Using the Django authentication system

This document explains the usage of Django's authentication system in its default configuration. This configuration has evolved to serve the most common project needs, handling a reasonably wide range of tasks, and has a careful implementation of passwords and permissions. For projects where authentication needs differ from the default, Django supports extensive extension and customization of authentication.

Django authentication provides both authentication and authorization together and is generally referred to as the authentication system, as these features are somewhat coupled.

`User` objects

`User` objects are the core of the authentication system. They typically represent the people interacting with your site and are used to enable things like restricting access, registering user profiles, associating content with creators etc. Only one class of user exists in Django's authentication framework, i.e., `'superusers'` or admin `'staff'` users are just user objects with special attributes set, not different classes of user objects.

The primary attributes of the default user are:

- `username`
- `password`
- `email`
- `first_name`
- `last_name`

See the `full API documentation` for full reference, the documentation that follows is more task oriented.

Creating users

The most direct way to create users is to use the included `create_user()` helper function:

```
>>> 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.last_name = "Lennon"
>>> user.save()
```

If you have the Django admin installed, you can also create users interactively.

Creating superusers

Create superusers using the `createsuperuser` command:

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

```
...\> py 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 `--email` options, it will prompt you for those values.

Changing passwords

Django does not store raw (clear text) passwords on the user model, but only a hash (see documentation of how passwords are managed for full details). Because of this, do not attempt to manipulate the password attribute of the user directly. This is why a helper function is used when creating a user.

To change a user's password, you have several options:

`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 system user.

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

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

If you have the Django admin installed, you can also change user's passwords on the authentication system's admin pages.

Django also provides views and forms that may be used to allow users to change their own passwords.

Changing a user's password will log out all their sessions. See Session invalidation on password change for details.

Authenticating users

```
authenticate(request=None, **credentials)
```
[source]

```
aauthenticate(request=None, **credentials)
```

Asynchronous version: `aauthenticate()`

Use `authenticate()` to verify a set of credentials. It takes credentials as keyword arguments, `username` and `password` for the default case, checks them against each authentication backend, and returns a `User` object if the credentials are valid for a backend. If the credentials aren't valid for any backend or if a backend raises `PermissionDenied`, it returns `None`. For example:

```
from django.contrib.auth import authenticate

user = authenticate(username="john", password="secret")
if user is not None:
    # A backend authenticated the credentials
    ...
else:
    # No backend authenticated the credentials
    ...
```

`request` is an optional `HttpRequest` which is passed on the `authenticate()` method of the authentication backends.

Note: This is a low level way to authenticate a set of credentials; for example, it's used by the `RemoteUserMiddleware`. Unless you are writing your own authentication system, you probably won't use this. Rather if you're looking for a way to login a user, use the `LoginView`.

Changed in Django 5.0:
`aauthenticate()` function was added.

Permissions and Authorization

Django comes with a built-in 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 objects is limited to users with the "view" or "change" permission for that type of object.
- 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 can be set not only per type of object, but also per specific object instance. By using the `has_view_permission()`, `has_add_permission()`, `has_change_permission()` and `has_delete_permission()` methods provided by the `ModelAdmin` class, it is possible to customize permissions for different object instances of the same type.

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

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

Default permissions

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

These permissions will be created when you run `manage.py migrate`; the first time you run `migrate` 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 migrate` (the function that creates permissions is connected to the `post_migrate` signal).

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')`
- view: `user.has_perm('foo.view_bar')`

The `Permission` model is rarely accessed directly.

Groups

`django.contrib.auth.models.Group` models 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 email messages.

Programmatically creating permissions

While custom permissions can be defined within a model's `Meta` class, you can also create permissions directly. For example, you can create the `can_publish` permission for a `BlogPost` model in `myapp`:

```
from myapp.models import BlogPost
from django.contrib.auth.models import Permission
from django.contrib.contenttypes.models import ContentType

content_type = ContentType.objects.get_for_model(BlogPost)
permission = Permission.objects.create(
    codename="can_publish",
    name="Can Publish Posts",
    content_type=content_type,
)
```

The permission can then be assigned to a `User` via its `user_permissions` attribute or to a `Group` via its `permissions` attribute.

Proxy models need their own content type: If you want to create permissions for a proxy model, pass `for_concrete_model=False` to `ContentTypeManager.get_for_model()` to get the appropriate `ContentType`:

