User authentication in Django

Django comes with a user authentication system. It handles user accounts, groups, permissions and cookie-based user sessions. This document explains how things work.

Overview

The auth system consists of:

- Users
- 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 messages for given users.

Deprecated in Django 1.2:

Deprecated since version 1.2: The Messages component of the auth system will be removed in Django 1.4.

Installation

Authentication support is bundled as a Django application in `django.contrib.auth`. To install it, do the following:

1. Put `'django.contrib.auth'` and `'django.contrib.contenttypes'` in your `INSTALLED_APPS` setting. (The `Permission` model in `django.contrib.auth` depends on `django.contrib.contenttypes`.)
2. Run the command `manage.py syncdb`.

Note that the default `settings.py` file created by `django-admin.py startproject` includes `'django.contrib.auth'` and `'django.contrib.contenttypes'` in `INSTALLED_APPS` for convenience. If your `INSTALLED_APPS` already contains these apps, feel free to run `manage.py syncdb` again; you can run that command as many times as you'd like, and each time it'll only install what's needed.

The `syncdb` command creates the necessary database tables, creates permission objects for all installed apps that need 'em, and prompts you to create a superuser account the first time you run it.

Once you've taken those steps, that's it.

Users

class **models.User**

API reference

Fields

class **models.User**

`User` objects have the following fields:

username

Required. 30 characters or fewer. Alphanumeric characters only (letters, digits and underscores).

Changed in Django 1.2: Usernames may now contain `@`, `+`, `.` and `-` characters.

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.) Raw passwords can be arbitrarily long and can contain any character. See the "Passwords" section below.

is_staff

Browse
- Prev: Testing Django applications
- Next: Django's cache framework

You are here:
- Django 1.3.7 documentation
  - Using Django
    - User authentication in Django

This Page
- Show Source

Quick search

Boolean. Designates whether this user can access the admin site.

is_active

Boolean. Designates whether this user account should be considered active. We recommend that you set this flag to `False` instead of deleting accounts; that way, if your applications have any foreign keys to users, the foreign keys won't break.

This doesn't necessarily control whether or not the user can log in. Authentication backends aren't required to check for the `is_active` flag, so if you want to reject a login based on `is_active` being `False`, it's up to you to check that in your own login view. However, the `AuthenticationForm` used by the `login()` view *does* perform this check, as do the permission-checking methods such as `has_perm()` and the authentication in the Django admin. All of those functions/methods will return `False` for inactive users.

is_superuser

Boolean. Designates that this user has all permissions without explicitly assigning them.

last_login

A datetime of the user's last login. Is set to the current date/time by default.

date_joined

A datetime designating when the account was created. Is set to the current date/time by default when the account is created.

Methods

class **models.User**

`User` objects have two many-to-many fields: models.User. `groups` and `user_permissions`. `User` objects can access their related objects in the same way as any other Django model:

```
myuser.groups = [group_list]
myuser.groups.add(group, group, ...)
myuser.groups.remove(group, group, ...)
myuser.groups.clear()
myuser.user_permissions = [permission_list]
myuser.user_permissions.add(permission, permission, ...)
myuser.user_permissions.remove(permission, permission, ...)
myuser.user_permissions.clear()
```

In addition to those automatic API methods, `User` objects have the following custom methods:

is_anonymous()

Always returns `False`. This is a way of differentiating `User` and `AnonymousUser` objects. Generally, you should prefer using `is_authenticated()` to this method.

is_authenticated()

Always returns `True`. This is a way to tell if the user has been authenticated. This does not imply any permissions, and doesn't check if the user is active - it only indicates that the user has provided a valid username and password.

get_full_name()

Returns the `first_name` plus the `last_name`, with a space in between.

set_password(*raw_password***)**

Sets the user's password to the given raw string, taking care of the password hashing. Doesn't save the `User` object.

check_password(*raw_password***)**

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.)

set_unusable_password()

Marks the user as having no password set. This isn't the same as having a blank string for a password. `check_password()` for this user will never return `True`. Doesn't save the `User` object.

You may need this if authentication for your application takes place against an existing external source such as an LDAP directory.

has_usable_password()

Returns `False` if `set_unusable_password()` has been called for this user.

`get_group_permissions`(*obj=None*)

Returns a set of permission strings that the user has, through his/her groups.

New in Django 1.2: Please, see the release notes

If `obj` is passed in, only returns the group permissions for this specific object.

`get_all_permissions`(*obj=None*)

Returns a set of permission strings that the user has, both through group and user permissions.

New in Django 1.2: Please, see the release notes

If `obj` is passed in, only returns the permissions for this specific object.

