Chapter 14: Sessions, Users, and Registration

It’s time for a confession: we’ve been deliberately ignoring an 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 (most 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 account's username 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, first 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 simple. Every `HttpRequest` 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:

```
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:

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

        # Create an HttpResponse object...
        response = HttpResponse("Your favorite color is 
now %s" % \
            request.GET["favorite_color"])

        # ... and set a cookie on the response
        response.set_cookie("favorite_color",

request.GET["favorite_color"])

        return response

    else:
        return HttpResponse("You didn't give a favorite 
color.")
```

You can also pass a number of optional arguments to `response.set_cookie()` that control aspects of the cookie, as shown in Table 14-1.

Table 14-1: Cookie options

Parameter	Default	Description
`max_age`	`None`	Age (in seconds) that the cookie should last. If this parameter is `None`, the cookie will last only until the browser is closed.
`expires`	`None`	The actual date/time when the cookie should expire. It needs to be in the format `"Wdy, DD-Mth-YY HH:MM:SS GMT"`. If given, this parameter overrides the `max_age` parameter.
`path`	`"/"`	The path prefix that this cookie is valid for. Browsers will only pass the cookie back to pages below this path prefix, so you can use this to prevent cookies from being sent to other sections of your site.

Table 14-1: Cookie options

Parameter	Default	Description
		This is especially useful when you don't control the top level of your site's domain.
`domain`	`None`	The domain that this cookie is valid for. You can use this parameter to set a cross-domain cookie. For example, `domain=".example.com"` will set a cookie that is readable by the domains `www.example.com`,`www2.example.com`, and `an.other.sub.domain.example.com`. If this parameter is set to `None`, a cookie will only be readable by the domain that set it.
`secure`	`False`	If set to `True`, this parameter instructs the browser to only return this cookie to pages accessed over HTTPS.

The Mixed Blessing of Cookies

You might notice a number of potential problems with the way cookies work. Let's look at some of the more important ones:

- Storage of cookies is voluntary; a client does not have to accept or store cookies. In fact, all browsers enable users to control the policy for accepting cookies. If you want to see just how vital cookies are to the Web, try turning on your browser's "prompt to accept every cookie" option.

  Despite their nearly universal use, cookies are still the definition of unreliability. This means that developers should check that a user actually accepts cookies before relying on them.

- Cookies (especially those not sent over HTTPS) are not secure. Because HTTP data is sent in cleartext, cookies are extremely vulnerable to snooping attacks. That is, an attacker snooping on the wire can intercept a cookie and read it. This means you should never store sensitive information in a cookie.

  There's an even more insidious attack, known as a *man-in-the-middle* attack, wherein an attacker intercepts a cookie and uses it to pose as another user. Chapter 20 discusses attacks of this nature in depth, as well as ways to prevent it.

- Cookies aren't even secure from their intended recipients. Most browsers provide easy ways to edit the content of individual cookies, and resourceful users can always use tools like mechanize (http://wwwsearch.sourceforge.net/mechanize/) to construct HTTP requests by hand.

  So you can't store data in cookies that might be sensitive to tampering. The canonical mistake in this scenario is storing something like `IsLoggedIn=1` in a cookie when a user logs in. You'd be amazed at the number of sites that make mistakes of this nature; it takes only a second to fool these sites' "security" systems.

Django's Session Framework

With all of these limitations and potential security holes, it's obvious that cookies and persistent sessions are examples of those "pain points" in Web development. Of course, Django's goal is to be an effective painkiller, so it comes with a session framework designed to smooth over these difficulties for you.

This session framework lets you store and retrieve arbitrary data on a per-site visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies use only a hashed session ID – not the data itself – thus protecting you from most of the common cookie problems.

Let’s look at how to enable sessions and use them in views.

Enabling Sessions

Sessions are implemented via a piece of middleware (see Chapter 17) and a Django model. To enable sessions, you’ll need to follow these steps:

1. Edit your `MIDDLEWARE_CLASSES` setting and make sure `MIDDLEWARE_CLASSES` contains`'django.contrib.sessions.middleware.SessionMiddleware'`.
2. Make sure `'django.contrib.sessions'` is in your `INSTALLED_APPS` setting (and run `manage.py syncdb` if you have to add it).

