Cookies

Goals

- Know what statelessness means.
- Know how a cookie can store data on the client.

HTTP is stateless

HTTP is stateless; each request that a client makes is independent of every other request. A stateless protocol is a communications protocol that treats each request as an independent transaction that is unrelated to any previous request so that the communication consists of independent pairs of requests and responses.

Okay, but what the hell does that mean? Let's take an example: say we login to amazon, then fetch our personal wishlist. HTTP itself doesn't build in a way to tie the first request (the login) to the second request (fetch the wishlist). That means that, without additional work, our web app won't understand whose wishlist to display.

This is the problem that cookies were invented to solve.

Stateless protocols allow server software to be written more simply and with fewer bugs. They can also make certain intelligent optimizations that let the web service operator scale up his or her machines more easily. These points aren't essential for us to dive into right now, but you should know that the simplicity of HTTP and other stateless protocols is part of what makes them so powerful.

Cookies

Okay, statelessness is a good thing, but how do we solve our problem?

Of course, it starts when we post login credentials to the server. We may exchange a request-response with `www.amazon.com` like this:

```
Request:
    POST /session HTTP/1.1
    Host: amazon.com

    user=ruggeri&password=gizmo_is_great

Response:
    HTTP/1.1 302
    Set-Cookie: user=ruggeri; session_token=135792468
    Location: http://www.amazon.com/
```

Here we post our credentials to the Amazon server. Amazon sends us the response: the 302 tells the browser to make a new request for the amazon home page. But the response also has a `Set-Cookie` header line.

The `Set-Cookie` line is telling the client (our web browser) to save some state in the browser: namely, the logged in Amazon user and a randomly generated session token (we'll see why in a minute). The browser is expected to upload this cookie as part of each subsequent request to `www.amazon.com`.

```
GET /wishlist http/1.1
Host: www.amazon.com
Cookie: user=ruggeri; session_token=135792468
```

The way the server can know this new request is being made by user `ruggeri` is through the cookie. It can verify the user is who they claim to be and that the cookie is not faked. When the user submitted the credentials, the session token set in the cookie would also be stored in the server's database. On future requests, we compare the session token for the specified user with the session token uploaded in the cookie.

Note that the server doesn't have any other way to tie together the login request and the wishlist request. Without the cookie, the server would have no way to differentiate this request from user `ruggeri` from another request from user `patel`.

Once a cookie is set, it continues to be uploaded to the server on every request until it expires or is cleared out by the user. It is common to set the cookie to expire on a date far in the future so that it continues to be uploaded. This is called a permanent cookie. A permanent cookie is not truly permanent; all things must pass. However, it does live for a very long time, and since a cookie can be renewed by being set again, it can live practically forever.

Another common option is not to set an expiration date at all. In this case, the cookie will be deleted when the web browser closes. This is called a session cookie.

Another example: shopping carts

Let's consider a second example. Say a logged in user (`ruggeri`) is navigating `www.amazon.com` and wants to add items to his shopping cart. Each shopping cart request might trigger a POST request to the server; the server would then update its database to reflect that user `ruggeri` has added another item to his shopping cart. When user `ruggeri` checks out, the server queries the database to check the items in his cart, and computes the price.

What about a marketplace that wants to allow users to add items to their carts without having logged in yet? Maybe it is only at checkout that the user logs in. Or maybe they never log in, and just provide credit-card information at checkout.

Here the users are anonymous: the server doesn't have a username for this user. For that reason, it wouldn't be easy to record in the database that user X has added an item to their cart, since there is no username for X.

One solution would be to store the shopping cart in the user's cookie. Each time the user makes a request, they'll upload the cookie containing the shopping cart. Whenever they add an item, the server will send the shopping cart cookie back, having first appended one more item to it.

Advantages and disadvantages

There are a few additional advantages to each approach. The approach of storing state on the server means that the user can use different devices (maybe a computer and a phone) to use the service, and still access the cart across machines. Also, if the cookies get cleared, they don't lose the items in their cart.