`has_perm`(*perm*, *obj=None*)

Returns `True` if the user has the specified permission, where perm is in the format `"<app label>.<permission codename>"`. (see permissions section below). If the user is inactive, this method will always return `False`.

New in Django 1.2: Please, see the release notes

If `obj` is passed in, this method won't check for a permission for the model, but for this specific object.

`has_perms`(*perm_list*, *obj=None*)

Returns `True` if the user has each of the specified permissions, where each perm is in the format `"<app label>.<permission codename>"`. If the user is inactive, this method will always return `False`.

New in Django 1.2: Please, see the release notes

If `obj` is passed in, this method won't check for permissions for the model, but for the specific object.

`has_module_perms`(*package_name*)

Returns `True` if the user has any permissions in the given package (the Django 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`(*subject*, *message*, *from_email=None*)

Sends an e-mail to the user. If `from_email` is `None`, Django uses the `DEFAULT_FROM_EMAIL`.

`get_profile`()

Returns a site-specific profile for this user. Raises `django.contrib.auth.models.SiteProfileNotAvailable` if the current site doesn't allow profiles. For information on how to define a site-specific user profile, see the section on storing additional user information below.

Manager functions

class **`models.UserManager`**

The `User` model has a custom manager that has the following helper functions:

`create_user`(*username*, *email*, *password=None*)

Creates, saves and returns a `User`.

The `username` and `password` are set as given. The domain portion of `email` is automatically converted to lowercase, and the returned `User` object will have `is_active` set to `True`.

If no password is provided, `set_unusable_password()` will be called.

See Creating users for example usage.

`make_random_password`(*length=10*, *allowed_chars='abcdefghjkmnpqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ23456789'*)

Returns a random password with the given length and given string of allowed characters. (Note that the default value of `allowed_chars` doesn't contain letters that can cause user confusion, including:

- `i`, `l`, `I`, and `1` (lowercase letter i, lowercase letter L, uppercase letter i, and the number one)
- `o`, `O`, and `0` (uppercase letter o, lowercase letter o, and zero)

Basic usage

Creating users

The most basic way to create users is to use the `create_user()` helper function that comes with Django:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')

# At this point, user is a User object that has already been saved
# to the database. You can continue to change its attributes
# if you want to change other fields.
>>> user.is_staff = True
>>> user.save()
```

You can also create users using the Django admin site. Assuming you've enabled the admin site and hooked it to the URL `/admin/`, the "Add user" page is at `/admin/auth/user/add/`. You should also see a link to "Users" in the "Auth" section of the main admin index page. The "Add user" admin page is different than standard admin pages in that it requires you to choose a username and password before allowing you to edit the rest of the user's fields.

Also note: if you want your own user account to be able to create users using the Django admin site, you'll need to give yourself permission to add users *and* change users (i.e., the "Add user" and "Change user" permissions). If your account has permission to add users but not to change them, you won't be able to add users. Why? Because if you have permission to add users, you have the power to create superusers, which can then, in turn, change other users. So Django requires add *and* change permissions as a slight security measure.

Changing passwords

New in Django 1.2: The `manage.py changepassword` command was added.

`manage.py changepassword *username*` offers a method of changing a User's password from the command line. It prompts you to change the password of a given user which you must enter twice. If they both match, the new password will be changed immediately. If you do not supply a user, the command will attempt to change the password whose username matches the current user.

You can also change a password programmatically, using `set_password()`:

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username__exact='john')
>>> u.set_password('new password')
>>> u.save()
```

Don't set the `password` attribute directly unless you know what you're doing. This is explained in the next section.

Passwords

The `password` attribute of a `User` object is a string in this format:

```
hashtype$salt$hash
```

That's hashtype, salt and hash, separated by the dollar-sign character.

Hashtype is either `sha1` (default), `md5` or `crypt` -- 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. Note that the `crypt` method is only supported on platforms that have the standard Python `crypt` module available.

For example:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

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

Previous Django versions, such as 0.90, used simple MD5 hashes without password salts. For backwards compatibility, those are still supported; they'll be converted automatically to the new style the first time `check_password()` works correctly for a given user.

Anonymous users

class **models.AnonymousUser**

`django.contrib.auth.models.AnonymousUser` is a class that implements the `django.contrib.auth.models.User` interface, with these differences:

- `id` is always `None`.
- `is_staff` and `is_superuser` are always `False`.
- `is_active` is always `False`.
- `groups` and `user_permissions` are always empty.

- `is_anonymous()` returns `True` instead of `False`.
- `is_authenticated()` returns `False` instead of `True`.
- `set_password()`, `check_password()`, `save()`, `delete()`, `set_groups()` and `set_permissions()` raise `NotImplementedError`.

