Customizing authentication in Django

The authentication that comes with Django is good enough for most common cases, but you may have needs not met by the out-of-the-box defaults. Customizing authentication in your projects requires understanding what points of the provided system are extensible or replaceable. This document provides details about how the auth system can be customized.

Authentication backends provide an extensible system for when a username and password stored with the user model need to be authenticated against a different service than Django's default.

You can give your models custom permissions that can be checked through Django's authorization system.

You can extend the default `User` model, or substitute a completely customized model.

Other authentication sources

There may be times you 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 – 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 list 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 backend that checks the Django users database and queries the built-in permissions. It does not provide protection against brute force attacks via any rate limiting mechanism. You may either implement your own rate limiting mechanism in a custom auth backend, or use the mechanisms provided by most web servers.

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.

If a backend raises a `PermissionDenied` exception, authentication will immediately fail. Django won't check the backends that follow.

Note: Once a user has authenticated, Django stores which backend was used to authenticate the user in the user's session, and reuses 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 to execute `Session.objects.all().delete()`.

Writing an authentication backend

An authentication backend is a class that implements two required methods: `get_user(user_id)` and `authenticate(request, **credentials)`, as well as a set of optional permission related authorization methods.

The `get_user` method takes a `user_id` – which could be a username, database ID or whatever, but has to be the primary key of your user object – and returns a user object or `None`.

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

```
from django.contrib.auth.backends import BaseBackend


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

But it could also authenticate a token, like so:

```
from django.contrib.auth.backends import BaseBackend


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

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

`request` is an `HttpRequest` and may be `None` if it wasn't provided to `authenticate()` (which passes it on to the backend).

The Django admin is tightly coupled to the Django User object. 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.backends import BaseBackend
from django.contrib.auth.hashers import check_password
from django.contrib.auth.models import User


class SettingsBackend(BaseBackend):
    """
    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 = 'pbkdf2_sha256$30000$Vo0VlMnkR4Bk$qEvtdyZRWTcOsCnI/oQ7fVOu1XAURIZYoOZ3iq8Dr4M='
    """

    def authenticate(self, request, 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. There's no need to set a password
                # because only the password from settings.py is checked.
                user = User(username=username)
                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 and its manager will delegate permission lookup functions (`get_user_permissions()`, `get_group_permissions()`, `get_all_permissions()`, `has_perm()`, `has_module_perms()`, and `with_perm()`) 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.

If a backend raises a `PermissionDenied` exception in `has_perm()` or `has_module_perms()`, the authorization will immediately fail and Django won't check the backends that follow.

A backend could implement permissions for the magic admin like this:

```
from django.contrib.auth.backends import BaseBackend


class MagicAdminBackend(BaseBackend):
    def has_perm(self, user_obj, perm, obj=None):
        return user_obj.username == settings.ADMIN_LOGIN
```

This gives full permissions to the user granted access in the above example. Notice that in addition to the same arguments given to the associated `django.contrib.auth.models.User` functions, the backend auth functions all take the user object, which may be an anonymous user, as an argument.

A full authorization implementation can be found in the `ModelBackend` class 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

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 websites 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, the user object passed to an authentication backend may be an `django.contrib.auth.models.AnonymousUser` object, allowing the backend to specify custom authorization behavior for anonymous users. This is especially useful for the authors of reusable apps, who can delegate all questions of authorization to the auth backend, rather than needing settings, for example, to control anonymous access.

Authorization for inactive users

An inactive user is one that has its `is_active` field set to `False` . The `ModelBackend` and `RemoteUserBackend` authentication backends prohibits these users from authenticating. If a custom user model doesn't have an `is_active` field, all users will be allowed to authenticate.

You can use `AllowAllUsersModelBackend` or `AllowAllUsersRemoteUserBackend` if you want to allow inactive users to authenticate.

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

Do not forget to test for the `is_active` attribute of the user in your own backend permission 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). An authentication backend will receive the keyword parameters `obj` and `user_obj` for each object related authorization method and can return the object level permission as appropriate.