The information stored in a cart is very small: probably just the id numbers of the items. It won't hurt to upload them on every request. If there were more data to store (for instance, images), then it would slow down the interaction with the website to upload the cookie every time. Cookies have reasonable limitations to how much data they can hold; usually 4kB. But if we wanted to store more than this amount of data, we need to do this server-side.

There are advantages to storing the cart in the cookie. It may be simpler to just have the client store the cart in cookies. We don't have to store the cart in the database, clean up and remove old carts that users don't ever check out, etc. Depending on how interactive the site is, we may not have to set up a database at all.

Another consideration is that the web application may not be able to search data stored in the cookie. Say we wanted to run a query to ask what items were stored in all users' carts currently. With the cookie-based architecture, we haven't stored that information on our servers; it exists in the cookies saved on our users' computers, which we can't get access to. We only get access to that data when the user's browser makes a web request.

One last point: any information that must be reliably persisted must not be stored in a cookie. People clear cookies all the time, so you can't count on the data you save in the cookie being around later. And users can edit or fake the cookies, so you can't necessarily trust that the data won't be modified by a malicious user.

Sessions

Information like the logged in username or the items in a cart is called session data. Session data is part of the context in which future requests are interpreted. For instance, when a client makes a GET request for a wishlist, the context includes the currently logged in user. The session data, in aggregate, is called the session.

Session is a concept; cookies are the way that concept is implemented. When we start working with Rails, we'll see the word session more frequently than cookie; that's because Rails is trying to be flexible about whether it uses cookies or some other means to implement a session (e.g., by using the database, as described above).

Controllers and State

Session

HTTP is stateless. That means requests are handled independently from one another; your Rails controller dies at the end of every request (i.e. each request is handled by a new instance of the controller) and carries no data forward to the next request. But oftentimes we need to track information across requests.

We can hang onto data across requests by having the Rails application store it in the database. For instance, when a user signs up for our service, we `create` a new `User` object and store their credentials (username/password) in the `users` table.

Sometimes the client needs to store data across requests. For instance, think about login functionality. Let's break this down for the example of Facebook:

0. Facebook presents a login page when you go to facebook.com and are not logged in.
1. User fills out the web form, presses "submit". The web browser makes an HTTP request to POST credentials to the Facebook's server (perhaps to `/session`).
2. Facebook server verifies username/password. Sends a redirect to the browser instructing it to GET `/feed`.
3. Client makes GET request for `/feed`.

The problem is in the last step. From the point of view of Facebook's server, the GET for `/feed` is entirely unrelated to the POST to `/session`. Not being able to connect these two requests means that the GET for `/feed` looks like it's coming from an un-logged-in user.

How would Rails connect these two requests? They hit different controllers; even if they hit the same controller class, the controller is thrown away at the end of the request anyway; you'll be working with different instances. Either way, Rails won't remember the previous request.

To fix this problem, we need to have the client web browser store some data for us. Check it out:

0. Facebook presents a login page when you go to facebook.com and are not logged in.
1. User fills out the web form, presses "submit". The web browser makes an HTTP request to POST credentials to the Facebook's server (perhaps to `/session`).
2. Facebook server verifies username/password. Sends a redirect to the browser instructing it to GET `/feed`.
   - Also, Facebook generates a temporary *login token* for the user.
   - In an HTTP header, Facebook sets a cookie with the token when it issues redirect response.
3. Client makes GET request for `/feed`.
   - Client uploads the login token cookie.

  - Facebook looks up the user associated with the login token and shows their feed.

You know all this from the cookies chapter. Let's talk about how Rails lets you set cookies.

Rails does much of the work of implementing the session for us. Within our controller, we can use the `ActionController::Base#session` method to get a hash-like object where we can retrieve and set state (e.g. `session[:user_id] = @user.id`). When we call `render` or `redirect`, Rails will take the contents of the `session` hash and store it in the cookie.