In practice, you probably won't need to use `AnonymousUser` objects on your own, but they're used by Web requests, as explained in the next section.

Creating superusers

`manage.py syncdb` prompts you to create a superuser the first time you run it after adding `'django.contrib.auth'` to your `INSTALLED_APPS`. If you need to create a superuser at a later date, you can use a command line utility:

```
manage.py createsuperuser --username=joe --email=joe@example.com
```

You will be prompted for a password. After you enter one, the user will be created immediately. If you leave off the `--username` or the `--email` options, it will prompt you for those values.

If you're using an older release of Django, the old way of creating a superuser on the command line still works:

```
python /path/to/django/contrib/auth/create_superuser.py
```

...where `/path/to` is the path to the Django codebase on your filesystem. The `manage.py` command is preferred because it figures out the correct path and environment for you.

Storing additional information about users

If you'd like to store additional information related to your users, Django provides a method to specify a site-specific related model -- termed a "user profile" -- for this purpose.

To make use of this feature, define a model with fields for the additional information you'd like to store, or additional methods you'd like to have available, and also add a `OneToOneField` named `user` from your model to the `User` model. This will ensure only one instance of your model can be created for each `User`.

To indicate that this model is the user profile model for a given site, fill in the setting `AUTH_PROFILE_MODULE` with a string consisting of the following items, separated by a dot:

1. The name of the application (case sensitive) in which the user profile model is defined (in other words, the name which was passed to `manage.py startapp` to create the application).
2. The name of the model (not case sensitive) class.

For example, if the profile model was a class named `UserProfile` and was defined inside an application named `accounts`, the appropriate setting would be:

```
AUTH_PROFILE_MODULE = 'accounts.UserProfile'
```

When a user profile model has been defined and specified in this manner, each `User` object will have a method -- `get_profile()` -- which returns the instance of the user profile model associated with that `User`.

The method `get_profile()` does not create the profile, if it does not exist. You need to register a handler for the signal `django.db.models.signals.post_save` on the User model, and, in the handler, if created=True, create the associated user profile.

For more information, see Chapter 12 of the Django book.

Authentication in Web requests

Until now, this document has dealt with the low-level APIs for manipulating authentication-related objects. On a higher level, Django can hook this authentication framework into its system of `request objects`.

First, install the `SessionMiddleware` and `AuthenticationMiddleware` middlewares by adding them to your `MIDDLEWARE_CLASSES` setting. See the session documentation for more information.

Once you have those middlewares installed, you'll be able to access `request.user` in views. `request.user` will give you a `User` object representing the currently logged-in user. If a user isn't currently logged in, `request.user` will be set to an instance of `AnonymousUser` (see the previous section). You can tell them apart with `is_authenticated()`, like so:

```
if request.user.is_authenticated():
    # Do something for authenticated users.
else:
    # Do something for anonymous users.
```

How to log a user in

Django provides two functions in `django.contrib.auth`: `authenticate()` and `login()`.

`authenticate()`

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`. Example:

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    if user.is_active:
        print "You provided a correct username and password!"
    else:
        print "Your account has been disabled!"
else:
    print "Your username and password were incorrect."
```

`login()`

To log a user in, in a view, use `login()`. It takes an `HttpRequest` object and a `User` object. `login()` saves the user's ID in the session, using Django's session framework, so, as mentioned above, you'll need to make sure to have the session middleware installed.

This example shows how you might use both `authenticate()` and `login()`:

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(username=username, password=password)
    if user is not None:
        if user.is_active:
            login(request, user)
            # Redirect to a success page.
        else:
            # Return a 'disabled account' error message
    else:
        # Return an 'invalid login' error message.
```

Calling `authenticate()` first

When you're manually logging a user in, you *must* call `authenticate()` before you call `login()`. `authenticate()` sets an attribute on the `User` noting which authentication backend successfully authenticated that user (see the backends documentation for details), and this information is needed later during the login process.

Manually checking a user's password

`check_password()`

If you'd like to manually authenticate a user by comparing a plain-text password to the hashed password in the database, use the convenience function `django.contrib.auth.models.check_password()`. It takes two arguments: the plain-text password to check, and the full value of a user's `password` field in the database to check against, and returns `True` if they match, `False` otherwise.