The default skeleton settings created by `startproject` have both of these bits already installed, so unless you’ve removed them, you probably don’t have to change anything to get sessions to work.

If you don’t want to use sessions, you might want to remove the `SessionMiddleware` line from`MIDDLEWARE_CLASSES` and `'django.contrib.sessions'` from your `INSTALLED_APPS`. It will save you only a small amount of overhead, but every little bit counts.

Using Sessions in Views

When `SessionMiddleware` is activated, each `HttpRequest` object – the first argument to any Django view function – will have a `session` attribute, which is a dictionary-like object. You can read it and write to it in the same way you’d use a normal dictionary. For example, in a view you could do stuff like this:

```
# Set a session value:

request.session["fav_color"] = "blue"



# Get a session value -- this could be called in a
different view,

# or many requests later (or both):

fav_color = request.session["fav_color"]



# Clear an item from the session:
```

```
del request.session["fav_color"]


# Check if the session has a given key:

if "fav_color" in request.session:

    ...
```

You can also use other dictionary methods like `keys()` and `items()` on `request.session`.

There are a couple of simple rules for using Django's sessions effectively:

- Use normal Python strings as dictionary keys on `request.session` (as opposed to integers, objects, etc.).
- Session dictionary keys that begin with an underscore are reserved for internal use by Django. In practice, the framework uses only a small number of underscore-prefixed session variables, but unless you know what they all are (and you are willing to keep up with any changes in Django itself), staying away from underscore prefixes will keep Django from interfering with your application.

  For example, don't use a session key called `_fav_color`, like this:

```
request.session['_fav_color'] = 'blue' # Don't do this!
```

- Don't replace `request.session` with a new object, and don't access or set its attributes. Use it like a Python dictionary. Examples:

```
request.session = some_other_object # Don't do this!

request.session.foo = 'bar' # Don't do this!
```

Let's take a look at a few quick examples. This simplistic view sets a `has_commented` variable to `True` after a user posts a comment. It's a simple (if not particularly secure) way of preventing a user from posting more than one comment:

```
def post_comment(request):
    if request.method != 'POST':
        raise Http404('Only POSTs are allowed')


    if 'comment' not in request.POST:
        raise Http404('Comment not submitted')


    if request.session.get('has_commented', False):

        return HttpResponse("You've already
commented.")


    c =
comments.Comment(comment=request.POST['comment'])

    c.save()

    request.session['has_commented'] = True

    return HttpResponse('Thanks for your comment!')
```

This simplistic view logs in a “member” of the site:

```
def login(request):

    if request.method != 'POST':

        raise Http404('Only POSTs are allowed')

    try:

        m =
Member.objects.get(username=request.POST['username'])

        if m.password == request.POST['password']:

            request.session['member_id'] = m.id

            return HttpResponseRedirect('/you-are-
logged-in/')

    except Member.DoesNotExist:

        return HttpResponse("Your username and password
didn't match.")
```

And this one logs out a member who has been logged in via `login()` above:

```
def logout(request):
    try:

        del request.session['member_id']

    except KeyError:

        pass

    return HttpResponse("You're logged out.")
```

Note

In practice, this is a lousy way of logging users in. The authentication framework discussed shortly handles this task for you in a much more robust and useful manner. These examples are deliberately simplistic so that you can easily see what's going on.

Setting Test Cookies

As mentioned above, you can't rely on every browser accepting cookies. So, as a convenience, Django provides an easy way to test whether a user's browser accepts cookies. Just call`request.session.set_test_cookie()` in a view, and check `request.session.test_cookie_worked()` in a subsequent view – not in the same view call.

This awkward split between `set_test_cookie()` and `test_cookie_worked()` is necessary due to the way cookies work. When you set a cookie, you can't actually tell whether a browser accepted it until the browser's next request.

It's good practice to use `delete_test_cookie()` to clean up after yourself. Do this after you've verified that the test cookie worked.