```
content_type = ContentType.objects.get_for_model(
    BlogPostProxy, for_concrete_model=False
)
```

Permission caching

The `ModelBackend` caches permissions on the user object after the first time they need to be fetched for a permissions check. This is typically fine for the request-response cycle since permissions aren't typically checked immediately after they are added (in the admin, for example). If you are adding permissions and checking them immediately afterward, in a test or view for example, the easiest solution is to re-fetch the user from the database. For example:

```python
from django.contrib.auth.models import Permission, User
from django.contrib.contenttypes.models import ContentType
from django.shortcuts import get_object_or_404

from myapp.models import BlogPost


def user_gains_perms(request, user_id):
    user = get_object_or_404(User, pk=user_id)
    # any permission check will cache the current set of permissions
    user.has_perm("myapp.change_blogpost")

    content_type = ContentType.objects.get_for_model(BlogPost)
    permission = Permission.objects.get(
        codename="change_blogpost",
        content_type=content_type,
    )
    user.user_permissions.add(permission)

    # Checking the cached permission set
    user.has_perm("myapp.change_blogpost")  # False

    # Request new instance of User
    # Be aware that user.refresh_from_db() won't clear the cache.
    user = get_object_or_404(User, pk=user_id)

    # Permission cache is repopulated from the database
    user.has_perm("myapp.change_blogpost")  # True

    ...
```

Proxy models

Proxy models work exactly the same way as concrete models. Permissions are created using the own content type of the proxy model. Proxy models don't inherit the permissions of the concrete model they subclass:

```python
class Person(models.Model):
    class Meta:
        permissions = [("can_eat_pizzas", "Can eat pizzas")]


class Student(Person):
    class Meta:
        proxy = True
        permissions = [("can_deliver_pizzas", "Can deliver pizzas")]
```

```
>>> # Fetch the content type for the proxy model.
>>> content_type = ContentType.objects.get_for_model(Student, for_concrete_model=False)
>>> student_permissions = Permission.objects.filter(content_type=content_type)
>>> [p.codename for p in student_permissions]
['add_student', 'change_student', 'delete_student', 'view_student',
'can_deliver_pizzas']
>>> for permission in student_permissions:
...     user.user_permissions.add(permission)
...
>>> user.has_perm("app.add_person")
False
>>> user.has_perm("app.can_eat_pizzas")
False
>>> user.has_perms(("app.add_student", "app.can_deliver_pizzas"))
True
```

Authentication in web requests

Django uses sessions and middleware to hook the authentication system into `request objects`.

These provide a `request.user` attribute and a `request.auser` async method on every request which represents the current user. If the current user has not logged in, this attribute will be set to an instance of `AnonymousUser`, otherwise it will be an instance of `User`.

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.
    ...
```

Or in an asynchronous view:

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

Changed in Django 5.0:

The `HttpRequest.auser()` method was added.

How to log a user in

If you have an authenticated user you want to attach to the current session - this is done with a `login()` function.

`login(request, user, backend=None)` [source]

`alogin(request, user, backend=None)`

Asynchronous version: `alogin()`

To log a user in, from 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.

Note that any data set during the anonymous session is retained in the session after a user logs in.

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(request, username=username, password=password)
    if user is not None:
        login(request, user)
        # Redirect to a success page.
        ...
    else:
        # Return an 'invalid login' error message.
        ...
```

Changed in Django 5.0:

`alogin()` function was added.

Selecting the authentication backend

When a user logs in, the user's ID and the backend that was used for authentication are saved in the user's session. This allows the same authentication backend to fetch the user's details on a future request. The authentication backend to save in the session is selected as follows:

1. Use the value of the optional `backend` argument, if provided.
2. Use the value of the `user.backend` attribute, if present. This allows pairing `authenticate()` and `login()`: `authenticate()` sets the `user.backend` attribute on the user object it returns.

3. Use the `backend` in `AUTHENTICATION_BACKENDS`, if there is only one.
4. Otherwise, raise an exception.

In cases 1 and 2, the value of the `backend` argument or the `user.backend` attribute should be a dotted import path string (like that found in `AUTHENTICATION_BACKENDS`), not the actual backend class.

How to log a user out

`logout(request)` [source]

`alogout(request)`

Asynchronous version: `alogout()`

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()`.

Changed in Django 5.0:
`alogout()` function was added.

Limiting access to logged-in users

The raw way

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

```
from django.conf import settings
from django.shortcuts import redirect


def my_view(request):
    if not request.user.is_authenticated:
        return redirect(f"{settings.LOGIN_URL}?next={request.path}")
    # ...
```

...or display an error message:

