypt::Engine.cost
      BCrypt::Password.create(string, cost: cost)
    end

    # Returns a random token.
    def new_token
      SecureRandom.urlsafe_base64
    end
  end
  .
  .
  .
```

9.1.2 Login with remembering

Having created a working `user.remember` method, we're now in a position to create a persistent session by storing a user's (encrypted) id and remember token as permanent cookies on the browser. The way to do this is with the `cookies` method, which (as with `session`) we can treat as a hash. A cookie consists of two pieces of information, a `value` and an optional `expires` date. For example, we could make a persistent session by creating a cookie with value equal to the remember token that expires 20 years from now:

```
cookies[:remember_token] = { value:   remember_token,
                             expires: 20.years.from_now.utc }
```

(This uses one of the convenient Rails time helpers, as discussed in Box 9.1.) This pattern of setting a cookie that expires 20 years in the future is so common that Rails has a special `permanent` method to implement it, so that we can simply write

```
cookies.permanent[:remember_token] = remember_token
```

This causes Rails to set the expiration to `20.years.from_now` automatically.

Box 9.1. Cookies expire `20.years.from_now`

You may recall from Section 4.4.2 that Ruby lets you add methods to *any* class, even built-in ones. In that section, we added a `palindrome?` method to the `String` class (and discovered as a result that `"deified"` is a palindrome), and we also saw how Rails adds a `blank?` method to class `Object` (so that `"".blank?`, `" ".blank?`, and `nil.blank?` are all `true`). The `cookies.permanent` method, which creates "permanent" cookies with an expiration `20.years.from_now`, gives yet another example of this practice through one of Rails' *time helpers*, which are methods added to `Fixnum` (the base class for integers):

```
$ rails console
>> 1.year.from_now
=> Wed, 21 Jun 2017 19:36:29 UTC +00:00
>> 10.weeks.ago
=> Tue, 12 Apr 2016 19:36:44 UTC +00:00
```

Rails adds other helpers, too:

```
>> 1.kilobyte
   => 1024
>> 5.megabytes
  => 5242880
```

These are useful for upload validations, making it easy to restrict, say, image uploads to `5.megabytes`.

Although it should be used with caution, the flexibility to add methods to built-in classes allows for extraordinarily natural additions to plain Ruby. Indeed, much of the elegance of Rails ultimately derives from the malleability of the underlying Ruby language.

To store the user's id in the cookies, we could follow the pattern used with the `session` method (Listing 8.14) using something like

```
cookies[:user_id] = user.id
```

Because it places the id as plain text, this method exposes the form of the application's cookies and makes it easier for an attacker to compromise user accounts. To avoid this problem, we'll use a *signed* cookie, which securely encrypts the cookie before placing it on the browser:[8]

```
cookies.signed[:user_id] = user.id
```

Because we want the user id to be paired with the permanent remember token, we should make it permanent as well, which we can do by chaining the `signed` and `permanent` methods:

```
cookies.permanent.signed[:user_id] = user.id
```

After the cookies are set, on subsequent page views we can retrieve the user with code like

```
User.find_by(id: cookies.signed[:user_id])
```

where `cookies.signed[:user_id]` automatically decrypts the user id cookie. We can then use bcrypt to verify that `cookies[:remember_token]` matches the `remember_digest` generated in Listing 9.3. (In case you're wondering why we don't just use the signed user id, without the remember token, this would allow an attacker with possession

of the encrypted id to log in as the user in perpetuity. In the present design, an attacker with both cookies can log in as the user only until the user logs out.)

The final piece of the puzzle is to verify that a given remember token matches the user's remember digest, and in this context there are a couple of equivalent ways to use bcrypt to verify a match. If you look at the secure password source code, you'll find a comparison like this:[9]

```
BCrypt::Password.new(password_digest) == unencrypted_password
```

In our case, the analogous code would look like this:

```
BCrypt::Password.new(remember_digest) == remember_token
```

If you think about it, this code is really strange: it appears to be comparing a bcrypt password digest directly with a token, which would imply *decrypting* the digest in order to compare using ==. But the whole point of using bcrypt is for hashing to be irreversible, so this can't be right. Indeed, digging into the source code of the bcrypt gem verifies that the comparison operator == is being *redefined*, and under the hood the comparison above is equivalent to the following:

```
BCrypt::Password.new(remember_digest).is_password?(remember_token)
```

Instead of ==, this uses the boolean method `is_password?` to perform the comparison. Because its meaning is a little clearer, we'll prefer this second comparison form in the application code.

The above discussion suggests putting the digest–token comparison into an `authenticated?` method in the User model, which plays a role similar to that of the `authenticate` method provided by `has_secure_password` for authenticating a user (Listing 8.15). The implementation appears in Listing 9.6. (Although the `authenticated?` method in Listing 9.6 is tied specifically to the remember digest, it will turn out to be useful in other contexts as well, and we'll generalize it in Chapter 11.)

Listing 9.6: Adding an `authenticated?` method to the User model. `app/models/user.rb`

```
class User < ApplicationRecord
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def User.new_token
    SecureRandom.urlsafe_base64
  end

  # Remembers a user in the database for use in persistent sessions.
  def remember
    self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
  end

