thoughtbot

How HTTP Cookies Work

Edward Loveall – December 21, 2018 UPDATED ON March 23, 2019

LUCKY, WEB, HTTP

Lucky, a type-safe web framework written in Crystal by thoughtbot, recently released version 0.12. Along with many other changes, it includes a new cookie system. When starting this work, I knew very little about how HTTP cookies actually worked. We'll explore what I learned about cookies and how they are implemented.

Back and Forth

Let's talk about how cookies are transferred between the browser and the server. Cookies use two headers: `Set-Cookie` and `Cookie`. When a server responds to a browser request, it can send down a `Set-Cookie` header with one or many cookies:

```
Set-Cookie: user_id=5; Expires=Fri, 5 Oct 2018 14:28:00 GMT; Secure; H
```

Each cookie is separated by a comma `,` and each cookie attributes are separated by semicolons `;`. The two values required are the first `name=value` pair which are always string values. The remaining attributes that set other parameters of a cookie are optional and set other parameters of a cookie.

To send a cookie back to the server, the browser uses the `Cookie` header:

Each cookie is separated by a semicolon `;`. Don't confuse this with the `Set-Cookie` header which confusingly uses a `,` to separate multiple cookies. Notice that each cookie only contains the `name=value` pair. The browser cannot send other attributes of a cookie back to the server.

Although many programming languages and frameworks will abstract the parsing and creation of these cookie headers for you (Crystal, Ruby, PHP, Phoenix, Node.js, Python), it is occasionally good to know how it all works behind the scenes.

Expiration and Removal

Now that we know how cookies are set let's look at how they are un-set.

Cookies can expire. A cookie with no expiration date specified will expire when the browser is closed. These are often called session cookies because they are removed after the browser session ends (when the browser is closed). Cookies with an expiration date in the past will be removed from the browser.

To remove a cookie, you must set it's set its expiration date in the past. This will signal to the browser that the cookie should be removed. For cleanliness, it's also a good idea to set its value to an empty string.

```
Set-Cookie: user_id=; Expires=Fri, 5 Oct 2018 14:28:00 GMT;
```

Other Goodies

Cookies have a few other interesting attributes that are used to restrict or permit them from certain locations:

- `Secure` : This will ensure that cookies can only be sent to HTTPS servers.

include the `Path`.

I won't cover all these in detail. You can read more about them on the Mozilla's web resource.

Also not covered here will be the maximum amount of data you can store on a cookie. For the most part, you can assume that you're fine if you're under 4k worth of data. In practice, it's more complicated.

Sessions

Sessions are less straightforward. A session represents the currently logged in user. This might be done by storing something as simple as the `user_id`, but there is no standard. As far as HTTP is concerned there is no such thing as a session. We must come up with our own way to store this information.

One common method is to use a cookie's value to store the session:

```
Set-Cookie: _myapp_session={"user_id": "5"};
```

An app can now look at the cookie with the name of `_myapp_session`, read and parse the JSON, and use it for things like setting the `current_user` on the request. However, the method above as-is is extremely easy to hack. It's just plain text!

Lock It Down

A good session is encrypted. A more real-world example would be:

```
Set-Cookie: _myapp_session=zjMvwPnfH7BSRrVIppsUI41eCimOtMOcMwjhAupZntB
```

string of JSON is encrypted using the AES 256 standard which turns it into garbled bytes. It is then base64 encoded so it is an ASCII string, since the underlying HTTP protocols expect to work with ASCII. That base64 encoded string becomes the value of the cookie.

When cookies are sent back to the server, they are read, (base64) decoded, decrypted, JSON parsed, and stored in memory as key/value pairs.

This is how sessions work in Lucky. The session will be a cookie with a name like `_myapp_session`. The value of the cookie is an encrypted JSON string that can only be decrypted by a server with the session key. Storing it as JSON allows us to have a key/value like store but using a single cookie instead of multiple.

There are other ways to store session data, such as a key-value store like Redis. But even this requires a cookie to identify which values to retrieve.

A Side Note About The Flash

The flash is a one-off message displayed to the user after they do something. For example, after you sign in you might see the message:

> *Welcome back Edward!*

If you refresh the page or navigate elsewhere, this message disappears.

Flash messages are stored as two separates parts: messages for the current request and messages to be displayed on the next request. These parts are called `now` and `next`. At the start of the request, the existing data is read into an internal `now` hash which is read and displayed to the user.

When the flash is set during a request/response cycle, it's stored in an internal `next` hash. At the response stage, `next` is converted to JSON and stored in the session.

Feeling Lucky?

And that's how cookies, sessions, and the flash works in Lucky! We've made a nice, type-safe wrapper around all things cookies can represent. For example, if you wanted to set a cookie with an expiration date that is HTTP Only, you could write it like this:

```
cookies
  .set(:current_user_id, 123)
  .expires(1.year.from_now)
  .http_only(true)
```

Because Lucky is type safe, you can only pass a Date to the `expires` method and only a boolean to the `http_only` method. The same is true for all the other methods for setting cookie attributes.

This is only the start of what's new in the latest version of Lucky. Check out the Lucky Framework if you'd like to play around with any of these concepts or just try out a new framework!

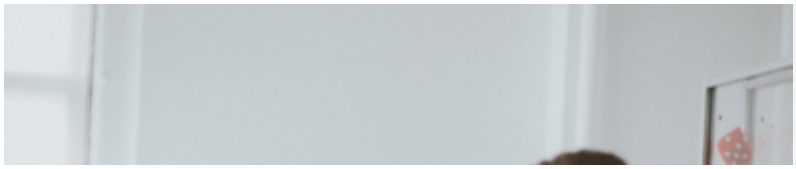
If you want to read more about HTTP Cookies, I highly recommend the MDN page on the topic.

If you enjoyed this post, you might also like:

Access Ruby Hash Values with Fallbacks for Missing Data

Avoiding Out of Memory Crashes on Mobile

HTTP Safety Doesn't Happen by Accident



Upgrade your codebase

thoughtbot

understand the current state of your code quality, speed up delivery times, improve developer happiness, and level up your user experience

Learn more about a Code Audit