```
from django.shortcuts import render


def my_view(request):
    if not request.user.is_authenticated:
        return render(request, "myapp/login_error.html")
    # ...
```

The `login_required` decorator

`login_required(redirect_field_name='next', login_url=None)` [source]

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 its key rather than `"next"` (the default).

`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 ensure that the `settings.LOGIN_URL` and your login view are properly associated. For example, using the defaults, add the following lines to your URLconf:

```
from django.contrib.auth import views as auth_views

path("accounts/login/", auth_views.LoginView.as_view()),
```

The `settings.LOGIN_URL` also accepts view function names and named URL patterns. This allows you to freely remap your login view within your URLconf without having to update the setting.

Note: The `login_required` decorator does NOT check the `is_active` flag on a user, but the default `AUTHENTICATION_BACKENDS` reject inactive users.

See also: If you are writing custom views for Django's admin (or need the same authorization check that the built-in views use), you may find the `django.contrib.admin.views.decorators.staff_member_required()` decorator a useful alternative to `login_required()` .

Changed in Django 5.1:
Support for wrapping asynchronous view functions was added.

The `LoginRequiredMixin` mixin

When using class-based views, you can achieve the same behavior as with `login_required` by using the `LoginRequiredMixin` . This mixin should be at the leftmost position in the inheritance list.

`class LoginRequiredMixin` [source]

If a view is using this mixin, all requests by non-authenticated users will be redirected to the login page or shown an HTTP 403 Forbidden error, depending on the `raise_exception` parameter.

You can set any of the parameters of AccessMixin to customize the handling of unauthorized users:

```
from django.contrib.auth.mixins import LoginRequiredMixin


class MyView(LoginRequiredMixin, View):
    login_url = "/login/"
    redirect_field_name = "redirect_to"
```

Note: Just as the `login_required` decorator, this mixin does NOT check the `is_active` flag on a user, but the default AUTHENTICATION_BACKENDS reject inactive users.

The `login_not_required` decorator

New in Django 5.1.

When LoginRequiredMiddleware is installed, all views require authentication by default. Some views, such as the login view, may need to disable this behavior.

`login_not_required()` [source]

Allows unauthenticated requests to this view when LoginRequiredMiddleware is installed.

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.

You can run your test on request.user in the view directly. For example, this view checks to make sure the user has an email in the desired domain and if not, redirects to the login page:

```
from django.shortcuts import redirect


def my_view(request):
    if not request.user.email.endswith("@example.com"):
        return redirect("/login/?next=%s" % request.path)
    # ...
```

`user_passes_test(test_func, login_url=None, redirect_field_name='next')` [source]

As a shortcut, you can use the convenient `user_passes_test` decorator which performs a redirect when the callable returns `False`:

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


def email_check(user):
    return user.email.endswith("@example.com")


@user_passes_test(email_check)
def my_view(request): ...
```

`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 two optional arguments:

`login_url`

Lets you specify the URL that users who don't pass the test will be redirected to. It may be a login page and defaults to `settings.LOGIN_URL` if you don't specify one.

`redirect_field_name`

Same as for `login_required()`. Setting it to `None` removes it from the URL, which you may want to do if you are redirecting users that don't pass the test to a non-login page where there's no "next page".

For example:

```
@user_passes_test(email_check, login_url="/login/")
def my_view(request): ...
```

Changed in Django 5.1:
Support for wrapping asynchronous view functions and using asynchronous test callables was added.

`class UserPassesTestMixin` [source]

When using class-based views, you can use the `UserPassesTestMixin` to do this.

`test_func()` [source]

You have to override the `test_func()` method of the class to provide the test that is performed. Furthermore, you can set any of the parameters of `AccessMixin` to customize the handling of unauthorized users:

```
from django.contrib.auth.mixins import UserPassesTestMixin


class MyView(UserPassesTestMixin, View):
    def test_func(self):
        return self.request.user.email.endswith("@example.com")
```

`get_test_func()` [source]

You can also override the `get_test_func()` method to have the mixin use a differently named function for its checks (instead of `test_func()`).