  # Returns true if the given token matches the digest.
  def authenticated?(remember_token)
    BCrypt::Password.new(remember_digest).is_password?(remember_token)
  end
end
```

Note that the `remember_token` argument in the `authenticated?` method defined in Listing 9.6 is not the same as the accessor that we defined in Listing 9.3 using `attr_accessor :remember_token`; instead, it is a variable local to the method. (Because the argument refers to the remember token, it is not uncommon to use a method argument

that has the same name.) Also note the use of the `remember_digest` attribute, which is the same as `self.remember_digest` and, like `name` and `email` in Chapter 6, is created automatically by Active Record based on the name of the corresponding database column (Listing 9.1).

We're now in a position to remember a logged-in user, which we'll do by adding a `remember` helper to go along with `log_in`, as shown in Listing 9.7.

Listing 9.7: Logging in and remembering a user. `app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

                            def new
                              end

                           def create
  user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
                            log_in user
                           remember user
                          redirect_to user
                               else
     flash.now[:danger] = 'Invalid email/password combination'
                            render 'new'
                                end
                              end

                           def destroy
                              log_out
                        redirect_to root_url
                              end
                            end
```

As with `log_in`, Listing 9.7 defers the real work to the Sessions helper, where we define a `remember` method that calls `user.remember`, thereby generating a remember token and saving its digest to the database. It then uses `cookies` to create permanent cookies for the user id and remember token as described above. The result appears in Listing 9.8.

Listing 9.8: Remembering the user. `app/helpers/sessions_helper.rb`

```
                        module SessionsHelper

                       # Logs in the given user.
                            def log_in(user)
                       session[:user_id] = user.id
                                  end

                    # Remembers a user in a persistent session.
                                def remember(user)
                                   user.remember
                     cookies.permanent.signed[:user_id] = user.id
                cookies.permanent[:remember_token] = user.remember_token
                                  end

                   # Returns the current logged-in user (if any).
                                def current_user
                  @current_user ||= User.find_by(id: session[:user_id])
                                  end

                # Returns true if the user is logged in, false otherwise.
                                 def logged_in?
                                !current_user.nil?
                                  end

                               # Logs out the current user.
                                  def log_out
                                session.delete(:user_id)
                                   @current_user = nil
                                  end
                                end
```

With the code in Listing 9.8, a user logging in will be remembered in the sense that their browser will get a valid remember token, but it doesn't yet do us any good because the `current_user` method defined in Listing 8.16 knows only about the temporary session:

```
@current_user ||= User.find_by(id: session[:user_id])
```

In the case of persistent sessions, we want to retrieve the user from the temporary session if `session[:user_id]` exists, but otherwise we should look for `cookies[:user_id]` to retrieve (and log in) the user corresponding to the persistent session. We can accomplish this as follows:

```
if session[:user_id]
  @current_user ||= User.find_by(id: session[:user_id])
elsif cookies.signed[:user_id]
  user = User.find_by(id: cookies.signed[:user_id])
  if user && user.authenticated?(cookies[:remember_token])
    log_in user
    @current_user = user
  end
end
```

(This follows the same `user && user.authenticated` pattern we saw in Listing 8.7.) The code above will work, but note the repeated use of both `session` and `cookies`. We can eliminate this duplication as follows:

```
if (user_id = session[:user_id])
  @current_user ||= User.find_by(id: user_id)
elsif (user_id = cookies.signed[:user_id])
  user = User.find_by(id: user_id)
  if user && user.authenticated?(cookies[:remember_token])
    log_in user
    @current_user = user
  end
end
```

This uses the common but potentially confusing construction

```
if (user_id = session[:user_id])
```

Despite appearances, this is *not* a comparison (which would use double-equals ==), but rather is an *assignment*. If you were to read it in words, you wouldn't say "If user id equals session of user id…", but rather something like "If session of user id exists (while setting user id to session of user id)…".[10]

Defining the `current_user` helper as discussed above leads to the implementation shown in Listing 9.9.

Listing 9.9: Updating `current_user` for persistent sessions. **red** `app/helpers/sessions_helper.rb`

```
module SessionsHelper

# Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

# Remembers a user in a persistent session.
  def remember(user)
    user.remember
    cookies.permanent.signed[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

# Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.signed[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
```

```
  # Returns true if the user is logged in, false otherwise.
  def logged_in?
    !current_user.nil?
  end

  # Logs out the current user.
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end
end
```

With the code as in Listing 9.9, newly logged in users are correctly remembered, as you can verify by logging in, closing the browser, and checking that you're still logged in when you restart the sample application and revisit the sample application.[11] If you want, you can even inspect the browser cookies to see the result directly (Figure 9.2).[12]

Name	remember_token
Value	vb4iQ7Oy3dCLv2R2TEdQ0g
Host	rails-tutorial-c9-mhartl.c9.io
Path	/
Expires	Sun, 30 Jul 2034 00:18:56 GMT
Secure	No
HttpOnly	No

Figure 9.2: The remember token cookie in the local browser.

There's only one problem with our application as it stands: short of clearing their browser cookies (or waiting 20 years), there's no way for users to log out. This is exactly the sort of thing our test suite should catch, and indeed it should currently be **red**:

Listing 9.10: **red**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. By finding the cookie in your local browser, verify that a remember token and encrypted user id are present after logging in.
2. At the console, verify directly that the `authenticated?` method defined in Listing 9.6 works correctly.

9.1.3 Forgetting users

To allow users to log out, we'll define methods to forget users in analogy with the ones to remember them. The resulting `user.forget` method just undoes `user.remember` by updating the remember digest with `nil`, as shown in Listing 9.11.

