Sessions, Users, and Registration

It's time for a confession: we've been deliberately ignoring an incredibly important aspect of Web development prior to this point. So far, we've thought of the traffic visiting our sites as some faceless, anonymous mass hurtling itself against our carefully designed pages.

This isn't true, of course. The browsers hitting our sites have real humans behind them (some of the time, at least). That's a big thing to ignore: the Internet is at its best when it serves to connect people, not machines. If we're going to develop truly compelling sites, eventually we're going to have to deal with the bodies behind the browsers.

Unfortunately, it's not all that easy. HTTP is designed to be stateless—that is, each and every request happens in a vacuum. There's no persistence between one request and the next, and we can't count on any aspects of a request (IP address, user agent, etc.) to consistently indicate successive requests from the same person.

In this chapter you'll learn how to handle this lack of state. We'll start at the lowest level (cookies), and work up to the high-level tools for handling sessions, users, and registration.

Cookies

Browser developers long ago recognized that HTTP's statelessness poses a huge problem for Web developers, and thus cookies were born. A cookie is a small piece of information that browsers store on behalf of Web servers. Every time a browser requests a page from a certain server, it gives back the cookie that it initially received.

Let's take a look how this might work. When you open your browser and type in google.com, your browser sends an HTTP request to Google that starts something like this:

GET / HTTP/1.1
Host: google.com
...
When Google replies, the HTTP response looks something like the following:

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671;
    expires=Sun, 17-Jan-2038 19:14:07 GMT;
    path=/; domain=.google.com
Server: GWS/2.1
...

Notice the Set-Cookie header. Your browser will store that cookie value (PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671) and serve it back to Google every time you access the site. So the next time you access Google, your browser is going to send a request like this:

GET / HTTP/1.1
Host: google.com
Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671
...

Google then can use that Cookie value to know that you're the same person who accessed the site earlier. This value might, for example, be a key into a database that stores user information. Google could (and does) use it to display your name on the page.

Getting and Setting Cookies

When dealing with persistence in Django, most of the time you'll want to use the higher-level session and/or user frameworks discussed a little later in this chapter. However, we'll pause and look at how to read and write cookies at a low level. This should help you understand how the rest of the tools discussed in the chapter actually work, and it will come in handy if you ever need to play with cookies directly.

Reading cookies that are already set is incredibly simple. Every request object has a `COOKIES` object that acts like a dictionary; you can use it to read any cookies that the browser has sent to the view:

```python
def show_color(request):
    if "favorite_color" in request.COOKIES:
        return HttpResponse("Your favorite color is %s" % \
            request.COOKIES["favorite_color"])
    else:
        return HttpResponse("You don't have a favorite color.")
```

Writing cookies is slightly more complicated. You need to use the `set_cookie()` method on an `HttpResponse` object. Here's an example that sets the `favorite_color` cookie based on a `GET` parameter:

```python
def set_color(request):
    if "favorite_color" in request.GET:

        # Create an HttpResponse object...
```