How to log a user out

`logout()`

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

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # Redirect to a success page.
```

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

When you call `logout()`, the session data for the current request is completely cleaned out. All existing data is removed. This is to prevent another person from using the same Web browser to log in and have access to the previous user's session data. If you want to put anything into the session that will be available to the user immediately after logging out, do that *after* calling `django.contrib.auth.logout()`.

Login and logout signals

New in Django 1.3: Please, see the release notes

The auth framework uses two signals that can be used for notification when a user logs in or out.

`django.contrib.auth.signals.user_logged_in`

Sent when a user logs in successfully.

Arguments sent with this signal:

sender
As above: the class of the user that just logged in.

request
The current `HttpRequest` instance.

user
The user instance that just logged in.

django.contrib.auth.signals.user_logged_out

Sent when the logout method is called.

sender
As above: the class of the user that just logged out or `None` if the user was not authenticated.

request
The current `HttpRequest` instance.

user
The user instance that just logged out or `None` if the user was not authenticated.

Limiting access to logged-in users

The raw way

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

```
from django.http import HttpResponseRedirect

def my_view(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/login/?next=%s' % request.path)
    # ...
```

...or display an error message:

```
def my_view(request):
    if not request.user.is_authenticated():
        return render_to_response('myapp/login_error.html')
    # ...
```

The login_required decorator

decorators.login_required([*redirect_field_name=REDIRECT_FIELD_NAME***,** *login_url=None***])**

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 `settings.LOGIN_URL`, passing the current absolute path in the query string. Example: `/accounts/login/?next=/polls/3/`.
- If the user is logged in, execute the view normally. The view code is free to assume the user is logged in.

By default, the path that the user should be redirected to upon successful authentication is stored in a query string parameter called `"next"`. If you would prefer to use a different name for this parameter, `login_required()` takes an optional `redirect_field_name` parameter:

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

@login_required(redirect_field_name='my_redirect_field')
def my_view(request):
    ...
```

Note that if you provide a value to `redirect_field_name`, you will most likely need to customize your login template as well, since the template context variable which stores the redirect path will use the value of `redirect_field_name` as it's key rather than `"next"` (the default).

New in Django 1.3: Please, see the release notes

`login_required()` also takes an optional `login_url` parameter. Example:

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

@login_required(login_url='/accounts/login/')
def my_view(request):
    ...
```

Note that if you don't specify the `login_url` parameter, you'll need to map the appropriate Django view to `settings.LOGIN_URL`. For example, using the defaults, add the following line to your URLconf:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login'),
```

`views.login(`*`request`*`[,` *`template_name`*`,` *`redirect_field_name`*`,` *`authentication_form`*`])`

Here's what `django.contrib.auth.views.login` does:

- If called via `GET`, it displays a login form that POSTs to the same URL. More on this in a bit.
- If called via `POST`, it tries to log the user in. If login is successful, the view redirects to the URL specified in `next`. If `next` isn't provided, it redirects to `settings.LOGIN_REDIRECT_URL` (which defaults to `/accounts/profile/`). If login isn't successful, it redisplays the login form.

It's your responsibility to provide the login form in a template called `registration/login.html` by default. This template gets passed four template context variables:

- `form`: A `Form` object representing the login form. See the forms documentation for more on `Form` objects.
- `next`: The URL to redirect to after successful login. This may contain a query string, too.
- `site`: The current `Site`, according to the `SITE_ID` setting. If you don't have the site framework installed, this will be set to an instance of `RequestSite`, which derives the site name and domain from the current `HttpRequest`.
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of `request.META['SERVER_NAME']`. For more on sites, see The "sites" framework.

If you'd prefer not to call the template `registration/login.html`, you can pass the `template_name` parameter via the extra arguments to the view in your URLconf. For example, this URLconf line would use `myapp/login.html` instead:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login', {'template_name': 'myapp/login.html'}
```

You can also specify the name of the `GET` field which contains the URL to redirect to after login by passing `redirect_field_name` to the view. By default, the field is called `next`.

Here's a sample `registration/login.html` template you can use as a starting point. It assumes you have a `base.html` template that defines a `content` block:

```
{% extends "base.html" %}
{% load url from future %}

{% block content %}

{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

<form method="post" action="{% url 'django.contrib.auth.views.login' %}">
{% csrf_token %}
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>