Listing 9.11: Adding a `forget` method to the User model. **red** `app/models/user.rb`

```
class User < ApplicationRecord
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
```

```
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
                    has_secure_password
  validates :password, presence: true, length: { minimum: 6 }

        # Returns the hash digest of the given string.
                  def User.digest(string)
cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                BCrypt::Engine.cost
          BCrypt::Password.create(string, cost: cost)
                                end

                      # Returns a random token.
                          def User.new_token
                      SecureRandom.urlsafe_base64
                                end

  # Remembers a user in the database for use in persistent sessions.
                              def remember
                  self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
                                end

         # Returns true if the given token matches the digest.
                  def authenticated?(remember_token)
   BCrypt::Password.new(remember_digest).is_password?(remember_token)
                                end

                          # Forgets a user.
                              def forget
                  update_attribute(:remember_digest, nil)
                                end
                               end
```

With the code in Listing 9.11, we're now ready to forget a permanent session by adding a `forget` helper and calling it from the `log_out` helper (Listing 9.12). As seen in Listing 9.12, the `forget` helper calls `user.forget` and then deletes the `user_id` and `remember_token` cookies.

Listing 9.12: Logging out from a persistent session. **green** `app/helpers/sessions_helper.rb`

```
                    module SessionsHelper

                  # Logs in the given user.
                      def log_in(user)
                  session[:user_id] = user.id
                                end
                                 .
                                 .
                                 .
              # Forgets a persistent session.
                     def forget(user)
                         user.forget
                   cookies.delete(:user_id)
                cookies.delete(:remember_token)
                                end

                # Logs out the current user.
                       def log_out
                    forget(current_user)
                  session.delete(:user_id)
                    @current_user = nil
                                end
                               end
```

At this point, the tests suite should be **green**:

Listing 9.13: **green**

```
$ rails test
```

Exercises

1. After logging out, verify that the corresponding cookies have been removed from your browser.

9.1.4 Two subtle bugs

There are two closely related subtleties left to address. The first subtlety is that, even though the "Log out" link appears only when logged-in, a user could potentially have multiple browser windows open to the site. If the user logged out in one window, thereby setting `current_user` to `nil`, clicking the "Log out" link in a second window would result in an error because of `forget(current_user)` in the `log_out` method (Listing 9.12).[13] We can avoid this by logging out only if the user is logged in.

The second subtlety is that a user could be logged in (and remembered) in multiple browsers, such as Chrome and Firefox, which causes a problem if the user logs out in the first browser but not the second, and then closes and re-opens the second one.[14] For example, suppose that the user logs out in Firefox, thereby setting the remember digest to `nil` (via `user.forget` in Listing 9.11). The application will still work in Firefox; because the `log_out` method in Listing 9.12 deletes the user's id, both highlighted conditionals are `false`:

```
# Returns the user corresponding to the remember token cookie.
                        def current_user
                    if (user_id = session[:user_id])
                @current_user ||= User.find_by(id: user_id)
               elsif (user_id = cookies.signed[:user_id])
                     user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
                            log_in user
                        @current_user = user
                               end
                             end
                           end
```

As a result, evaluation falls off the end of the `current_user` method, thereby returning `nil` as required.

In contrast, if we close Chrome, we set `session[:user_id]` to `nil` (because all `session` variables expire automatically on browser close), but the `user_id` *cookie* will still be present. This means that the corresponding user will still be pulled out of the database when Chrome is re-launched:

```
# Returns the user corresponding to the remember token cookie.
                        def current_user
                    if (user_id = session[:user_id])
                @current_user ||= User.find_by(id: user_id)
               elsif (user_id = cookies.signed[:user_id])
                     user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
                            log_in user
                        @current_user = user
                               end
                             end
                           end
```

Consequently, the inner `if` conditional will be evaluated:

```
user && user.authenticated?(cookies[:remember_token])
```

In particular, because `user` isn't `nil`, the *second* expression will be evaluated, which raises an error. This is because the user's remember digest was deleted as part of logging out (Listing 9.11) in Firefox, so when we access the application in Chrome we end up calling

```
BCrypt::Password.new(remember_digest).is_password?(remember_token)
```

with a `nil` remember digest, thereby raising an exception inside the bcrypt library. To fix this, we want `authenticated?` to return `false` instead.

These are exactly the sorts of subtleties that benefit from test-driven development, so we'll write tests to catch the two errors before correcting them. We first get the integration test from Listing 8.31 to **red**, as shown in Listing 9.14.

Listing 9.14: A test for user logout. **red** `test/integration/users_login_test.rb`

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest
                                .
                                .
                                .
 test "login with valid information followed by logout" do
                          get login_path
 post login_path, params: { session: { email:    @user.email,
                                       password: 'password' } }
                     assert is_logged_in?
                   assert_redirected_to @user
                        follow_redirect!
                  assert_template 'users/show'
      assert_select "a[href=?]", login_path, count: 0
          assert_select "a[href=?]", logout_path
        assert_select "a[href=?]", user_path(@user)
                     delete logout_path
                   assert_not is_logged_in?
                assert_redirected_to root_url
    # Simulate a user clicking logout in a second window.
                     delete logout_path
                      follow_redirect!
           assert_select "a[href=?]", login_path
  assert_select "a[href=?]", logout_path,      count: 0
  assert_select "a[href=?]", user_path(@user), count: 0
                             end
                            end
```

The second call to `delete logout_path` in Listing 9.14 should raise an error due to the missing `current_user`, leading to a **red** test suite:

Listing 9.15: **red**

```
$ rails test
```

The application code simply involves calling `log_out` only if `logged_in?` is true, as shown in Listing 9.16.

Listing 9.16: Only logging out if logged in. **green** `app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
                                .
                                .
                                .
                            def destroy
                          log_out if logged_in?
                          redirect_to root_url
                                end
                               end
```

The second case, involving a scenario with two different browsers, is harder to simulate with an integration test, but it's easy to check in the User model test directly. All we need is to start with a user that has no remember digest (which is true for the `@user` variable defined in the `setup` method) and then call `authenticated?`, as shown in Listing 9.17. (Note that we've just left the remember token blank; it doesn't matter what its value is, because the error occurs before it ever gets used.)