Here's a barebones, simple demo:

```
# config/routes.rb
SecretApp::Application.routes.draw do
  resource :feed
  resource :session
end

# app/controllers/sessions_controller.rb
class SessionsController < ActionController::Base
  def create
    username = params[:user_name]
    password = params[:password]

    user = User.where(
      username: username,
      password: password
    ).first

    # generate a 16-digit random token
    user.session_token = SecureRandom::urlsafe_base64(16)
    user.save!

    # put the generated token in the client's cookies
    session[:session_token] = user.session_token
    redirect_to feed_url
  end
end

# app/controllers/feeds_controller.rb
class FeedsController < ActionController::Base
  def show
    # pull the session token out of the client's cookies
    # it will be right where we left it in session[:session_token]

    session_token = session[:session_token]
    @user = User.find_by(session_token: session_token)

    # render feed for user...
  end
end
```

Please note that the above is not very well-written Rails code; it's just trying to be simple. In particular, you should probably write helper methods in `ApplicationController` for `login_user!(username, password)` and `current_user`.

Anything you set in `session` will be stored in the cookie. On subsequent requests, Rails will read the cookie and deserialize the session. So `session` will contain the values you had set in previous requests.

Note that like `params`, `session` is actually a call to `ActionController::Base#session`, which returns a hash-like object.

To remove something from the `session`, set it to `nil`:

```
class SessionsController < ActionController::Base
  def destroy
    # logout
    session_token = session[:session_token]
```

```ruby
    user = User.find_by(session_token: session_token)

    user.session_token = SecureRandom::urlsafe_base64(16)
    user.save!
    session[:session_token] = nil

    redirect_to root_url
  end
end
```

The flash

When you store data in the session, it will keep coming back on subsequent requests. Oftentimes you'll find that you want to store some data only for the current request or until the next request.

The flash is a special part of the session that stores values for only the current and next request, which is particularly useful for passing error messages or other success/failure messages.

It is accessed in much the same way as the session, as a hash (it's a FlashHash instance).

Let's use the act of confirming an object was destroyed. The controller can set a message which will be displayed to the user on the next request:

```ruby
class SessionsController < ActionController::Base
  def destroy
    # logout
    session_token = session[:session_token]
    user = User.find_by(session_token: session_token)

    user.session_token = SecureRandom::urlsafe_base64(16)
    user.save!
    session[:session_token] = nil

    flash[:notices] ||= []
    flash[:notices] << "You logged out. See you soon!"

    redirect_to root_url
  end
end
```

This flash notice is available for the current and next controller and view to use. Let's modify `views/layouts/application.html.erb` (the template that every template is rendered in) to display notices.

```
<!-- in views/layouts/application.html.erb -->
<!-- `Object#try` is a Rails addition which will try to call the
      method if the object isn't `nil`.  -->

<!-- ... -->
<% flash[:notices].try(:each) do |msg| %>
  <%= msg %>
<% end %>

<!-- ... -->
```

This way, if an action sets a notice message, the layout will display it automatically.

Storing messages in flash is particularly useful for redirecting. When we `redirect_to` in rails, it tells the client to make an entirely separate request from the original request. Therefore if we want values to persist to that second request, we use flash. However, there are times when we want some values to be stored only for the lifecycle of the current request...

`flash.now`

By default, adding values to the flash will make them available for the current and next request, but sometimes you may want to store values only for the current request. In particular, if the `create` action fails to save a resource and you render (NOT redirect to) the `new` template directly, that's not going to result in a new request, but you may still want to display a message using the flash. To do this, you can use `flash.now` in the same way you use the normal `flash`:

```
class SessionsController < ActionController::Base
  def create
    username = params[:user_name]
    password = params[:password]

    user = User.where(
      username: username,
      password: password
    ).first

    if user.nil?
      # uh-oh, bad password
      flash.now[:notices] ||= []
      flash.now[:notices] << "Username/password combo was bad"

      render :new
      return
    end

    # ...
  end
end
```

This is useful because we are `render`-ing the `new.html.erb` view as the response to the *current* request, so we only need to store the error message for the current request. We wouldn't want the error message to persist through to a second request; otherwise, we might re-submit the form with correct credentials and upon that second request still see the error message from our original bad login attempt.