Here's a typical usage example:

```
def login(request):



    # If we submitted the form...

    if request.method == 'POST':



        # Check that the test cookie worked (we set it 
below):

        if request.session.test_cookie_worked():
            # The test cookie worked, so delete it.

            request.session.delete_test_cookie()


            # In practice, we'd need some logic to
check username/password

            # here, but since this is an example...

            return HttpResponse("You're logged in.")


        # The test cookie failed, so display an error
message. If this

        # were a real site, we'd want to display a
friendlier message.

        else:

            return HttpResponse("Please enable cookies
and try again.")


    # If we didn't post, send the test cookie along
with the login form.

    request.session.set_test_cookie()

    return render(request, 'foo/login_form.html')
```

Note

Again, the built-in authentication functions handle this check for you.

Using Sessions Outside of Views

Internally, each session is just a normal Django model defined in `django.contrib.sessions.models`. Each session is identified by a more-or-less random 32-character hash stored in a cookie. Because it's a normal model, you can access sessions using the normal Django database API:

```
>>> from django.contrib.sessions.models import Session
>>> s =
Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceea
d')

>>> s.expire_date

datetime.datetime(2005, 8, 20, 13, 35, 12)
```

You'll need to call `get_decoded()` to get the actual session data. This is necessary because the dictionary is stored in an encoded format:

```
>>> s.session_data

'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj
...'

>>> s.get_decoded()

{'user_id': 42}
```

When Sessions Are Saved

By default, Django only saves to the database if the session has been modified – that is, if any of its dictionary values have been assigned or deleted:

```
# Session is modified.

request.session['foo'] = 'bar'


# Session is modified.

del request.session['foo']


# Session is modified.

request.session['foo'] = {}


# Gotcha: Session is NOT modified, because this alters

# request.session['foo'] instead of request.session.

request.session['foo']['bar'] = 'baz'
```

To change this default behavior, set SESSION_SAVE_EVERY_REQUEST to True. If SESSION_SAVE_EVERY_REQUEST isTrue, Django will save the session to the database on every single request, even if it wasn't changed.

Note that the session cookie is sent only when a session has been created or modified. IfSESSION_SAVE_EVERY_REQUEST is True, the session cookie will be sent on every request. Similarly, the expirespart of a session cookie is updated each time the session cookie is sent.

Browser-Length Sessions vs. Persistent Sessions

You might have noticed that the cookie Google sent us at the beginning of this chapter containedexpires=Sun, 17-Jan-2038 19:14:07 GMT;. Cookies can optionally contain an expiration date that advises the browser on when to remove the cookie. If a cookie doesn't contain an expiration value, the browser will expire it when the user closes his or her browser window. You can control the session framework's behavior in this regard with the SESSION_EXPIRE_AT_BROWSER_CLOSE setting.

By default, SESSION_EXPIRE_AT_BROWSER_CLOSE is set to False, which means session cookies will be stored in users' browsers for SESSION_COOKIE_AGE seconds (which defaults to two weeks, or 1,209,600 seconds). Use this if you don't want people to have to log in every time they open a browser.

If SESSION_EXPIRE_AT_BROWSER_CLOSE is set to True, Django will use browser-length cookies.

Other Session Settings

Besides the settings already mentioned, a few other settings influence how Django's session framework uses cookies, as shown in Table 14-2.

Table 14-2. Settings that influence cookie behavior

Setting	Description	Default
SESSION_COOKIE_DOMAIN	The domain to use for session cookies. Set this to a string such as ".example.com" for cross-domain cookies, or use None for a standard cookie.	None
SESSION_COOKIE_NAME	The name of the cookie to use for sessions. This can be any string.	"sessionid"
SESSION_COOKIE_SECURE	Whether to use a "secure" cookie for the session cookie. If this is set to True, the cookie will be marked as "secure," which means that browsers will ensure that the cookie is only sent via HTTPS.	False

Technical Details

For the curious, here are a few technical notes about the inner workings of the session framework:

- The session dictionary accepts any Python object capable of being "pickled." See the documentation for Python's built-in pickle module for information about how this works.
- Session data is stored in a database table named django_session.

- Session data is fetched upon demand. If you never access `request.session`, Django won't hit that database table.
- Django only sends a cookie if it needs to. If you don't set any session data, it won't send a session cookie (unless `SESSION_SAVE_EVERY_REQUEST` is set to `True`).
- The Django sessions framework is entirely, and solely, cookie based. It does not fall back to putting session IDs in URLs as a last resort, as some other tools (PHP, JSP) do.