Listing 9.17: A test of `authenticated?` with a nonexistent digest. **red** `test/models/user_test.rb`

```
                          require 'test_helper'

                      class UserTest < ActiveSupport::TestCase

                                  def setup
        @user = User.new(name: "Example User", email: "user@example.com",
                         password: "foobar", password_confirmation: "foobar")
                                  end
                                .
                                .
                                .
  test "authenticated? should return false for a user with nil digest" do
                         assert_not @user.authenticated?('')
```

```
  end
end
```

Because `BCrypt::Password.new(nil)` raises an error, the test suite should now be **red**:

Listing 9.18: **red**

```
$ rails test
```

To fix the error and get to **green**, all we need to do is return `false` if the remember digest is `nil`, as shown in Listing 9.19.

Listing 9.19: Updating `authenticated?` to handle a nonexistent digest. **green** `app/models/user.rb`

```
class User < ApplicationRecord
                .
                .
                .
  # Returns true if the given token matches the digest.
          def authenticated?(remember_token)
          return false if remember_digest.nil?
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
          end

                .
                .
                .
          # Forgets a user.
             def forget
    update_attribute(:remember_digest, nil)
          end
end
```

This uses the `return` keyword to return immediately if the remember digest is `nil`, which is a common way to emphasize that the rest of the method gets ignored in that case. The equivalent code

```
if remember_digest.nil?
          false
        else
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
          end
```

would also work fine, but I prefer the explicitness of the version in Listing 9.19 (which also happens to be slightly shorter).

With the code in Listing 9.19, our full test suite should be **green**, and both subtleties should now be addressed:

Listing 9.20: **green**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Comment out the fix in Listing 9.16 and then verify that the first subtle bug is present by opening two logged-in tabs, logging out in one, and then clicking "Log out" link in the other.
2. Comment out the fix in Listing 9.19 and verify that the second subtle bug is present by logging out in one browser and closing and opening the second browser.
3. Uncomment the fixes and confirm that the test suite goes from **red** to **green**.

9.2 "Remember me" checkbox

With the code in Section 9.1.3, our application has a complete, professional-grade authentication system. As a final step, we'll see how to make staying logged in optional using a "remember me" checkbox. A mockup of the login form with such a checkbox appears in Figure 9.3.

Figure 9.3: A mockup of a “remember me” checkbox.

To write the implementation, we start by adding a checkbox to the login form from Listing 8.4. As with labels, text fields, password fields, and submit buttons, checkboxes can be created with a Rails helper method. In order to get the styling right, though, we have to *nest* the checkbox inside the label, as follows:

```
<%= f.label :remember_me, class: "checkbox inline" do %>
  <%= f.check_box :remember_me %>
  <span>Remember me on this computer</span>
<% end %>
```

Putting this into the login form gives the code shown in Listing 9.21.

Listing 9.21: Adding a “remember me” checkbox to the login form. `app/views/sessions/new.html.erb`

```
<% provide(:title, "Log in") %>
<h1>Log in</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(:session, url: login_path) do |f| %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.label :remember_me, class: "checkbox inline" do %>
        <%= f.check_box :remember_me %>
        <span>Remember me on this computer</span>
      <% end %>

      <%= f.submit "Log in", class: "btn btn-primary" %>
```

```
    <% end %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>
```

In Listing 9.21, we've included the CSS classes `checkbox` and `inline`, which Bootstrap uses to put the checkbox and the text ("Remember me on this computer") in the same line. In order to complete the styling, we need just a few more CSS rules, as shown in Listing 9.22. The resulting login form appears in Figure 9.4.

Listing 9.22: CSS for the "remember me" checkbox. `app/assets/stylesheets/custom.scss`

```
.
.
.
/* forms */
.
.
.
.checkbox {
  margin-top: -10px;
  margin-bottom: 10px;
  span {
    margin-left: 20px;
    font-weight: normal;
  }
}

#session_remember_me {
  width: auto;
  margin-left: 0;
}
```