In summary:

- `redirect_to` --> makes second, separate request --> `flash` --> values stored for current AND next request
- `render` --> responds to current request --> `flash.now` --> values stored for current request only

`cookies` vs. `session`

If we're asking Rails to store data in an HTTP cookie, why is it called `session`? The reason is that "session" is a more abstract concept: the session is the accumulated context for an HTTP request. You can store (and remove) data from the session, which will be available in future requests.

There are different ways to implement a session:

- Store the data in an HTTP cookie: this is the default.
- Store all session data in the server's DB. Instead of setting the cookie with the data, only send the database key (the id of the session data in the DB) to the client. On subsequent requests the client ships the key. Rails looks up this key in the DB when session data is requested.

Rails lets you choose between these (I won't tell you how): the default is to store all session data in the cookie. Cookies are limited in size: 4kB. If you wanted to store lots more data as part of the session, you may want to store that server side and just store a small key with the user. You might also do that if you wanted to store private data in the cookie and didn't want to send it to the client. But it is atypical to store private data in the session, anyway...

You can also get/set data to an HTTP cookie directly using `ActionController::Base#cookies`. You won't want to do this often (don't fixate too much on it right now), but it does give you a little more power:

```
session_token = SecureRandom::urlsafe_base64(16)
cookies[:session_token] = {
  value: session_token,
  expires: 20.year.from_now
}
```

Cookies will typically expire at the end of a browser session (argh! session used again another way!). When the user closes the browser, cookies are by default cleared. But by using the low-level `cookies` method we can set the cookie to live for up to 20 years (this is the max allowed by the HTTP cookie specifications).

You may keep in mind that the commercial internet is less than 20 years old. Cookies were introduced in 1997. Twenty years is truly forever in internet terms.

You can abbreviate this by writing `cookies.permanent[:session_token] = session_token`; I like the explicit way better. This is the primary reason (maybe 90% of the cases) for using `cookies` in preference to `session`. Use `session` for everything else.

One limitation: A cookie's value can only be a string. You're responsible for any serialization (usually to JSON) or deserialization that needs to happen. With `session`, Rails will do that extra work for you so that you can store other data types like arrays and hashes.

Resources

- Session Rails Guide
- Flash Rails Guide
- FlashHash API

Cross-site Request Forgery

You've been learning how to write forms. Here's a tricky form:

```
<!-- this is a form on my www.appacademy.io -->
<form action="https://www.facebook.com/pages/appacademy/like"
      method="post">
  <input type="submit" value="Click to win a puppy!">
</form>
```

This form on appacademy.io advertises a chance to win a free puppy. However, when the user clicks the button, it instead issues a request to facebook.com to like App Academy.

Now, Facebook only processes likes for logged in users: that's how it enforces that a user can only like a page at most once. If Facebook didn't require me to be logged in, I wouldn't bother tricking users to like App Academy; I'd just click the button myself a lot.

Because I can only like a page once, I want to try to trick other Facebook users to like my page for me. For that reason, I publish a deceptive form on my site, which then POSTs a like request to Facebook's site. This is called a cross-site (originates on my site but attacks Facebook) request forgery (issues an unintended request to like my page).

Note that the forgery only succeeds if the target is signed into Facebook. If they are not, a request to like my page will not succeed: Facebook will return an error saying the like request is not authorized for an anonymous user. However, since many, many browsers are likely to be logged into Facebook, this attack will work fairly often.

CSRF Authenticity Token

Rails, by default, tries to protect your forms from being attacked like this. Here's how.

On each request, Rails will set an *authenticity token* in your session. The authenticity token is just a random number; it has no special meaning. Like everything in the session, it will be uploaded with each subsequent request.

When you make any non-GET request to Rails (POST, PUT, DELETE), Rails will expect the client to also upload the authenticity token in the params. If we were Facebook engineers writing Rails (oh no! they use PHP!), we could do this:

```
<!-- this is a form on www.facebook.com -->
<form action="pages/appacademy/like" method="post">
  <input type="hidden"
         name="authenticity_token"
```

```
          value="<%= form_authenticity_token %>">

  <input type="submit" value="Like App Academy!">
</form>
```

The `form_authenticity_token` helper just looks at the user session, takes the authenticity token stored in there, and puts it in the form. When the form is uploaded, Rails will check that the uploaded token in the params equals the token stored in the session. If they are equal, Rails knows that the user submitting the form was the one who requested it in the first place.

