6.170 Tutorial 4 - Sessions and Authentication

Prerequisites
1. Having rails installed is recommended.

Goals of this Tutorial
1. Understand the basics of cookies and sessions and how to use them in rails.
2. Basic Authentication mechanisms in rails.

Resources
http://railscasts.com/episodes/250-authentication-from-scratch
http://railscasts.com/episodes/270-authentication-in-rails-3-1
http://railscasts.com/episodes/274-remember-me-reset-password

Topic 1: Cookies
A cookie is a piece of text that a website (web server) can store on a user's hard disk. Cookies allow a website to store information on a user's machine and later retrieve it. When the user browses the same website in the future, the data stored in the cookie can be retrieved by the website to notify the website of the user's previous activity. The data is stored as key-value pairs. For example, a website might generate a unique ID number for each visitor and store the ID number on each user's machine using a cookie file.

When your browser sends a request to a webserver it will look on your machine for a cookie file that the same web server has set. If it finds a cookie file, your browser will send all the key-value pairs along with the request.

The web server will receive the request and be able to access the cookie data. It will be able to use these cookies to gather information about the user.

Additionally, the web server can send extra information with the cookie such as expiration date or path (so the site can have different cookie values with different parts of the site).

In the broadest sense, a cookie allows a site to store state information on your machine. This information lets a web server remember what state your browser is in. An ID is one simple piece of state information -- if an ID exists on your machine, the site knows that you have visited before. The state is, "Your browser has visited the site at least one time," and the site knows your ID from that visit.

Things to keep in mind: People often share machines, cookies are easily (and often) erased, people have multiple machines.

Warning: Cookies are sent with every request to the server! As a web developer, you should avoid storing large amounts of data inside cookies. Cookies are limited to 4kb in size.

Setting Cookies

Typically, cookies are set in an HTTP response to the browser -- the server can specify in the header that a certain cookie should be set. (Note, cookies may also be set by Javascript: document.cookie = "key = value")

An HTTP response header may look like this:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value
Set-Cookie: name2=value2; Expires=Wed, 09-Jun-2021 10:18:14 GMT
```

The "Set-Cookie" directive tells the browser to create a cookie with that key and value, and to send that cookie back on future requests.

E.g. a future request *from the site* to the server might be:

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie: name=value; name2=value2
Accept: */*
```

Along with key-value pairs, the server can also set *cookie attributes*. These attributes tell the browser when to send the key-value pairs back. (Cookie attributes themselves are not sent back to the server.)

Cookie attributes

- Domain and Path
    - Tells the browser that the cookie should only be sent back for the given domain and path
- Expires and Max-Age
    - Tells the browser when to delete the cookie
        - Expires: provide a date
        - Max-Age: provide a number of seconds to persist
    - If neither are specified, the default is that it will be deleted by the browser after the user closes the browser.
- Secure and HttpOnly
    - These are binary attributes -- either present or not. They don't have an associated value.
    - Secure: tells browsers only to use the cookie under secure/encrypted connections

- HttpOnly: tells browsers to only use cookies via the HTTP protocol -- e.g. don't allow Javascript to modify cookies. (This attribute is used extensively by Facebook and Google to prevent some security vulnerabilities which we won't get into yet.)

For example:

```
Set-Cookie: HSID=AYQEVn….DKrdst; Domain=.foo.com; Path=/; Expires=Wed,
13-Jan-2021 22:23:01 GMT; HttpOnly
```

Topic 2: Sessions

HTTP is a stateless protocol -- it doesn't require the server to remember anything about a single user across multiple requests.

This works fine for serving static content, but what about dynamic/customized content? For example, a user on Amazon would want to see the same items in his "shopping cart" as he browses from page to page. We need some way to keep track of data from such a user session.

A common solution to this problem is using browser cookies. (Other solutions: server side sessions, hidden form variables, adding parameters to the URL.)

Topic 3: Sessions in Rails

Rails has built-in support for keeping track of user sessions.

There are a few storage mechanisms. All of these storage mechanisms use a cookie to store a unique ID for each session. They differ in where the rest of the data is kept.

- ActionDispatch::Session::CookieStore – Stores everything on the client.
- ActiveRecord::SessionStore – Stores the data in a database using Active Record.
- ActionDispatch::Session::CacheStore – Stores the data in the Rails cache.

If you'd like to change how you're storing sessions you can take a look and change it in the `config/initializers/session_store.rb`.

The default store is CookieStore, which stores all data in the browser cookie. Note that the data stored in the cookie isn't encrypted, so users can read it if they wanted. However, the cookie is signed so that users can't modify their cookie -- if they do, Rails will not accept it.

No matter which mechanism you choose, the session will be accessible via the sessions hash.