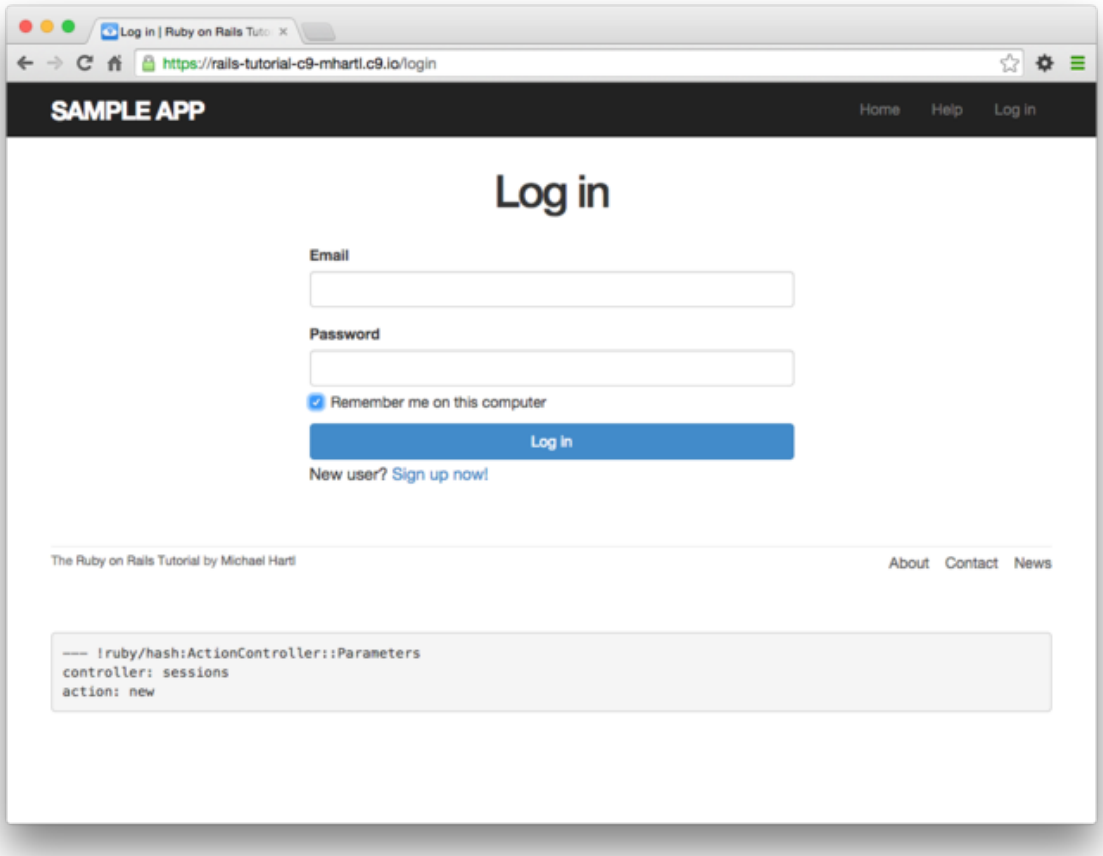

Figure 9.4: The login form with an added "remember me" checkbox.

Having edited the login form, we're now ready to remember users if they check the checkbox and forget them otherwise. Incredibly, because of all our work in the previous sections, the implementation can be reduced to one line. We start by noting that the `params` hash for submitted login forms now includes a value based on the checkbox

(as you can verify by submitting the form in Listing 9.21 with invalid information and inspecting the values in the debug section of the page). In particular, the value of

```
params[:session][:remember_me]
```

is `'1'` if the box is checked and `'0'` if it isn't.

By testing the relevant value of the `params` hash, we can now remember or forget the user based on the value of the submission:[15]

```
if params[:session][:remember_me] == '1'
                remember(user)
                    else
                 forget(user)
                    end
```

As explained in Box 9.2, this sort of `if-then` branching structure can be converted to one line using the *ternary operator* as follows:[16]

```
params[:session][:remember_me] == '1' ? remember(user) : forget(user)
```

Using this to replace `remember user` in the Sessions controller's `create` method (Listing 9.7) leads to the amazingly compact code shown in Listing 9.23. (Now you're in a position to understand the code in Listing 8.21, which uses the ternary operator to define the bcrypt `cost` variable.)

Listing 9.23: Handling the submission of the "remember me" checkbox.
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

                          def new
                            end

                         def create
    user = User.find_by(email: params[:session][:email].downcase)
      if user && user.authenticate(params[:session][:password])
                         log_in user
    params[:session][:remember_me] == '1' ? remember(user) : forget(user)
                        redirect_to user
                             else
       flash.now[:danger] = 'Invalid email/password combination'
                          render 'new'
                             end
                            end

                        def destroy
                      log_out if logged_in?
                      redirect_to root_url
                            end
                           end
```

With the implementation in Listing 9.23, our login system is complete, as you can verify by checking or unchecking the box in your browser.

Box 9.2. 10 types of people

There's an old joke that there are 10 kinds of people in the world: those who understand binary and those who don't (10, of course, being 2 in binary). In this spirit, we can say that there are 10 kinds of people in the world: those who like the ternary operator, those who don't, and those who don't yet know about it. (If you happen to be in the third category, soon you won't be any longer.)

When you do a lot of programming, you quickly learn that one of the most common bits of control flow goes something like this:

```
if boolean?
 do_one_thing
    else
do_something_else
    end
```

Ruby, like many other languages (including C/C++, Perl, PHP, and Java), allows you to replace this with a much more compact expression using the *ternary operator* (so called because it consists of three parts):

```
boolean? ? do_one_thing : do_something_else
```

You can also use the ternary operator to replace assignment, so that

```
if boolean?
  var = foo
   else
  var = bar
    end
```

becomes

```
var = boolean? ? foo : bar
```

Finally, it's often convenient to use the ternary operator in a function's return value:

```
def foo
 do_stuff
boolean? ? "bar" : "baz"
end
```

Since Ruby implicitly returns the value of the last expression in a function, here the `foo` method returns `"bar"` or `"baz"` depending on whether `boolean?` is `true` or `false`.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. By inspecting your browser's cookies directly, verify that the "remember me" checkbox is having its intended effect.
2. At the console, invent examples showing both possible behaviors of the ternary operator (Box 9.2).

9.3 Remember tests

Although our "remember me" functionality is now working, it's important to write some tests to verify its behavior. One reason is to catch implementation errors, as discussed in a moment. Even more important, though, is that the core user persistence code is in fact completely untested at present. Fixing these issues will require some trickery, but the result will be a far more powerful test suite.

9.3.1 Testing the "remember me" box

When I originally implemented the checkbox handling in Listing 9.23, instead of the correct

```
params[:session][:remember_me] == '1' ? remember(user) : forget(user)
```

I actually used

```
params[:session][:remember_me] ? remember(user) : forget(user)
```

In this context, `params[:session][:remember_me]` is either `'0'` or `'1'`, both of which are `true` in a boolean context, so the resulting expression is *always true*, and the application acts as if the checkbox is always checked. This is exactly the kind of error a test can catch.

Because remembering users requires that they be logged in, our first step is to define a helper to log users in inside tests. In Listing 8.23, we logged a user in using the `post` method and a valid `session` hash, but it's cumbersome to do this every time. To avoid needless repetition, we'll write a helper method called `log_in_as` to log in for us.

Our method for logging a user in depends on the type of test. Inside controller tests, we can manipulate the `session` method directly, assigning `user.id` to the `:user_id` key (as first seen in Listing 8.14):

```
def log_in_as(user)
session[:user_id] = user.id
end
```

We call the method `log_in_as` to avoid any confusion with the application code's `log_in` method as defined in Listing 8.14. Its location is in the `ActiveSupport::TestCase` class inside the `test_helper` file, the same location as the `is_logged_in?` helper from Listing 8.26:

```
class ActiveSupport::TestCase
  fixtures :all

  # Returns true if a test user is logged in.
  def is_logged_in?
    !session[:user_id].nil?
  end

  # Log in as a particular user.
  def log_in_as(user)
    session[:user_id] = user.id
  end
end
```

We won't actually need this version of the method in this chapter, but we'll put it to use in Chapter 10.

Inside integration tests, we can't manipulate `session` directly, but we can `post` to the sessions path as in Listing 8.23, which leads to the `log_in_as` method shown here:

```
class ActionDispatch::IntegrationTest

  # Log in as a particular user.
  def log_in_as(user, password: 'password', remember_me: '1')
    post login_path, params: { session: { email: user.email,
                                          password: password,
                                          remember_me: remember_me } }
  end
end
```

Because it's located inside the `ActionDispatch::IntegrationTest` class, this is the version of `log_in_as` that will be called inside integration tests. We use the same method name in both cases because it lets us do things like use code from a controller test in an integration without making any changes to the login method.

Putting these two methods together yields the parallel `log_in_as` helpers shown in Listing 9.24.

Listing 9.24: Adding a `log_in_as` helper. `test/test_helper.rb`

```
ENV['RAILS_ENV'] ||= 'test'
  .
  .
  .
class ActiveSupport::TestCase
  fixtures :all

  # Returns true if a test user is logged in.
  def is_logged_in?
    !session[:user_id].nil?
  end

  # Log in as a particular user.
  def log_in_as(user)
    session[:user_id] = user.id
  end
end

class ActionDispatch::IntegrationTest

  # Log in as a particular user.
  def log_in_as(user, password: 'password', remember_me: '1')
    post login_path, params: { session: { email: user.email,
                                          password: password,
                                          remember_me: remember_me } }
  end
end
```

Note that, for maximum flexibility, the second `log_in_as` method in Listing 9.24 accepts keyword arguments (as in Listing 7.13), with default values for the password and for the "remember me" checkbox set to `'password'` and `'1'`,

respectively.

To verify the behavior of the “remember me” checkbox, we’ll write two tests, one each for submitting with and without the checkbox checked. This is easy using the login helper defined in Listing 9.24, with the two cases appearing as

```
log_in_as(@user, remember_me: '1')
```

and

```
log_in_as(@user, remember_me: '0')
```

(Because `'1'` is the default value of `remember_me`, we could omit the corresponding option in the first case above, but I’ve included it to make the parallel structure more apparent.)

After logging in, we can check if the user has been remembered by looking for the `remember_token` key in the `cookies`. Ideally, we would check that the cookie’s value is equal to the user’s remember token, but as currently designed there’s no way for the test to get access to it: the `user` variable in the controller has a remember token attribute, but (because `remember_token` is virtual) the `@user` variable in the test doesn’t. Fixing this minor blemish is left as an exercise (Section 9.3.1.1), but for now we can just test to see if the relevant cookie is `nil` or not.

There’s one more subtlety, which is that for some reason inside tests the `cookies` method doesn’t work with symbols as keys, so that

```
cookies[:remember_token]
```

is always `nil`. Luckily, `cookies` *does* work with string keys, so that

```
cookies['remember_token']
```

has the value we need. The resulting tests appear in Listing 9.25. (Recall from Listing 8.23 that `users(:michael)` references the fixture user from Listing 8.22.)

Listing 9.25: A test of the “remember me” checkbox. **green** `test/integration/users_login_test.rb`

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "login with remembering" do
    log_in_as(@user, remember_me: '1')
    assert_not_empty cookies['remember_token']
  end

  test "login without remembering" do
    # Log in to set the cookie.
    log_in_as(@user, remember_me: '1')
    # Log in again and verify that the cookie is deleted.
    log_in_as(@user, remember_me: '0')
    assert_empty cookies['remember_token']
  end
end
```