How The Authenticity Token Check Helps

On any non-GET request, if the authenticity token in the uploaded form data does not match the token in the session, Rails will raise the **`ActionController::InvalidAuthenticityToken`** exception. This is because in the `application_controller.rb` the default strategey is set to :exception. Optionally, you can change the strategy to :null_session, in which case Rails will blank out the session instead of raising an exception. That effectively logs the user out. Both of these strategies are ways that Rails can protect you from CSRF.

In previous versions of Rails, the default strategy was to blank out the session. That strategy would work for our previous example because Facebook will then presumably reject the submission because it looks like it has come from an anonymous user.

While it was very effective at protecting your application, this strategy also made it difficult to debug any problems with the authenticity token, since there was no explicit exception raised. The app would simply keep running as if the user had logged out, and it was not always obvious that a problem had occurred with the CSRF protection.

You Need To Use The Authenticity Token

For every form you write, you should make sure to upload the authenticity token. Otherwise, if the action you are POSTing (or PUTing, etc.) to requires session data, it will not have access to it. Some actions may not really require the session anyway, but keep things simple: just include the little boilerplate above as part of every form and you will be fine.

Validation

For sundry reasons, data saved to the database need to be validated. Model level validations are the most common, but validations can be created on the client side in JavaScript, or controller level validations. Database constraints can also be added to the database to prevent bad input.

Performing the validations at the model level has these benefits:

- Database agnostic
- Convenient to test and maintain
- Nice, specific, per-field error messages
- Avoids reliance on database constraints (which throw nasty errors)

ActiveRecord::Base instance methods that run validations:

- `#create`
- `#create!`
- `#save`
- `#save!`
- `#update`
- `#update_attributes`
- `#update_attributes!`

Model-Level Validation

Lets consider the `User` class with basic validations for username and password.

```ruby
class User < ActiveRecord::Base
  validates :username, :password, presence: true
  validates :password, length: { minimum: 6 }
end
```

So if we run

```ruby
u = User.new(username: "Billy Bob")
u.save
```

`u.save` will run validations. After completing the validations the `u.errors` attribute will contain a hash with all of the errors, by attribute.

```
=> { password: ["can't be blank", "is too short (minimum is 6 characters)"]}
```

`flash` vs `flash.now`

Data stored in `flash` will be available to the next controller action and can be used when redirecting.

Data stored in `flash.now` will only be available in the view currently being rendered by the `render` method.

Flash Hash

Rendering Errors

One common way to render errors is to store them in `flash[:errors]` after running validation, then displaying them on the page by iterating through each of the errors displaying each of them. Note that `flash` can be used as a hash and `:errors` is just an arbitrary, though semantically meaningful, key we have chosen.

By following the construct of always storing `obj.errors.full_messages` in `flash[:errors]` a partial can be rendered with the flash errors on each of the forms.

Controller

Lets make a simple controller that will store any validation errors in a `flash[:errors]` array.

```
def create
  @user = User.new(params[:user])
  if @user.save
    flash[:notice] = "Success!"
    redirect_to user_url(@user)
  else
    # sweet! now my flash.now[:errors] will be full of informative errors!
    flash.now[:errors] = @user.errors.full_messages
    render :new
  end
end
```

View

When the new or edit pages are rendered, also render a list of all of the errors if any.

```
<!-- layouts/_errors.html.erb -->
<% if flash.now[:errors] %>
  <% flash.now[:errors].each do |error| %>
   <%= error %><br />
  <% end %>
<% end %>

<!-- views/users/new.html.erb -->
<%= render partial: "layouts/errors" %>

<!-- views/users/edit.html.erb -->
<%= render partial: "layouts/errors" %>
```

Notice how trivial it is to add the list of errors to future new/edit pages by using a partial.

Resources

* Rails Guides - Validations
* 4h of error messages