{% endblock %}
```

New in Django 1.2: Please, see the release notes

If you are using alternate authentication (see Other authentication sources) you can pass a custom authentication form to the login view via the `authentication_form` parameter. This form must accept a `request` keyword argument in its `__init__` method, and provide a `get_user` method which returns the authenticated user object (this method is only ever called after successful form validation).

Other built-in views

In addition to the `login()` view, the authentication system includes a few other useful built-in views located in `django.contrib.auth.views`:

logout(*request***[, *next_page*, *template_name*, *redirect_field_name*])**

Logs a user out.

Optional arguments:

- `next_page`: The URL to redirect to after logout.
- `template_name`: The full name of a template to display after logging the user out. This will default to `registration/logged_out.html` if no argument is supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after log out. Overrides `next_page` if the given `GET` parameter is passed.

Template context:

- `title`: The string "Logged out", localized.

logout_then_login(*request***[, *login_url*])**

Logs a user out, then redirects to the login page.

Optional arguments:

- `login_url`: The URL of the login page to redirect to. This will default to `settings.LOGIN_URL` if not supplied.

password_change(*request***[, *template_name*, *post_change_redirect*, *password_change_form*])**

Allows a user to change their password.

Optional arguments:

- `template_name`: The full name of a template to use for displaying the password change form. This will default to `registration/password_change_form.html` if not supplied.
- `post_change_redirect`: The URL to redirect to after a successful password change.

  New in Django 1.2: Please, see the release notes `password_change_form`: A custom "change password" form which must accept a `user` keyword argument. The form is responsible for actually changing the user's password.

Template context:

- `form`: The password change form.

password_change_done(*request***[, *template_name*])**

The page shown after a user has changed their password.

Optional arguments:

- `template_name`: The full name of a template to use. This will default to `registration/password_change_done.html` if not supplied.

password_reset(*request***[, *is_admin_site*, *template_name*, *email_template_name*, *password_reset_form*, *token_generator*, *post_reset_redirect*, *from_email*])**

Allows a user to reset their password by generating a one-time use link that can be used to reset the password, and sending that link to the user's registered e-mail address.

Changed in Django 1.3: The `from_email` argument was added.

Optional arguments:

- `template_name`: The full name of a template to use for displaying the password reset form. This will default to `registration/password_reset_form.html` if not supplied.
- `email_template_name`: The full name of a template to use for generating the e-mail with the new password. This will default to `registration/password_reset_email.html` if not supplied.
- `password_reset_form`: Form that will be used to set the password. Defaults to `PasswordResetForm`.
- `token_generator`: Instance of the class to check the password. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `post_reset_redirect`: The URL to redirect to after a successful password change.
- `from_email`: A valid e-mail address. By default Django uses the `DEFAULT_FROM_EMAIL`.

Template context:

- `form`: The form for resetting the user's password.

password_reset_done(*request***[, *template_name*])**

The page shown after a user has been emailed a link to reset their password. This view is called by default if the `password_reset()` view doesn't have an explicit `post_reset_redirect` URL set.

Optional arguments:

- `template_name`: The full name of a template to use. This will default to `registration/password_reset_done.html` if not supplied.

`password_reset_confirm(`*request*`[,` *uidb36*`,` *token*`,` *template_name*`,` *token_generator*`,` *set_password_form*`,` *post_reset_redirect*`])`**

Presents a form for entering a new password.

Optional arguments:

- `uidb36`: The user's id encoded in base 36. This will default to `None`.
- `token`: Token to check that the password is valid. This will default to `None`.
- `template_name`: The full name of a template to display the confirm password view. Default value is `registration/password_reset_confirm.html`.
- `token_generator`: Instance of the class to check the password. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `set_password_form`: Form that will be used to set the password. This will default to `SetPasswordForm`.
- `post_reset_redirect`: URL to redirect after the password reset done. This will default to `None`.

`password_reset_complete(`*request*`[,` *template_name*`])`**

Presents a view which informs the user that the password has been successfully changed.

Optional arguments:

- `template_name`: The full name of a template to display the view. This will default to `registration/password_reset_complete.html`.

Helper functions

`redirect_to_login(`*next*`[,` *login_url*`,` *redirect_field_name*`])`**

Redirects to the login page, and then back to another URL after a successful login.

Required arguments:

- `next`: The URL to redirect to after a successful login.

Optional arguments:

- `login_url`: The URL of the login page to redirect to. This will default to `settings.LOGIN_URL` if not supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after log out. Overrides `next` if the given `GET` parameter is passed.

Built-in forms

If you don't want to use the built-in views, but want the convenience of not having to write forms for this functionality, the authentication system provides several built-in forms located in `django.contrib.auth.forms`:

class **`AdminPasswordChangeForm`**

A form used in the admin interface to change a user's password.

class **`AuthenticationForm`**

A form for logging a user in.

class **`PasswordChangeForm`**

A form for allowing a user to change their password.

class **`PasswordResetForm`**

A form for generating and e-mailing a one-time use link to reset a user's password.

class **`SetPasswordForm`**

A form that lets a user change his/her password without entering the old password.

class **UserChangeForm**

A form used in the admin interface to change a user's information and permissions.

class **UserCreationForm**

A form for creating a new user.

Limiting access to logged-in users that pass a test

To limit access based on certain permissions or some other test, you'd do essentially the same thing as described in the previous section.

The simple 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`:

```
def my_view(request):
    if not request.user.has_perm('polls.can_vote'):
        return HttpResponse("You can't vote in this poll.")
    # ...
```

user_passes_test()

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

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

@user_passes_test(lambda u: u.has_perm('polls.can_vote'))
def my_view(request):
    ...
```

We're using this particular test as a relatively simple example. However, if you just want to test whether a permission is available to a user, you can use the `permission_required()` decorator, described later in this document.