You might set a user_id in the session hash.

```
user = User.find_by_email(params[:email])
session[:user_id] = user.id
```

The user can then be retrieved on a subsequent call like so:
`User.find(session[:user_id]) if session[:user_id]`

Topic 4: Authentication (vs Authorization)

Authentication involves verifying that "this person is who they say they are." For example, you may show a photo ID to prove that you are this person with that name. On a website, you may enter in a username and password.

Don't confuse this with authorization! Authorization refers to answering the question of "what is this user allowed to access" - authentication asks "who is this user?"

Topic 5: HTTP Basic Auth

If you want something cheap and don't need much security, you can use http basic authentication. The following line at the top of a controller will create a popup that asks for the username and password before allowing the user to proceed to each page that the controller opens.

```
http_basic_authenticate_with :name => "dnj", :password => "password"
```

If you don't want this to apply to all methods within a controller, you can restrict it with extra parameters:

```
http_basic_authenticate_with :name => "username", :password =>
"password" :except => [:index, :show]
```

```
http_basic_authenticate_with :name => "username" :password =>
"password" :only => :destroy
```

Note that this option is insecure because your password is always sent in plaintext, and the password is stored in the code in plaintext. We might instead choose to use https as store a password hash instead of the password itself.

Topic 6: Secure Authentication

Rails has a built-in helper method for authentication, called `has_secure_password`. It encrypts user passwords for you before storing them. For the API docs on `has_secure_password`, you can refer to http://api.rubyonrails.org/classes/ActiveModel/SecurePassword/ClassMethods.html.

We'll do an example to see how to use it:
`rails generate model user email:string password_digest:string`
Note that we need a "password_digest" column. This field stores the encrypted passwords. `has_secure_password` assumes this field will exist.

has_secure_password

- must put “bcrypt-ruby” in Gemfile
- adds methods to set and authenticate the entered password
- adds validators to the password and password confirmation
- adds authentication functionality

Run the migration

```
rake db:migrate
```

app/models/user.rb

```
class User < ActiveRecord::Base
 attr_accessible :email, :password, :password_confirmation
 has_secure_password
 validates_presence_of :password, :on => :create
end
```

The attr_accessible statement prevents the password_digest from being set from the user registration form.

Create a users controller

```
rails generate controller users
```

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
   def new
     @user = User.new
   end

   def create
     @user = User.new(params[:user])
     if @user.save
       redirect_to root_url, :notice => “You are signed up.”
     else
       render “new”
   end
 end
end
```

Inside app/views/users/new.html.erb

```
<h1>Sign Up</h1>
<%= form_for @user do |form| %>
<div class=“field”>
```

```
 <%= f.label :email %>
 <%= f.text_field :email %>
</div>
<div class= "field">
 <%= f.label :password %>
 <%= f.text_field :password %>
</div>
<div class = "field">
 <%= f.label :password_confirmation %>
 <%= f.text_field :password_confirmation %>
</div>
<div class= "actions">
 <%= f.submit %>
</div>
<% end %>
```

We also need to let users log in, not just sign up. When we log a user in, though, we're not creating a new user -- we're creating a new session. We'll create a controller to handle sessions.

```
rails generate controller sessions
```

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
 def new
 end
 def create
   user = User.find_by_email(params[:email])
   if user && user.authenticate(params[:password])
     session[:user_id] = user.id
     redirect_to root_url, :notice => "Logged in"
   else
     flash.now.alert = "Invalid email or password"
     render "new"
   end
 end
 def destroy
   session[:user_id] = nil
   redirect_to root_url :notice=> "Logged out"
 end
end
```

The **authenticate** method is given to us by has_secure_password, and checks the given

password against the password in the database

app/views/sessions/new.html.erb

```
<h1>Log in</h1>
 <%= form_tag sessions_path do %>
   <div class= “field”>
     <%= label_tag :email %>
     <%= text_field_tag :email, params[:email] %>
   </div>
   <div class= “field”>
     <%= label_tag :password %>
     <%= password_field_tag :password %>
   </div>
   <div class = “actions”>
     <%= submit_tag “Log in” %>
   </div>
 <% end %>
```

Note: we use form_tag rather than form_for because we’re not editing a resource.

Now, we’re going to want to access the logged-in user from other parts of the site, after the user is logged in. We can make a helper method accessible to all views, that does this.

In app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
 protect_from_forgery

 private
   def current_user
     @current_user ||= User.find(session[:user_id]) if
session[:user_id]
   end
 helper_method :current_user
end
```

You can find a skeleton web app with basic user log in abilities at https://github.com/jtwarren/session_demo.

```
git clone https://github.com/jtwarren/session_demo.git
```

Next Tutorial: HTML & CSS

6.170 Software Studio
Spring 2013