  This is an intentional design decision. Putting sessions in URLs don't just make URLs ugly, but also make your site vulnerable to a certain form of session ID theft via the `Referer`header.

  If you're still curious, the source is pretty straightforward; look in `django.contrib.sessions` for more details.

Users and Authentication

Sessions give us a way of persisting data through multiple browser requests; the second part of the equation is using those sessions for user login. Of course, we can't just trust that users are who they say they are, so we need to authenticate them along the way.

Naturally, Django provides tools to handle this common task (and many others). Django's user authentication system handles user accounts, groups, permissions, and cookie-based user sessions. This system is often referred to as an *auth/auth* (authentication and authorization) system. That name recognizes that dealing with users is often a two-step process. We need to

1. Verify (*authenticate*) that a user is who he or she claims to be (usually by checking a username and password against a database of users)
2. Verify that the user is *authorized* to perform some given operation (usually by checking against a table of permissions)

Following these needs, Django's auth/auth system consists of a number of parts:

- *Users*: People registered with your site
- *Permissions*: Binary (yes/no) flags designating whether a user may perform a certain task
- *Groups*: A generic way of applying labels and permissions to more than one user
- *Messages*: A simple way to queue and display system messages to users

If you've used the admin tool (discussed in Chapter 6), you've already seen many of these tools, and if you've edited users or groups in the admin tool, you've actually been editing data in the auth system's database tables.

Enabling Authentication Support

Like the session tools, authentication support is bundled as a Django application in `django.contrib` that needs to be installed. Also like the session tools, it's also installed by default, but if you've removed it, you'll need to follow these steps to install it:

1. Make sure the session framework is installed as described earlier in this chapter. Keeping track of users obviously requires cookies, and thus builds on the session framework.
2. Put `'django.contrib.auth'` in your `INSTALLED_APPS` setting and run `manage.py syncdb` to install the appropriate database tables.

3. Make sure that `'django.contrib.auth.middleware.AuthenticationMiddleware'` is in your`MIDDLEWARE_CLASSES` setting – *after* `SessionMiddleware`.

   With that installation out of the way, we're ready to deal with users in view functions. The main interface you'll use to access users within a view is `request.user`; this is an object that represents the currently logged-in user. If the user isn't logged in, this will instead be an `AnonymousUser` object (see below for more details).

   You can easily tell if a user is logged in with the `is_authenticated()` method:

```
if request.user.is_authenticated():

    # Do something for authenticated users.

else:

    # Do something for anonymous users.
```

Using Users

Once you have a `User` – often from `request.user`, but possibly through one of the other methods discussed shortly – you have a number of fields and methods available on that object. `AnonymousUser` objects emulate*some* of this interface, but not all of it, so you should always check `user.is_authenticated()` before assuming you're dealing with a bona fide user object. Tables 14-3 and 14-4 list the fields and methods, respectively, on `User` objects.

Table 14-3. Fields on `User` Objects

Field	**Description**
`username`	Required; 30 characters or fewer. Alphanumeric characters only (letters, digits, and underscores).
`first_name`	Optional; 30 characters or fewer.
`last_name`	Optional; 30 characters or fewer.
`email`	Optional. E-mail address.
`password`	Required. A hash of, and metadata about, the password (Django doesn't store the raw password). See the "Passwords" section for more about this value.
`is_staff`	Boolean. Designates whether this user can access the admin site.
`is_active`	Boolean. Designates whether this account can be used to log in. Set this flag to`False` instead of deleting accounts.
`is_superuser`	Boolean. Designates that this user has all permissions without explicitly assigning them.
`last_login`	A datetime of the user's last login. This is set to the current date/time by default.
`date_joined`	A datetime designating when the account was created. This is set to the current date/time by default when the account is created.