Stacking `UserPassesTestMixin`**:** Due to the way `UserPassesTestMixin` is implemented, you cannot stack them in your inheritance list. The following does NOT work:

```
class TestMixin1(UserPassesTestMixin):
    def test_func(self):
        return self.request.user.email.endswith("@example.com")


class TestMixin2(UserPassesTestMixin):
    def test_func(self):
        return self.request.user.username.startswith("django")


class MyView(TestMixin1, TestMixin2, View): ...
```

If `TestMixin1` would call `super()` and take that result into account, `TestMixin1` wouldn't work standalone anymore.

The `permission_required` decorator

`permission_required(perm, login_url=None, raise_exception=False)` [source]

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:

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


@permission_required("polls.add_choice")
def my_view(request): ...
```

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

The decorator may also take an iterable of permissions, in which case the user must have all of the permissions in order to access the view.

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

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


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

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

If the `raise_exception` parameter is given, the decorator will raise `PermissionDenied`, prompting the 403 (HTTP Forbidden) view instead of redirecting to the login page.

If you want to use `raise_exception` but also give your users a chance to login first, you can add the `login_required()` decorator:

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


@login_required
@permission_required("polls.add_choice", raise_exception=True)
def my_view(request): ...
```

This also avoids a redirect loop when `LoginView`'s `redirect_authenticated_user=True` and the logged-in user doesn't have all of the required permissions.

Changed in Django 5.1:
Support for wrapping asynchronous view functions was added.

The `PermissionRequiredMixin` mixin

To apply permission checks to class-based views, you can use the `PermissionRequiredMixin`:

`class PermissionRequiredMixin` [source]

This mixin, just like the `permission_required` decorator, checks whether the user accessing a view has all given permissions. You should specify the permission (or an iterable of permissions) using the `permission_required` parameter:

```
from django.contrib.auth.mixins import PermissionRequiredMixin


class MyView(PermissionRequiredMixin, View):
    permission_required = "polls.add_choice"
    # Or multiple of permissions:
    permission_required = ["polls.view_choice", "polls.change_choice"]
```

You can set any of the parameters of `AccessMixin` to customize the handling of unauthorized users.

You may also override these methods:

`get_permission_required()` [source]

Returns an iterable of permission names used by the mixin. Defaults to the `permission_required` attribute, converted to a tuple if necessary.

`has_permission()` [source]

Returns a boolean denoting whether the current user has permission to execute the decorated view. By default, this returns the result of calling `has_perms()` with the list of permissions returned by `get_permission_required()`.

Redirecting unauthorized requests in class-based views

To ease the handling of access restrictions in class-based views, the `AccessMixin` can be used to configure the behavior of a view when access is denied. Authenticated users are denied access with an HTTP 403 Forbidden response. Anonymous users are redirected to the login page or shown an HTTP 403 Forbidden response, depending on the `raise_exception` attribute.

`class AccessMixin` [source]

`login_url`

Default return value for `get_login_url()`. Defaults to `None` in which case `get_login_url()` falls back to `settings.LOGIN_URL`.

`permission_denied_message`

Default return value for `get_permission_denied_message()`. Defaults to an empty string.

`redirect_field_name`

Default return value for `get_redirect_field_name()`. Defaults to `"next"`.

`raise_exception`

If this attribute is set to `True`, a `PermissionDenied` exception is raised when the conditions are not met. When `False` (the default), anonymous users are redirected to the login page.

`get_login_url()` [source]

Returns the URL that users who don't pass the test will be redirected to. Returns `login_url` if set, or `settings.LOGIN_URL` otherwise.

`get_permission_denied_message()` [source]

When `raise_exception` is `True`, this method can be used to control the error message passed to the error handler for display to the user. Returns the `permission_denied_message` attribute by default.

`get_redirect_field_name()` [source]

Returns the name of the query parameter that will contain the URL the user should be redirected to after a successful login. If you set this to `None`, a query parameter won't be added. Returns the `redirect_field_name` attribute by default.

`handle_no_permission()` [source]

Depending on the value of `raise_exception`, the method either raises a `PermissionDenied` exception or redirects the user to the `login_url`, optionally including the `redirect_field_name` if it is set.

Session invalidation on password change

If your `AUTH_USER_MODEL` inherits from `AbstractBaseUser` or implements its own `get_session_auth_hash()` method, authenticated sessions will include the hash returned by this function. In the `AbstractBaseUser` case, this is an HMAC of the password field. Django verifies that the hash in the session for each request matches the one that's computed during the request. This allows a user to log out all of their sessions by changing their password.