`user_passes_test()` takes a 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 not anonymous.

`user_passes_test()` takes an optional `login_url` argument, which lets you specify the URL for your login page (`settings.LOGIN_URL` by default).

For example:

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

@user_passes_test(lambda u: u.has_perm('polls.can_vote'), login_url='/login/')
def my_view(request):
    ...
```

The permission_required decorator

permission_required()

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

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

@permission_required('polls.can_vote')
def my_view(request):
    ...
```

As for the `User.has_perm()` method, permission names take the form `"<app label>.<permission codename>"` (i.e. `polls.can_vote` for a permission on a model in the `polls` application).

Note that `permission_required()` also takes an optional `login_url` parameter. Example:

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

@permission_required('polls.can_vote', login_url='/loginpage/')
def my_view(request):
    ...
```

As in the `login_required()` decorator, `login_url` defaults to `settings.LOGIN_URL`.

Limiting access to generic views

Controlling access to a class-based generic view is done by decorating the `View.dispatch` method on the class. See Decorating the class for the details.

Function-based generic views

To limit access to a function-based generic view, write a thin wrapper around the view, and point your URLconf to your wrapper instead of the generic view itself. For example:

```
from django.views.generic.date_based import object_detail

@login_required
def limited_object_detail(*args, **kwargs):
    return object_detail(*args, **kwargs)
```

Permissions

Django comes with a simple permissions system. It provides a way to assign permissions to specific users and groups of users.

It's used by the Django admin site, but you're welcome to use it 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 it's not currently possible to 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." The latter functionality is something Django developers are currently discussing.

Default permissions

When `django.contrib.auth` is listed in your `INSTALLED_APPS` setting, it will ensure that three default permissions -- add, change and delete -- are created for each Django model defined in one of your installed applications.

These permissions will be created when you run `manage.py syncdb`; the first time you run `syncdb` after adding `django.contrib.auth` to `INSTALLED_APPS`, the default permissions will be created for all previously-installed models, as well as for any new models being installed at that time. Afterward, it will create default permissions for new models each time you run `manage.py syncdb`.

Assuming you have an application with an `app_label` `foo` and a model named `Bar`, to test for basic permissions you should use:

- add: `user.has_perm('foo.add_bar')`
- change: `user.has_perm('foo.change_bar')`
- delete: `user.has_perm('foo.delete_bar')`

Custom permissions

To create custom permissions for a given model object, use the `permissions` model Meta attribute.

This example Task model creates three custom permissions, i.e., actions users can or cannot do with Task instances, specific to your application:

```
class Task(models.Model):
    ...
    class Meta:
        permissions = (
            ("view_task", "Can see available tasks"),
            ("change_task_status", "Can change the status of tasks"),
            ("close_task", "Can remove a task by setting its status as closed"),
        )
```

The only thing this does is create those extra permissions when you run `manage.py syncdb`. Your code is in charge of checking the value of these permissions when an user is trying to access the functionality provided by the application (viewing tasks, changing the status of tasks, closing tasks.) Continuing the above example, the following checks if a user may view tasks:

```
user.has_perm('app.view_task')
```

API reference

class **Permission**

Just like users, permissions are implemented in a Django model that lives in django/contrib/auth/models.py.

Fields

`Permission` objects have the following fields:

`Permission.name`

Required. 50 characters or fewer. Example: `'Can vote'`.

`Permission.content_type`

Required. A reference to the `django_content_type` database table, which contains a record for each installed Django model.

`Permission.codename`

Required. 100 characters or fewer. Example: `'can_vote'`.

Methods

`Permission` objects have the standard data-access methods like any other Django model.

Authentication data in templates

The currently logged-in user and his/her permissions are made available in the template context when you use `RequestContext`.

Technicality

Technically, these variables are only made available in the template context if you use `RequestContext` *and* your `TEMPLATE_CONTEXT_PROCESSORS` setting contains `"django.contrib.auth.context_processors.auth"`, which is default. For more, see the RequestContext docs.

Users

When rendering a template `RequestContext`, the currently logged-in 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 template context variable is not available if a `RequestContext` is not being used.

Permissions

The currently logged-in user's permissions are stored in the template variable `{{ perms }}`. This is an instance of `django.contrib.auth.context_processors.PermWrapper`, which is a template-friendly proxy of permissions.

Changed in Django 1.3: Prior to version 1.3, `PermWrapper` was located in `django.contrib.auth.context_processors`.

In the `{{ perms }}` object, single-attribute lookup is a proxy to `User.has_module_perms`. This example would display `True` if the logged-in user had any permissions in the `foo` app:

```
{{ perms.foo }}
```

Two-level-attribute lookup is a proxy to `User.has_perm`. This example would display `True` if the logged-in user had the permission `foo.can_vote`:

```
{{ perms.foo.can_vote }}
```

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

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.can_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.can_drive %}
        <p>You can drive!</p>
    {% endif %}
{% else %}
    <p>You don't have permission to do anything in the foo app.</p>
{% endif %}
```

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.

Beyond permissions, groups are 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 them access to a members-only portion of your site, or send them members-only e-mail messages.

Messages

Deprecated in Django 1.2:

Deprecated since version 1.2: This functionality will be removed in Django 1.4. You should use the messages framework for all new projects and begin to update your existing code immediately.

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 after successful actions. For example, `"The poll Foo was created successfully."` is a message.

The API is simple:

`models.User.message_set.create`(*message*)

To create a new message, use `user_obj.message_set.create(message='message_text')`.

To retrieve/delete messages, use `user_obj.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_to_response("playlists/create.html",
        context_instance=RequestContext(request))
```

When you use `RequestContext`, the currently logged-in user and his/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 %}
```

Changed in Django 1.2: The `messages` template variable uses a backwards compatible method in the messages framework to retrieve messages from both the user `Message` model and from the new framework. Unlike in previous revisions, the messages will not be erased unless they are actually displayed.

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

Other authentication sources

The authentication that comes with Django is good enough for most common cases, but you may have the need to hook into another authentication source -- that is, another source of usernames and passwords or authentication methods.

For example, your company may already have an LDAP setup that stores a username and password for every employee. It'd be a hassle for both the network administrator and the users themselves if users had separate accounts in LDAP and the Django-based applications.

So, to handle situations like this, the Django authentication system lets you plug in other authentication sources. You can override Django's default database-based scheme, or you can use the default system in tandem with other systems.

See the authentication backend reference for information on the authentication backends included with Django.

Specifying authentication backends

Behind the scenes, Django maintains a list of "authentication backends" that it checks for authentication.

When somebody calls `django.contrib.auth.authenticate()` -- as described in How to log a user in above -- Django tries authenticating across all of its authentication backends. If the first authentication method fails, Django tries the second one, and so on, until all backends have been attempted.

The list of authentication backends to use is specified in the `AUTHENTICATION_BACKENDS` setting. This should be a tuple of Python path names that point to Python classes that know how to authenticate. These classes can be anywhere on your Python path.

By default, `AUTHENTICATION_BACKENDS` is set to:

```
('django.contrib.auth.backends.ModelBackend',)
```

That's the basic authentication scheme that checks the Django users database.

The order of `AUTHENTICATION_BACKENDS` matters, so if the same username and password is valid in multiple backends, Django will stop processing at the first positive match.

Note

Once a user has authenticated, Django stores which backend was used to authenticate the user in the user's session, and re-uses the same backend for the duration of that session whenever access to the currently authenticated user is needed. This effectively means that authentication sources are cached on a per-session basis, so if you change `AUTHENTICATION_BACKENDS`, you'll need to clear out session data if you need to force users to re-authenticate using different methods. A simple way to do that is simply to execute `Session.objects.all().delete()`.

Writing an authentication backend

An authentication backend is a class that implements two methods: `get_user(user_id)` and `authenticate(**credentials)`.

The `get_user` method takes a `user_id` -- which could be a username, database ID or whatever -- and returns a `User` object.

The `authenticate` method takes credentials as keyword arguments. Most of the time, it'll just look like this:

```
class MyBackend:
    def authenticate(self, username=None, password=None):
        # Check the username/password and return a User.
```

But it could also authenticate a token, like so:

```
class MyBackend:
    def authenticate(self, token=None):
        # Check the token and return a User.
```

Either way, `authenticate` should check the credentials it gets, and it should return a `User` object that matches those credentials, if the credentials are valid. If they're not valid, it should return `None`.

The Django admin system is tightly coupled to the Django `User` object described at the beginning of this document. For now, the best way to deal with this is to create a Django `User` object for each user that exists for your backend (e.g., in your LDAP directory, your external SQL database, etc.) You can either write a script to do this in advance, or your `authenticate` method can do it the first time a user logs in.