Table 14-4. Methods on `User` Objects

Method	**Description**
`is_authenticated()`	Always returns `True` for “real” `User` objects. This is a way to tell if the user has been authenticated. This does not imply any permissions, and it doesn’t check if the user is active. It only indicates that the user has sucessfully authenticated.
`is_anonymous()`	Returns `True` only for `AnonymousUser` objects (and `False` for “real” `User` objects). Generally, you should prefer using`is_authenticated()` to this method.
`get_full_name()`	Returns the `first_name` plus the `last_name`, with a space in between.
`set_password(passwd)`	Sets the user’s password to the given raw string, taking care of the password hashing. This doesn’t actually save the`User` object.
`check_password(passwd)`	Returns `True` if the given raw string is the correct password for the user. This takes care of the password hashing in making the comparison.
`get_group_permissions()`	Returns a list of permission strings that the user has through the groups he or she belongs to.
`get_all_permissions()`	Returns a list of permission strings that the user has, both through group and user permissions.
`has_perm(perm)`	Returns `True` if the user has the specified permission, where`perm` is in the format `"package.codename"`. If the user is inactive, this method will always return `False`.
`has_perms(perm_list)`	Returns `True` if the user has *all* of the specified permissions. If the user is inactive, this method will always return `False`.
`has_module_perms(app_label)`	Returns `True` if the user has any permissions in the given`app_label`. If the user is inactive, this method will always return `False`.
`get_and_delete_messages()`	Returns a list of `Message` objects in the user’s queue and deletes the messages from the queue.
`email_user(subj, msg)`	Sends an email to the user. This email is sent from the`DEFAULT_FROM_EMAIL` setting. You can also pass a third argument, `from_email`, to override the From address on the email.

Finally, `User` objects have two many-to-many fields: `groups` and `permissions`. `User` objects can access their related objects in the same way as any other many-to-many field:

```
# Set a user's groups:

myuser.groups = group_list



# Add a user to some groups:

myuser.groups.add(group1, group2,...)
```

```
# Remove a user from some groups:

myuser.groups.remove(group1, group2,...)


# Remove a user from all groups:

myuser.groups.clear()


# Permissions work the same way

myuser.permissions = permission_list

myuser.permissions.add(permission1, permission2, ...)

myuser.permissions.remove(permission1, permission2,
...)

myuser.permissions.clear()
```

Logging In and Out

Django provides built-in view functions for handling logging in and out (and a few other nifty tricks), but before we get to those, let's take a look at how to log users in and out "by hand." Django provides two functions to perform these actions in `django.contrib.auth`: `authenticate()` and `login()`.

To authenticate a given username and password, use `authenticate()`. It takes two keyword arguments,`username` and `password`, and it returns a `User` object if the password is valid for the given username. If the password is invalid, `authenticate()` returns `None`:

```
>>> from django.contrib import auth

>>> user = auth.authenticate(username='john',
password='secret')

>>> if user is not None:

...     print "Correct!"

... else:

...     print "Invalid password."
```

`authenticate()` only verifies a user's credentials. To log in a user, use `login()`. It takes an `HttpRequest`object and a `User` object and saves the user's ID in the session, using Django's session framework.

This example shows how you might use both `authenticate()` and `login()` within a view function:

```
from django.contrib import auth


def login_view(request):
    username = request.POST.get('username', '')
    password = request.POST.get('password', '')
    user = auth.authenticate(username=username,
password=password)
    if user is not None and user.is_active:
        # Correct password, and the user is marked
"active"
        auth.login(request, user)
        # Redirect to a success page.
        return
HttpResponseRedirect("/account/loggedin/")
    else:
        # Show an error page
        return
HttpResponseRedirect("/account/invalid/")
```

To log out a user, use `django.contrib.auth.logout()` within your view. It takes an `HttpRequest` object and has no return value:

```
from django.contrib import auth


def logout_view(request):
    auth.logout(request)
    # Redirect to a success page.
    return HttpResponseRedirect("/account/loggedout/")
```

Note that `auth.logout()` doesn’t throw any errors if the user wasn’t logged in.

In practice, you usually will not need to write your own login/logout functions; the authentication system comes with a set of views for generically handling logging in and out. The first step in using these authentication views is to wire them up in your URLconf. You’ll need to add this snippet:

```
from django.contrib.auth.views import login, logout


urlpatterns = patterns('',
    # existing patterns here...
    (r'^accounts/login/$',  login),
    (r'^accounts/logout/$', logout),
)
```

`/accounts/login/` and `/accounts/logout/` are the default URLs that Django uses for these views.