The default password change views included with Django, `PasswordChangeView` and the `user_change_password` view in the `django.contrib.auth` admin, update the session with the new password hash so that a user changing their own password won't log themselves out. If you have a custom password change view and wish to have similar behavior, use the `update_session_auth_hash()` function.

`update_session_auth_hash(request, user)` [source]

`aupdate_session_auth_hash(request, user)`

Asynchronous version: `aupdate_session_auth_hash()`

This function takes the current request and the updated user object from which the new session hash will be derived and updates the session hash appropriately. It also rotates the session key so that a stolen session cookie will be invalidated.

Example usage:

```
from django.contrib.auth import update_session_auth_hash


def password_change(request):
    if request.method == "POST":
        form = PasswordChangeForm(user=request.user, data=request.POST)
        if form.is_valid():
            form.save()
            update_session_auth_hash(request, form.user)
    else:
        ...
```

Changed in Django 5.0:
`aupdate_session_auth_hash()` function was added.

Note: Since `get_session_auth_hash()` is based on `SECRET_KEY`, secret key values must be rotated to avoid invalidating existing sessions when updating your site to use a new secret. See `SECRET_KEY_FALLBACKS` for details.

Authentication Views

Django provides several views that you can use for handling login, logout, and password management. These make use of the stock auth forms but you can pass in your own forms as well.

Django provides no default template for the authentication views. You should create your own templates for the views you want to use. The template context is documented in each view, see All authentication views.

Using the views

There are different methods to implement these views in your project. The easiest way is to include the provided URLconf in `django.contrib.auth.urls` in your own URLconf, for example:

```
urlpatterns = [
    path("accounts/", include("django.contrib.auth.urls")),
]
```

This will include the following URL patterns:

```
accounts/login/ [name='login']
accounts/logout/ [name='logout']
accounts/password_change/ [name='password_change']
accounts/password_change/done/ [name='password_change_done']
accounts/password_reset/ [name='password_reset']
accounts/password_reset/done/ [name='password_reset_done']
accounts/reset/<uidb64>/<token>/ [name='password_reset_confirm']
accounts/reset/done/ [name='password_reset_complete']
```

The views provide a URL name for easier reference. See the URL documentation for details on using named URL patterns.

If you want more control over your URLs, you can reference a specific view in your URLconf:

```
from django.contrib.auth import views as auth_views

urlpatterns = [
    path("change-password/", auth_views.PasswordChangeView.as_view()),
]
```

The views have optional arguments you can use to alter the behavior of the view. For example, if you want to change the template name a view uses, you can provide the `template_name` argument. A way to do this is to provide keyword arguments in the URLconf, these will be passed on to the view. For example:

```
urlpatterns = [
    path(
        "change-password/",
        auth_views.PasswordChangeView.as_view(template_name="change-password.html"),
    ),
]
```

All views are class-based, which allows you to easily customize them by subclassing.

All authentication views

This is a list with all the views `django.contrib.auth` provides. For implementation details see Using the views.

`class LoginView` [source]

URL name: `login`

See the URL documentation for details on using named URL patterns.

Methods and Attributes

`template_name`

The name of a template to display for the view used to log the user in. Defaults to `registration/login.html` .

`next_page`

The URL to redirect to after login. Defaults to `LOGIN_REDIRECT_URL`.

```
redirect_field_name
```

The name of a `GET` field containing the URL to redirect to after login. Defaults to `next`. Overrides the `get_default_redirect_url()` URL if the given `GET` parameter is passed.

```
authentication_form
```

A callable (typically a form class) to use for authentication. Defaults to `AuthenticationForm`.

```
extra_context
```

A dictionary of context data that will be added to the default context data passed to the template.

```
redirect_authenticated_user
```

A boolean that controls whether or not authenticated users accessing the login page will be redirected as if they had just successfully logged in. Defaults to `False`.

> **Warning:** If you enable `redirect_authenticated_user`, other websites will be able to determine if their visitors are authenticated on your site by requesting redirect URLs to image files on your website. To avoid this "social media fingerprinting" information leakage, host all images and your favicon on a separate domain.
>
> Enabling `redirect_authenticated_user` can also result in a redirect loop when using the `permission_required()` decorator unless the `raise_exception` parameter is used.

```
success_url_allowed_hosts
```

A `set` of hosts, in addition to `request.get_host()`, that are safe for redirecting after login. Defaults to an empty `set`.

```
get_default_redirect_url()
```
[source]

Returns the URL to redirect to after login. The default implementation resolves and returns `next_page` if set, or `LOGIN_REDIRECT_URL` otherwise.

