I. Sessions

Why 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.)

A step back - What is a Browser Cookie?
It’s a small piece of data that websites can send to be stored in the browser. The data are stored in key-value pairs.

A browser is expected to be able to store at least 300 cookies of size 4 kB each, and at least 20 cookies per domain.

How Cookies are Set
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”)

E.g. 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
  o If neither are specified, the default is that it will be deleted by the browser after the user closes the browser.

Secure and HttpOnly
  o These are binary attributes -- either present or not. They don’t have an associated value.
  o Secure: tells browsers only to use the cookie under secure/encrypted connections
  o 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.)

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

Note: The most common browsers allow users to completely block cookies, and to delete all their existing cookies. You’ve probably done this yourself at some point. Just something to keep in mind.

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

There are a few storage mechanisms, and here are the three you might consider:
  ● 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.

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.

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. CookieStore stores up to 4kB of data.

Configuring your Rails Session Store
Settings are in config/initializers/session_store.rb
CookieStore is the default, so the contents looks like:

AppName::Application.config.session_store = :cookie_store,
key: '_appname_session'

If you wanted to use another type of storage, you can edit the :cookie_store to something else. Active record store is a better choice for sensitive information, since the user won’t be able to view it (it’s in the DB). Here’s how you set up active record store:
  ● Create the sessions database table
    o rake db:sessions:create
  ● rake db:migrate
in config/initializers/session_store.rb, change :cookie_store to :active_record_store.

You can specify other cookie attributes in config/initializers/session_store.rb e.g.

- :domain
  - YourApp::Application.config.session_store :cookie_store, :key => '_your_app_session', :domain => '.example.com'

Using the Rails Session Store
The session data is only available in the controller and view.

You can access session data via the session instance method. You can use it like a hash table.

From the Rails Guide --
Example of reading session data:
```ruby
class ApplicationController < ActionController::Base
  private

  # Finds the User with the ID stored in the session with the key # :current_user_id This is a common way to handle user login in # a Rails application; logging in sets the session value and # logging out removes it.
  def current_user
    @_current_user ||= session[:current_user_id] && User.find_by_id(session[:current_user_id])
  end
end
```

Example of setting session data:
```ruby
class LoginsController < ApplicationController
  # "Create" a login, aka "log the user in"
  def create
    if user = User.authenticate(params[:username], params[:password])
      # Save the user ID in the session so it can be used in # subsequent requests
      session[:current_user_id] = user.id
      redirect_to root_url
    end
  end
end
```

To remove data from a session, assign the key to nil.
```ruby
class LoginsController < ApplicationController
  # "Delete" a login, aka "log the user out"
  def destroy
    # Remove the user id from the session
    @_current_user = session[:current_user_id] = nil
    redirect_to root_url
  end
end
```
Or, to reset a session, issue the command `reset_session`.

**The Flash**

The flash is a special part of the session that is only available in the next request, and is then cleared. It's useful for storing error messages, which tells you about an error in the previous request.

For now, just know that the flash can be used like a hash, like the session: e.g. (from Rails Guide)

```ruby
class LoginsController < ApplicationController
  def destroy
    session[:current_user_id] = nil
    flash[:notice] = "You have successfully logged out"
    redirect_to root_url
  end
end
```

Then you might use it in your view like this:

```html
<html>
  <!-- <head/> -->
  <body>
    <% if flash[:notice] %>
      <p class="notice"><%= flash[:notice] %></p>
    <% end %>
    <% if flash[:error] %>
      <p class="error"><%= flash[:error] %></p>
    <% end %>
    <!-- more content -->
  </body>
</html>
```

**II. Authentication**

**What is Authentication?**

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.

**Authentication in Rails**

1. `http_basic_authenticate_with`

If you want something cheap and don’t need much security, you can use `http_basic_authenticate_with`
Just put

    http_basic_authenticate_with :name => "username", :password => "password"

at the top of a controller, and it will create a pop-up that asks for the username and password
before allowing the user to proceed to each page that the controller opens.

If you don’t want this to apply to all methods within a controller, you can restrict it with extra
parameters, e.g.

- http_basic_authenticate_with :name => "username", :password => "password" :except => [:index, :show]
  - allow unauthenticated users to see the content
- http_basic_authenticate_with :name => "username", :password => "password" :only => :destroy
  - only allow authenticated users to delete content

Example in context (from Rails Guide):

```ruby
class PostsController < ApplicationController

    http_basic_authenticate_with :name => "dhh", :password => "secret", :except => [:index, :show]

    # GET /posts
    # GET /posts.json
    def index
        @posts = Post.all
        respond_to do |format|
            ...
        end
    end

    # GET /posts
    def show
        @post = Post.find(params[:id])
        respond_to do |format|
            ...
        end
      end

    # PATCH/POST /posts
    def update
        @post = Post.find(params[:id])
        if @post.update_attributes(params[:post])
            respond_to do |format|
                format.html { redirect_to @post, notice: 'Post was successfully updated.' }
                format.json { render action: "show" }
            end
        else
            render :edit
        end
    end

    # DELETE /posts
    def destroy
        Post.find(params[:id]).destroy
        respond_to do |format|
            format.html { redirect_to posts_url, notice: 'Post was successfully destroyed.' }
            format.json { head "no_content" }
        end
    end

```

Note that this option is insecure because your password is always sent in plaintext, and the
password is stored in the code in plaintext.

2. has_secure_password

Rails has a built-in helper method for authentication, called has_secure_password. It encrypts user passwords for you before storing them.

We’ll do an example to see how to use it:

Create a user model

- rails generate model user email:string password_digest:string
  - Note that we need a “password_digest” column. This field stores the encrypted passwords.

Run the migration

- rake db:migrate

Inside app/models/user.rb
- class User < ActiveRecord::Base
  attr_accessible :email, :password, :password_confirmation
  has_secure_password
  validates_presence_of :password, :on => :create
end

- 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

- We also validate that each user has a password
- The attr_accessible statement prevents the password_digest from being set from the user registration form.

Inside 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
A Sign Up form
- <h1>Sign Up</h1>
  <%= form_for @user do |form| %>
  <% if @user.errors.any? %>
    <div class=“errors”>
      <h2>Form is invalid</h2>
      ....#print errors here
    </div>
  <% end %>
  <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 %>
  </div>
  ..
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

In 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

- the authenticate method is given to us by has_secure_password, and checks the given password against the password in the database

Then create the view: app/views/sessions/new.html.erb

A Log In form

- <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>
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

```ruby
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
```

HTTPS
At this point, the password is still being submitted in plain text. To send it encrypted, we have to use HTTPS. Check out the Rails “force_ssl” method.