By default, the `login` view renders a template at `registration/login.html` (you can change this template name by passing an extra view argument ,``template_name``). This form needs to contain a `username` and a `password` field. A simple template might look like this:

```
{% extends "base.html" %}


{% block content %}


  {% if form.errors %}
    <p class="error">Sorry, that's not a valid username
or password</p>
  {% endif %}

  <form action="" method="post">

    <label for="username">User name:</label>

    <input type="text" name="username" value=""
id="username">

    <label for="password">Password:</label>

    <input type="password" name="password" value=""
id="password">



    <input type="submit" value="login" />

    <input type="hidden" name="next" value="{{
next|escape }}" />

  </form>



{% endblock %}
```

If the user successfully logs in, he or she will be redirected to `/accounts/profile/` by default. You can override this by providing a hidden field called `next` with the URL to redirect to after logging in. You can also pass this value as a `GET` parameter to the login view and it will be automatically added to the context as a variable called `next` that you can insert into that hidden field.

The logout view works a little differently. By default it renders a template at `registration/logged_out.html`(which usually contains a “You’ve successfully logged out” message). However, you can call the view with an extra argument, `next_page`, which will instruct the view to redirect after a logout.

Limiting Access to Logged-in Users

Of course, the reason we’re going through all this trouble is so we can limit access to parts of our site.

The simple, raw way to limit access to pages is to check `request.user.is_authenticated()` and redirect to a login page:

```
from django.http import HttpResponseRedirect



def my_view(request):
    if not request.user.is_authenticated():

        return
HttpResponseRedirect('/accounts/login/?next=%s' %
request.path)

    # ...
```

or perhaps display an error message:

```
def my_view(request):

    if not request.user.is_authenticated():

        return render(request,
'myapp/login_error.html')

    # ...
```

As a shortcut, you can use the convenient `login_required` decorator:

```
from django.contrib.auth.decorators import
login_required



@login_required

def my_view(request):

    # ...
```

`login_required` does the following:

- If the user isn't logged in, redirect to `/accounts/login/`, passing the current URL path in the query string as `next`, for example: `/accounts/login/?next=/polls/3/`.
- If the user is logged in, execute the view normally. The view code can then assume that the user is logged in.

Limiting Access to Users Who Pass a Test

Limiting access based on certain permissions or some other test, or providing a different location for the login view works essentially the same way.

The raw way is to run your test on `request.user` in the view directly. For example, this view checks to make sure the user is logged in and has the permission `polls.can_vote` (more about how permissions works follows):

```
def vote(request):

    if request.user.is_authenticated() and
request.user.has_perm('polls.can_vote')):

        # vote here

    else:

        return HttpResponse("You can't vote in this
poll.")
```

Again, Django provides a shortcut called `user_passes_test`. It takes arguments and generates a specialized decorator for your particular situation:

```
def user_can_vote(user):

    return user.is_authenticated() and
user.has_perm("polls.can_vote")



@user_passes_test(user_can_vote, login_url="/login/")

def vote(request):

    # Code here can assume a logged-in user with the
correct permission.

    ...
```

`user_passes_test` takes one required argument: a callable that takes a `User` object and returns `True` if the user is allowed to view the page. Note that `user_passes_test` does not automatically check that the `User` is authenticated; you should do that yourself.

In this example we're also showing the second (optional) argument, `login_url`, which lets you specify the URL for your login page (`/accounts/login/` by default). If the user doesn't pass the test, then the`user_passes_test` decorator will redirect the user to the `login_url`.

Because it's a relatively common task to check whether a user has a particular permission, Django provides a shortcut for that case: the `permission_required()` decorator. Using this decorator, the earlier example can be written as follows:

```
from django.contrib.auth.decorators import
permission_required
```

```
@permission_required('polls.can_vote',
login_url="/login/")

def vote(request):

    # ...
```

Note that `permission_required()` also takes an optional `login_url` parameter, which also defaults to `'/accounts/login/'`.

Limiting Access to Generic Views

One of the most frequently asked questions on the Django users list deals with limiting access to a generic view. To pull this off, you'll need to write a thin wrapper around the view and point your URLconf to your wrapper instead of the generic view itself:

```
from django.contrib.auth.decorators import
login_required

from django.views.generic.date_based import
object_detail



@login_required

def limited_object_detail(*args, **kwargs):

    return object_detail(*args, **kwargs)
```

You can, of course, replace `login_required` with any of the other limiting decorators.