Here's what `LoginView` 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` with user submitted credentials, 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 html for the login template , called `registration/login.html` by default. This template gets passed four template context variables:

- `form`: A `Form` object representing the `AuthenticationForm`.
- `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 `as_view` method in your URLconf. For example, this URLconf line would use `myapp/login.html` instead:

```
path("accounts/login/", auth_views.LoginView.as_view(template_name="myapp/login.html")),
```

You can also specify the name of the `GET` field which contains the URL to redirect to after login using `redirect_field_name`. 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" %}

{% block content %}

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

{% if next %}
    {% if user.is_authenticated %}
    <p>Your account doesn't have access to this page. To proceed,
    please login with an account that has access.</p>
    {% else %}
    <p>Please login to see this page.</p>
    {% endif %}
{% endif %}

<form method="post" action="{% url '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>

{# Assumes you set up the password_reset view in your URLconf #}
<p><a href="{% url 'password_reset' %}">Lost password?</a></p>

{% endblock %}
```

If you have customized authentication (see Customizing Authentication) you can use a custom authentication form by setting the `authentication_form` attribute. 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).

`class LogoutView` [source]

Logs a user out on `POST` requests.

URL name: `logout`

Attributes:

`next_page`

The URL to redirect to after logout. Defaults to `LOGOUT_REDIRECT_URL`.

`template_name`

The full name of a template to display after logging the user out. Defaults to `registration/logged_out.html`.

`redirect_field_name`

The name of a `GET` field containing the URL to redirect to after log out. Defaults to `'next'`. Overrides the `next_page` URL if the given `GET` parameter is passed.

`extra_context`

A dictionary of context data that will be added to the default context data passed to the template.

`success_url_allowed_hosts`

A `set` of hosts, in addition to `request.get_host()`, that are safe for redirecting after logout. Defaults to an empty `set`.

Template context:

- `title` : The string “Logged out”, localized.
- `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.

`logout_then_login(request, login_url=None)` [source]

Logs a user out on `POST` requests, then redirects to the login page.

URL name: No default URL provided

Optional arguments:

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

`class PasswordChangeView` [source]

URL name: `password_change`

Allows a user to change their password.

Attributes:

`template_name`

The full name of a template to use for displaying the password change form. Defaults to `registration/password_change_form.html` if not supplied.

`success_url`

The URL to redirect to after a successful password change. Defaults to `'password_change_done'` .

`form_class`

A custom "change password" form which must accept a `user` keyword argument. The form is responsible for actually changing the user's password. Defaults to `PasswordChangeForm`.

`extra_context`

A dictionary of context data that will be added to the default context data passed to the template.

Template context:

- `form` : The password change form (see `form_class` above).

`class PasswordChangeDoneView` [source]

URL name: `password_change_done`

The page shown after a user has changed their password.

Attributes:

`template_name`

The full name of a template to use. Defaults to `registration/password_change_done.html` if not supplied.

`extra_context`

A dictionary of context data that will be added to the default context data passed to the template.

`class PasswordResetView` [source]

URL name: `password_reset`

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 email address.

This view will send an email if the following conditions are met:

- The email address provided exists in the system.
- The requested user is active (`User.is_active` is `True`).
- The requested user has a usable password. Users flagged with an unusable password (see `set_unusable_password()`) aren't allowed to request a password reset to prevent misuse when using an external authentication source like LDAP.

If any of these conditions are *not* met, no email will be sent, but the user won't receive any error message either. This prevents information leaking to potential attackers. If you want to provide an error message in this case, you can subclass `PasswordResetForm` and use the `form_class` attribute.

Note: Be aware that sending an email costs extra time, hence you may be vulnerable to an email address enumeration timing attack due to a difference between the duration of a reset request for an existing email address and the duration of a reset request for a nonexistent email address. To reduce the overhead, you can use a 3rd party package that allows to send emails asynchronously, e.g. django-mailer.

Attributes:

`template_name`

The full name of a template to use for displaying the password reset form. Defaults to `registration/password_reset_form.html` if not supplied.

`form_class`

Form that will be used to get the email of the user to reset the password for. Defaults to `PasswordResetForm`.

`email_template_name`

The full name of a template to use for generating the email with the reset password link. Defaults to `registration/password_reset_email.html` if not supplied.

`subject_template_name`

The full name of a template to use for the subject of the email with the reset password link. Defaults to `registration/password_reset_subject.txt` if not supplied.

`token_generator`

Instance of the class to check the one time link. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.

`success_url`

The URL to redirect to after a successful password reset request. Defaults to `'password_reset_done'`.

`from_email`

A valid email address. By default Django uses the `DEFAULT_FROM_EMAIL`.

`extra_context`

A dictionary of context data that will be added to the default context data passed to the template.

`html_email_template_name`

The full name of a template to use for generating a *text/html* multipart email with the password reset link. By default, HTML email is not sent.

`extra_email_context`

A dictionary of context data that will be available in the email template. It can be used to override default template context values listed below e.g. `domain`.

Template context:

- `form`: The form (see `form_class` above) for resetting the user's password.

Email template context:

- `email`: An alias for `user.email`
- `user`: The current User, according to the `email` form field. Only active users are able to reset their passwords (`User.is_active is True`).
- `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.
- `domain`: An alias for `site.domain`. If you don't have the site framework installed, this will be set to the value of `request.get_host()`.
- `protocol`: http or https
- `uid`: The user's primary key encoded in base 64.

- `token` : Token to check that the reset link is valid.

Sample `registration/password_reset_email.html` (email body template):