Here's an example backend that authenticates against a username and password variable defined in your `settings.py` file and creates a Django `User` object the first time a user authenticates:

```
from django.conf import settings
from django.contrib.auth.models import User, check_password

class SettingsBackend:
    """
    Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWORD.

    Use the login name, and a hash of the password. For example:

    ADMIN_LOGIN = 'admin'
    ADMIN_PASSWORD = 'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'
    """

    supports_object_permissions = False
    supports_anonymous_user = False
    supports_inactive_user = False

    def authenticate(self, username=None, password=None):
        login_valid = (settings.ADMIN_LOGIN == username)
        pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # Create a new user. Note that we can set password
                # to anything, because it won't be checked; the password
```

```
                # from settings.py will.
                user = User(username=username, password='get from settings.py')
                user.is_staff = True
                user.is_superuser = True
                user.save()
            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

Handling authorization in custom backends

Custom auth backends can provide their own permissions.

The user model will delegate permission lookup functions (`get_group_permissions()`, `get_all_permissions()`, `has_perm()`, and `has_module_perms()`) to any authentication backend that implements these functions.

The permissions given to the user will be the superset of all permissions returned by all backends. That is, Django grants a permission to a user that any one backend grants.

The simple backend above could implement permissions for the magic admin fairly simply:

```
class SettingsBackend:

    # ...

    def has_perm(self, user_obj, perm):
        if user_obj.username == settings.ADMIN_LOGIN:
            return True
        else:
            return False
```

This gives full permissions to the user granted access in the above example. Notice that the backend auth functions all take the user object as an argument, and they also accept the same arguments given to the associated `django.contrib.auth.models.User` functions.

A full authorization implementation can be found in django/contrib/auth/backends.py, which is the default backend and queries the `auth_permission` table most of the time.

Authorization for anonymous users

Changed in Django 1.2: Please, see the release notes

An anonymous user is one that is not authenticated i.e. they have provided no valid authentication details. However, that does not necessarily mean they are not authorized to do anything. At the most basic level, most Web sites authorize anonymous users to browse most of the site, and many allow anonymous posting of comments etc.

Django's permission framework does not have a place to store permissions for anonymous users. However, it has a foundation that allows custom authentication backends to specify authorization for anonymous users. This is especially useful for the authors of re-usable apps, who can delegate all questions of authorization to the auth backend, rather than needing settings, for example, to control anonymous access.

To enable this in your own backend, you must set the class attribute `supports_anonymous_user` to `True`. (This precaution is to maintain compatibility with backends that assume that all user objects are actual instances of the `django.contrib.auth.models.User` class). With this in place, `django.contrib.auth.models.AnonymousUser` will delegate all the relevant permission methods to the authentication backends.

A nonexistent `supports_anonymous_user` attribute will raise a hidden `PendingDeprecationWarning` if used in Django 1.2. In Django 1.3, this warning will be upgraded to a `DeprecationWarning`, which will be displayed loudly. Additionally `supports_anonymous_user` will be set to `False`. Django 1.4 will assume that every backend supports anonymous users being passed to the authorization methods.

Authorization for inactive users

New in Django 1.3: Please, see the release notes

An inactive user is a one that is authenticated but has its attribute `is_active` set to `False`. However this does not mean they are not authorized to do anything. For example they are allowed to activate their account.

The support for anonymous users in the permission system allows for anonymous users to have permissions to do something while inactive authenticated users do not.

To enable this on your own backend, you must set the class attribute `supports_inactive_user` to `True`.

A nonexisting `supports_inactive_user` attribute will raise a `PendingDeprecationWarning` if used in Django 1.3. In Django 1.4, this warning will be updated to a `DeprecationWarning` which will be displayed loudly. Additionally `supports_inactive_user` will be set to `False`. Django 1.5 will assume that every backend

supports inactive users being passed to the authorization methods.

Handling object permissions

Django's permission framework has a foundation for object permissions, though there is no implementation for it in the core. That means that checking for object permissions will always return `False` or an empty list (depending on the check performed).

To enable object permissions in your own authentication backend you'll just have to allow passing an `obj` parameter to the permission methods and set the `supports_object_permissions` class attribute to `True`.

A nonexistent `supports_object_permissions` will raise a hidden `PendingDeprecationWarning` if used in Django 1.2. In Django 1.3, this warning will be upgraded to a `DeprecationWarning`, which will be displayed loudly. Additionally `supports_object_permissions` will be set to `False`. Django 1.4 will assume that every backend supports object permissions and won't check for the existence of `supports_object_permissions`, which means not supporting `obj` as a parameter will raise a `TypeError`.