Managing Users, Permissions, and Groups

The easiest way by far to manage the auth system is through the admin interface. Chapter 6 discusses how to use Django's admin site to edit users and control their permissions and access, and most of the time you'll just use that interface.

However, there are low-level APIs you can dive into when you need absolute control, and we discuss these in the sections that follow.

Creating Users

Create users with the `create_user` helper function:

```
>>> from django.contrib.auth.models import User

>>> user = User.objects.create_user(username='john',
...
email='jlennon@beatles.com',

...                                         password='glass
onion')
```

At this point, `user` is a `User` instance ready to be saved to the database (`create_user()` doesn't actually call`save()` itself). You can continue to change its attributes before saving, too:

```
>>> user.is_staff = True

>>> user.save()
```

Changing Passwords

You can change a password with `set_password()`:

```
>>> user = User.objects.get(username='john')

>>> user.set_password('goo goo goo joob')

>>> user.save()
```

Don't set the `password` attribute directly unless you know what you're doing. The password is actually stored as a *salted hash* and thus can't be edited directly.

More formally, the `password` attribute of a `User` object is a string in this format:

```
hashtype$salt$hash
```

That's a hash type, the salt, and the hash itself, separated by the dollar sign ($) character.

`hashtype` is either `sha1` (default) or `md5`, the algorithm used to perform a one-way hash of the password.`salt` is a random string used to salt the raw password to create the hash, for example:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

The `User.set_password()` and `User.check_password()` functions handle the setting and checking of these values behind the scenes.

Salted hashes

A *hash* is a one-way cryptographic function – that is, you can easily compute the hash of a given value, but it's nearly impossible to take a hash and reconstruct the original value.

If we stored passwords as plain text, anyone who got their hands on the password database would instantly know everyone’s password. Storing passwords as hashes reduces the value of a compromised database.

However, an attacker with the password database could still run a *brute- force* attack, hashing millions of passwords and comparing those hashes against the stored values. This takes some time, but less than you might think.

Worse, there are publicly available *rainbow tables*, or databases of pre-computed hashes of millions of passwords. With a rainbow table, an experienced attacker could break most passwords in seconds.

Adding a *salt* – basically an initial random value – to the stored hash adds another layer of difficulty to breaking passwords. Because salts differ from password to password, they also prevent the use of a rainbow table, thus forcing attackers to fall back on a brute-force attack, itself made more difficult by the extra entropy added to the hash by the salt.

While salted hashes aren’t absolutely the most secure way of storing passwords, they’re a good middle ground between security and convenience.

Handling Registration

We can use these low-level tools to create views that allow users to sign up for new accounts. Different developers implement registration differently, so Django leaves writing a registration view up to you. Luckily, it’s pretty easy.

At its simplest, we could provide a small view that prompts for the required user information and creates those users. Django provides a built-in form you can use for this purpose, which we’ll use in this example:

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.http import HttpResponseRedirect
from django.shortcuts import render


def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            new_user = form.save()
            return HttpResponseRedirect("/books/")
    else:

        form = UserCreationForm()

    return render(request, 
"registration/register.html", {

        'form': form,

    })
```

This form assumes a template named `registration/register.html`. Here's an example of what that template might look like:

```
{% extends "base.html" %}


{% block title %}Create an account{% endblock %}


{% block content %}

  <h1>Create an account</h1>


  <form action="" method="post">

      {{ form.as_p }}

      <input type="submit" value="Create the account">

  </form>

{% endblock %}
```

Using Authentication Data in Templates

The current logged-in user and his or her permissions are made available in the template context when you use the `render()` shortcut or explicitly use a `RequestContext` (see Chapter 9).

Note

Technically, these variables are only made available in the template context if you use`RequestContext` *and* your `TEMPLATE_CONTEXT_PROCESSORS` setting contains`"django.core.context_processors.auth"`, which is the default. Again, see Chapter 9 for more information.

When using `RequestContext`, the current user (either a `User` instance or an `AnonymousUser` instance) is stored in the template variable `{{ user }}`:

```
{% if user.is_authenticated %}

  <p>Welcome, {{ user.username }}. Thanks for logging in.</p>

{% else %}

  <p>Welcome, new user. Please log in.</p>

{% endif %}
```

This user's permissions are stored in the template variable `{{ perms }}`. This is a template-friendly proxy to a couple of permission methods described shortly.

There are two ways you can use this `perms` object. You can use something like `{% if perms.polls %}` to check if the user has *any* permissions for some given application, or you can use something like `{% if perms.polls.can_vote %}` to check if the user has a specific permission.

Thus, you can check permissions in template `{% if %}` statements:

```
{% if perms.polls %}

  <p>You have permission to do something in the polls app.</p>

  {% if perms.polls.can_vote %}

    <p>You can vote!</p>

  {% endif %}

{% else %}

  <p>You don't have permission to do anything in the polls app.</p>

{% endif %}
```

Permissions, Groups and Messages

There are a few other bits of the authentication framework that we've only dealt with in passing. We'll take a closer look at them in the following sections.

Permissions

Permissions are a simple way to "mark" users and groups as being able to perform some action. They are usually used by the Django admin site, but you can easily use them in your own code.

The Django admin site uses permissions as follows:

- Access to view the "add" form, and add an object is limited to users with the *add* permission for that type of object.
- Access to view the change list, view the "change" form, and change an object is limited to users with the *change* permission for that type of object.
- Access to delete an object is limited to users with the *delete* permission for that type of object.

Permissions are set globally per type of object, not per specific object instance. For example, it's possible to say "Mary may change news stories," but permissions don't let you say "Mary may change news stories, but only the ones she created herself" or "Mary may only change news stories that have a certain status, publication date, or ID."

These three basic permissions – add, change, and delete – are automatically created for each Django model. Behind the scenes, these permissions are added to the `auth_permission` database table when you run `manage.py syncdb`.

These permissions will be of the form `"<app>.<action>_<object_name>"`. That is, if you have a `polls`application with a `Choice` model, you'll get permissions named `"polls.add_choice"`, `"polls.change_choice"`, and `"polls.delete_choice"`.

Just like users, permissions are implemented in a Django model that lives in `django.contrib.auth.models`. This means that you can use Django's database API to interact directly with permissions if you like.

Groups

Groups are a generic way of categorizing users so you can apply permissions, or some other label, to those users. A user can belong to any number of groups.

A user in a group automatically has the permissions granted to that group. For example, if the group`Site editors` has the permission `can_edit_home_page`, any user in that group will have that permission.

Groups are also a convenient way to categorize users to give them some label, or extended functionality. For example, you could create a group `'Special users'`, and you could write code that could, say, give those users access to a members-only portion of your site, or send them members-only e-mail messages.

Like users, the easiest way to manage groups is through the admin interface. However, groups are also just Django models that live in `django.contrib.auth.models`, so once again you can always use Django's database APIs to deal with groups at a low level.

Messages

The message system is a lightweight way to queue messages for given users. A message is associated with a `User`. There's no concept of expiration or timestamps.

Messages are used by the Django admin interface after successful actions. For example, when you create an object, you'll notice a "The object was created successfully" message at the top of the admin page.

You can use the same API to queue and display messages in your own application. The API is simple:

- To create a new message, use `user.message_set.create(message='message_text')`.
- To retrieve/delete messages, use `user.get_and_delete_messages()`, which returns a list of `Message`objects in the user's queue (if any) and deletes the messages from the queue.

In this example view, the system saves a message for the user after creating a playlist:

```
def create_playlist(request, songs):

    # Create the playlist with the given songs.

    # ...

    request.user.message_set.create(

        message="Your playlist was added successfully."

    )

    return render(request, "playlists/create.html")
```

When you use the `render()` shortcut or render a template with a `RequestContext`, the current logged-in user and his or her messages are made available in the template context as the template variable`{{ messages }}`. Here's an example of template code that displays messages:

```
{% if messages %}

<ul>

    {% for message in messages %}

    <li>{{ message }}</li>

    {% endfor %}

</ul>

{% endif %}
```

Note that `RequestContext` calls `get_and_delete_messages` behind the scenes, so any messages will be deleted even if you don't display them.

Finally, note that this messages framework only works with users in the user database. To send messages to anonymous users, use the session framework directly.