```
Someone asked for password reset for email {{ email }}. Follow the link below:
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token %}
```

The same template context is used for subject template. Subject must be single line plain text string.

`class PasswordResetDoneView` [source]

URL name: `password_reset_done`

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

Note: If the email address provided does not exist in the system, the user is inactive, or has an unusable password, the user will still be redirected to this view but no email will be sent.

Attributes:

`template_name`

The full name of a template to use. Defaults to `registration/password_reset_done.html` if not supplied.

`extra_context`

A dictionary of context data that will be added to the default context data passed to the template.

`class PasswordResetConfirmView` [source]

URL name: `password_reset_confirm`

Presents a form for entering a new password.

Keyword arguments from the URL:

- `uidb64` : The user's id encoded in base 64.
- `token` : Token to check that the password is valid.

Attributes:

`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` .

`post_reset_login`

A boolean indicating if the user should be automatically authenticated after a successful password reset. Defaults to `False` .

`post_reset_login_backend`

A dotted path to the authentication backend to use when authenticating a user if `post_reset_login` is `True` . Required only if you have multiple `AUTHENTICATION_BACKENDS` configured. Defaults to `None` .

`form_class`

Form that will be used to set the password. Defaults to `SetPasswordForm`.

`success_url`

URL to redirect after the password reset done. Defaults to `'password_reset_complete'`.

`extra_context`

A dictionary of context data that will be added to the default context data passed to the template.

`reset_url_token`

Token parameter displayed as a component of password reset URLs. Defaults to `'set-password'`.

Template context:

- `form` : The form (see `form_class` above) for setting the new user's password.
- `validlink` : Boolean, True if the link (combination of `uidb64` and `token`) is valid or unused yet.

`class PasswordResetCompleteView` [source]

URL name: `password_reset_complete`

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

Attributes:

`template_name`

The full name of a template to display the view. Defaults to `registration/password_reset_complete.html`.

`extra_context`

A dictionary of context data that will be added to the default context data passed to the template.

Helper functions

`redirect_to_login(next, login_url=None, redirect_field_name='next')` [source]

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. Defaults to `settings.LOGIN_URL` if not supplied.
- `redirect_field_name` : The name of a `GET` field containing the URL to redirect to after login. 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`:

Note: The built-in authentication forms make certain assumptions about the user model that they are working with. If you're using a custom user model, it may be necessary to define your own forms for the authentication system. For more information, refer to the documentation about using the built-in authentication forms with custom user models.

`class AdminPasswordChangeForm` [source]

A form used in the admin interface to change a user's password, including the ability to set an `unusable password`, which blocks the user from logging in with password-based authentication.

Takes the `user` as the first positional argument.

Changed in Django 5.1:

Option to disable (or reenable) password-based authentication was added.

`class AuthenticationForm` [source]

A form for logging a user in.

Takes `request` as its first positional argument, which is stored on the form instance for use by sub-classes.

`confirm_login_allowed(user)` [source]

By default, `AuthenticationForm` rejects users whose `is_active` flag is set to `False`. You may override this behavior with a custom policy to determine which users can log in. Do this with a custom form that subclasses `AuthenticationForm` and overrides the `confirm_login_allowed()` method. This method should raise a `ValidationError` if the given user may not log in.

For example, to allow all users to log in regardless of "active" status:

```
from django.contrib.auth.forms import AuthenticationForm


class AuthenticationFormWithInactiveUsersOkay(AuthenticationForm):
    def confirm_login_allowed(self, user):
        pass
```

(In this case, you'll also need to use an authentication backend that allows inactive users, such as `AllowAllUsersModelBackend`.)

Or to allow only some active users to log in:

```
class PickyAuthenticationForm(AuthenticationForm):
    def confirm_login_allowed(self, user):
        if not user.is_active:
            raise ValidationError(
                _("This account is inactive."),
                code="inactive",
            )
        if user.username.startswith("b"):
            raise ValidationError(
                _("Sorry, accounts starting with 'b' aren't welcome here."),
                code="no_b_users",
            )
```

`class PasswordChangeForm` [source]

A form for allowing a user to change their password.

`class PasswordResetForm` [source]

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

`send_mail(subject_template_name, email_template_name, context, from_email, to_email, html_email_template_name=None)` [source]

Uses the arguments to send an `EmailMultiAlternatives`. Can be overridden to customize how the email is sent to the user.

Parameters:

- **subject_template_name** – the template for the subject.
- **email_template_name** – the template for the email body.
- **context** – context passed to the `subject_template`, `email_template`, and `html_email_template` (if it is not `None`).
- **from_email** – the sender's email.
- **to_email** – the email of the requester.
- **html_email_template_name** – the template for the HTML body; defaults to `None`, in which case a plain text email is sent.

By default, `save()` populates the `context` with the same variables that `PasswordResetView` passes to its email context.

`class SetPasswordForm` [source]

A form that lets a user change their password without entering the old password.

```
class UserChangeForm                                                                 [source]
```

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

```
class BaseUserCreationForm                                                           [source]
```

A `ModelForm` for creating a new user. This is the recommended base class if you need to customize the user creation form.

It has four fields: `username` (from the user model), `password1`, `password2`, and `usable_password` (the latter is enabled by default). If `usable_password` is enabled, it verifies that `password1` and `password2` are non empty and match, validates the password using `validate_password()`, and sets the user's password using `set_password()`. If `usable_password` is disabled, no password validation is done, and password-based authentication is disabled for the user by calling `set_unusable_password()`.

Changed in Django 5.1:
Option to create users with disabled password-based authentication was added.

```
class UserCreationForm                                                               [source]
```

Inherits from `BaseUserCreationForm`. To help prevent confusion with similar usernames, the form doesn't allow usernames that differ only in case.

Authentication data in templates

The currently logged-in user and their 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 the `'django.contrib.auth.context_processors.auth'` context processor is enabled. It is in the default generated settings file. 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.

Evaluating a single-attribute lookup of `{{ perms }}` as a boolean is a proxy to `User.has_module_perms()`. For example, to check if the logged-in user has any permissions in the `foo` app:

```
{% if perms.foo %}
```

Evaluating a two-level-attribute lookup as a boolean is a proxy to `User.has_perm()`. For example, to check if the logged-in user has the permission `foo.add_vote`:

```
{% if perms.foo.add_vote %}
```

Here's a more complete example of checking permissions in a template:

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.add_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.add_driving %}
```

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

It is possible to also look permissions up by `{% if in %}` statements. For example:

```
{% if 'foo' in perms %}
    {% if 'foo.add_vote' in perms %}
        <p>In lookup works, too.</p>
    {% endif %}
{% endif %}
```

Managing users in the admin

When you have both `django.contrib.admin` and `django.contrib.auth` installed, the admin provides a convenient way to view and manage users, groups, and permissions. Users can be created and deleted like any Django model. Groups can be created, and permissions can be assigned to users or groups. A log of user edits to models made within the admin is also stored and displayed.

Creating users

You should 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. Alternatively, on this page, you can choose a username and disable password-based authentication for the user.

Also note: if you want a user account to be able to create users using the Django admin site, you'll need to give them permission to add users *and* change users (i.e., the "Add user" and "Change user" permissions). If an account has permission to add users but not to change them, that account 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.

Be thoughtful about how you allow users to manage permissions. If you give a non-superuser the ability to edit users, this is ultimately the same as giving them superuser status because they will be able to elevate permissions of users including themselves!

Changing passwords

User passwords are not displayed in the admin (nor stored in the database), but the password storage details are displayed. Included in the display of this information is a link to a password change form that allows admins to change or unset user passwords.