Assuming you didn’t make the same implementation mistake I did, the tests should be **green**:

Listing 9.26: **green**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people’s answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. As mentioned above, the application currently doesn't have any way to access the virtual `remember_token` attribute in the integration test in Listing 9.25. It is possible, though, using a special test method called `assigns`. Inside a test, you can access *instance* variables defined in the controller by using `assigns` with the corresponding symbol. For example, if the `create` action defines an `@user` variable, we can access it in the test using `assigns(:user)`. Right now, the Sessions controller `create` action defines a normal (non-instance) variable called `user`, but if we change it to an instance variable we can test that `cookies` correctly contains the user's remember token. By filling in the missing elements in Listing 9.27 and Listing 9.28 (indicated with question marks `?` and `FILL_IN`), complete this improved test of the "remember me" checkbox.

Listing 9.27: A template for using an instance variable in the `create` action.
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

                          def new
                            end

                         def create
    ?user = User.find_by(email: params[:session][:email].downcase)
     if ?user && ?user.authenticate(params[:session][:password])
                         log_in ?user
params[:session][:remember_me] == '1' ? remember(?user) : forget(?user)
                       redirect_to ?user
                            else
       flash.now[:danger] = 'Invalid email/password combination'
                         render 'new'
                             end
                            end

                        def destroy
                      log_out if logged_in?
                      redirect_to root_url
                            end
                           end
```

Listing 9.28: A template for an improved "remember me" test. **green** `test/integration/users_login_test.rb`

```
                    require 'test_helper'

          class UsersLoginTest < ActionDispatch::IntegrationTest

                           def setup
                        @user = users(:michael)
                            end
                              .
                              .
                              .
                  test "login with remembering" do
                  log_in_as(@user, remember_me: '1')
            assert_equal FILL_IN, assigns(:user).FILL_IN
                            end

               test "login without remembering" do
                    # Log in to set the cookie.
                  log_in_as(@user, remember_me: '1')
      # Log in again and verify that the cookie is deleted.
                  log_in_as(@user, remember_me: '0')
                assert_empty cookies['remember_token']
                            end
                              .
                              .
                              .
                           end
```

9.3.2 Testing the remember branch

In Section 9.1.2, we verified by hand that the persistent session implemented in the preceding sections is working, but in fact the relevant branch in the `current_user` method is currently completely untested. My favorite way to handle this kind of situation is to raise an exception in the suspected untested block of code: if the code isn't

covered, the tests will still pass; if it is covered, the resulting error will identify the relevant test. The result in the present case appears in Listing 9.29.

Listing 9.29: Raising an exception in an untested branch. **green** `app/helpers/sessions_helper.rb`

```
module SessionsHelper
                                .
                                .
                                .
  # Returns the user corresponding to the remember token cookie.
                             def current_user
                       if (user_id = session[:user_id])
                    @current_user ||= User.find_by(id: user_id)
                   elsif (user_id = cookies.signed[:user_id])
raise       # The tests still pass, so this branch is currently untested.
                          user = User.find_by(id: user_id)
          if user && user.authenticated?(cookies[:remember_token])
                                log_in user
                             @current_user = user
                                    end
                                  end
                                end
                                .
                                .
                                .
                              end
```

At this point, the tests are **green**:

Listing 9.30: **green**

```
$ rails test
```

This is a problem, of course, because the code in Listing 9.29 is broken. Moreover, persistent sessions are cumbersome to check by hand, so if we ever want to refactor the `current_user` method (as we will in Chapter 11) it's important to test it.

Because both versions of the `log_in_as` helper method defined in Listing 9.24 automatically set `session[:user_id]` (either explicitly or by posting to the login path), testing the "remember" branch of the `current_user` method is difficult in an integration test. Luckily, we can bypass this restriction by testing the `current_user` method directly in a Sessions helper test, whose file we have to create:

```
$ touch test/helpers/sessions_helper_test.rb
```

The test sequence is simple:

1. Define a `user` variable using the fixtures.
2. Call the `remember` method to remember the given user.
3. Verify that `current_user` is equal to the given user.

Because the `remember` method doesn't set `session[:user_id]`, this procedure will test the desired "remember" branch. The result appears in Listing 9.31.

Listing 9.31: A test for persistent sessions. `test/helpers/sessions_helper_test.rb`

```
require 'test_helper'

class SessionsHelperTest < ActionView::TestCase

                        def setup
                      @user = users(:michael)
                          remember(@user)
                              end

  test "current_user returns right user when session is nil" do
                      assert_equal @user, current_user
                            assert is_logged_in?
                                    end

  test "current_user returns nil when remember digest is wrong" do
@user.update_attribute(:remember_digest, User.digest(User.new_token))
```

```
    assert_nil current_user
  end
end
```

Note that we've added a second test, which checks that the current user is `nil` if the user's remember digest doesn't correspond correctly to the remember token, thereby testing the `authenticated?` expression in the nested `if` statement:

```
if user && user.authenticated?(cookies[:remember_token])
```

Incidentally, in Listing 9.31 we could write

```
assert_equal current_user, @user
```

instead, and it would work just the same, but (as mentioned briefly in Section 5.3.4.1) the conventional order for the arguments to `assert_equal` is *expected*, *actual*:

```
assert_equal <expected>, <actual>
```

which in the case of Listing 9.31 gives

```
assert_equal @user, current_user
```

With the code as in Listing 9.31, the test is **red** as required:

Listing 9.32: **red**

```
$ rails test test/helpers/sessions_helper_test.rb
```

We can get the tests in Listing 9.31 to pass by removing the `raise` and restoring the original `current_user` method, as shown in Listing 9.33.

Listing 9.33: Removing the raised exception. **green** `app/helpers/sessions_helper.rb`

```
module SessionsHelper
                .
                .
                .
# Returns the user corresponding to the remember token cookie.
                def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.signed[:user_id])
      user = User.find_by(id: user_id)
    if user && user.authenticated?(cookies[:remember_token])
                log_in user
              @current_user = user
                end
              end
            end
              .
              .
              .
          end
```

At this point, the test suite should be **green**:

Listing 9.34: **green**

```
$ rails test
```

Now that the "remember" branch of `current_user` is tested, we can be confident of catching regressions without having to check by hand.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Verify by removing the `authenticated?` expression in Listing 9.33 that the second test in Listing 9.31 fails, thereby confirming that it tests the right thing.

9.4 Conclusion

We've covered a lot of ground in the last three chapters, transforming our promising but unformed application into a site capable of the full suite of signup and login behaviors. All that is needed to complete the authentication functionality is to restrict access to pages based on login status and user identity. We'll accomplish this task en route to giving users the ability to edit their information, which is the main goal of Chapter 10.

Before moving on, merge your changes back into the master branch:

```
$ rails test
$ git add -A
$ git commit -m "Implement advanced login"
$ git checkout master
$ git merge advanced-login
$ git push
```

Before deploying to Heroku, it's worth noting that the application will briefly be in an invalid state after pushing but before the migration is finished. On a production site with significant traffic, it's a good idea to turn *maintenance mode* on before making the changes:

```
$ heroku maintenance:on
$ git push heroku
$ heroku run rails db:migrate
$ heroku maintenance:off
```

This arranges to show a standard error page during the deployment and migration. (We won't bother with this step again, but it's good to see it at least once.) For more information, see the Heroku documentation on error pages.

9.4.1 What we learned in this chapter

- Rails can maintain state from one page to the next using persistent cookies via the `cookies` method.
- We associate to each user a remember token and a corresponding remember digest for use in persistent sessions.
- Using the `cookies` method, we create a persistent session by placing a permanent remember token cookie on the browser.
- Login status is determined by the presence of a current user based on the temporary session's user id or the permanent session's unique remember token.
- The application signs users out by deleting the session's user id and removing the permanent cookie from the browser.
- The ternary operator is a compact way to write simple if-then statements.

1. Session hijacking was widely publicized by the Firesheep application, which showed that remember tokens at many high-profile sites were visible when connected to public Wi-Fi networks. ↑
2. Rails 5 introduced a `has_secure_token` method that automatically generates random tokens, but it stores the *unhashed* values in the database, and hence is unsuitable for our present purposes. ↑
3. This choice is based on the RailsCast on remember me. ↑
4. In any case, with bcrypt's salted hashes there's no way for us to tell if two users' passwords match. ↑
5. With unique remember tokens, an attacker always needs *both* the user id and the remember token cookies to hijack the session. ↑
6. This hasn't stopped some developers from adding a check to verify that no collision has occurred, but such efforts result from failing to grasp just how small 10^{-40} 10−40 is. For example, if we generated a billion tokens a second for the entire age of the Universe, the expected number of collisions would still be on the order of 10^{-23} 10−23. ↑
7. As a general rule, if a method doesn't need an instance of an object, it should be a class method. Indeed, this decision will prove to be wise in Section 11.2. ↑
8. Signing and encrypting are different operations in general, but as of Rails 4 the `signed` method does both by default. ↑
9. As noted in Section 6.3.1, "unencrypted password" is a misnomer, as the secure password is *hashed*, not encrypted. ↑
10. I generally use the convention of putting such assignments in parentheses, which is a visual reminder that it's not a comparison. ↑
11. Alert reader Jack Fahnestock has noted that there is an edge case that isn't covered by the current design:
    1. Log in with "remember me" checked in browser A (saving hashed remember token A to `remember_digest`).

2. Log in with “remember me” checked in browser B (saving hashed `remember_token` B to `remember_digest`, overwriting remember token A saved in browser A).
3. Close browser A (now relying on permanent cookies for login—second conditional in `current_user` method).
4. Reopen browser A (`logged_in?` returns false, even though permanent cookies are on the browser).

Although this is a arguably a more secure design than remembering the user in multiple places, it violates the expectation that users can be permanently remembered on more than one browser. The solution, which is substantially more complicated than the present design, is to factor the remember digest into a separate table, where each row has a user id and a digest. Checking for the current user would then look through the table for a digest corresponding to a particular remember token. Furthermore, the `forget` in Listing 9.11 method would delete only the row corresponding to the digest of the current browser. For security purposes, logging out would remove all digests for that user.

↑

12. Google “<your browser name> inspect cookies” to learn how to inspect the cookies on your system. ↑
13. Thanks to reader Paulo Célio Júnior for pointing this out. ↑
14. Thanks to reader Niels de Ron for pointing this out. ↑
15. Note that this means unchecking the box will log out the user on all browsers on all computers. The alternate design of remembering user login sessions on each browser independently is potentially more convenient for users, but it’s less secure, and is also more complicated to implement. Ambitious readers are invited to try their hand at implementing it. ↑
16. Before we wrote `remember user` without parentheses, but when used with the ternary operator omitting them results in a syntax error. ↑