Custom permissions

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

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

```
class Task(models.Model):
    ...

    class Meta:
        permissions = [
            ("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 migrate` (the function that creates permissions is connected to the `post_migrate` signal). Your code is in charge of checking the value of these permissions when a user is trying to access the functionality provided by the application (changing the status of tasks or closing tasks.) Continuing the above example, the following checks if a user may close tasks:

```
user.has_perm("app.close_task")
```

Extending the existing `User` model

There are two ways to extend the default `User` model without substituting your own model. If the changes you need are purely behavioral, and don't require any change to what is stored in the database, you can create a proxy model based on `User`. This allows for any of the features offered by proxy models including default ordering, custom managers, or custom model methods.

If you wish to store information related to `User`, you can use a `OneToOneField` to a model containing the fields for additional information. This one-to-one model is often called a profile model, as it might store non-auth related information about a site user. For example you might create an Employee model:

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


class Employee(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    department = models.CharField(max_length=100)
```

Assuming an existing Employee Fred Smith who has both a User and Employee model, you can access the related information using Django's standard related model conventions:

```
>>> u = User.objects.get(username="fsmith")
>>> freds_department = u.employee.department
```

To add a profile model's fields to the user page in the admin, define an `InlineModelAdmin` (for this example, we'll use a `StackedInline`) in your app's `admin.py` and add it to a `UserAdmin` class which is registered with the `User` class:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import User

from my_user_profile_app.models import Employee


# Define an inline admin descriptor for Employee model
# which acts a bit like a singleton
class EmployeeInline(admin.StackedInline):
    model = Employee
    can_delete = False
    verbose_name_plural = "employee"


# Define a new User admin
class UserAdmin(BaseUserAdmin):
    inlines = [EmployeeInline]


# Re-register UserAdmin
admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

These profile models are not special in any way - they are just Django models that happen to have a one-to-one link with a user model. As such, they aren't auto created when a user is created, but a `django.db.models.signals.post_save` could be used to create or update related models as appropriate.

Using related models results in additional queries or joins to retrieve the related data. Depending on your needs, a custom user model that includes the related fields may be your better option, however, existing relations to the default user model within your project's apps may justify the extra database load.

Substituting a custom `User` model

Some kinds of projects may have authentication requirements for which Django's built-in `User` model is not always appropriate. For instance, on some sites it makes more sense to use an email address as your identification token instead of a username.

Django allows you to override the default user model by providing a value for the `AUTH_USER_MODEL` setting that references a custom model:

```
AUTH_USER_MODEL = "myapp.MyUser"
```

This dotted pair describes the `label` of the Django app (which must be in your `INSTALLED_APPS`), and the name of the Django model that you wish to use as your user model.

Using a custom user model when starting a project

If you're starting a new project, it's highly recommended to set up a custom user model, even if the default `User` model is sufficient for you. This model behaves identically to the default user model, but you'll be able to customize it in the future if the need arises:

```
from django.contrib.auth.models import AbstractUser


class User(AbstractUser):
    pass
```

Don't forget to point `AUTH_USER_MODEL` to it. Do this before creating any migrations or running `manage.py migrate` for the first time.

Also, register the model in the app's `admin.py`:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from .models import User

admin.site.register(User, UserAdmin)
```

Changing to a custom user model mid-project

Changing `AUTH_USER_MODEL` after you've created database tables is significantly more difficult since it affects foreign keys and many-to-many relationships, for example.

This change can't be done automatically and requires manually fixing your schema, moving your data from the old user table, and possibly manually reapplying some migrations. See #25313 for an outline of the steps.

Due to limitations of Django's dynamic dependency feature for swappable models, the model referenced by `AUTH_USER_MODEL` must be created in the first migration of its app (usually called `0001_initial`); otherwise, you'll have dependency issues.

In addition, you may run into a `CircularDependencyError` when running your migrations as Django won't be able to automatically break the dependency loop due to the dynamic dependency. If you see this error, you should break the loop by moving the models depended on by your user model into a second migration. (You can try making two normal models that have a `ForeignKey` to each other and seeing how `makemigrations` resolves that circular dependency if you want to see how it's usually done.)

Reusable apps and `AUTH_USER_MODEL`

Reusable apps shouldn't implement a custom user model. A project may use many apps, and two reusable apps that implemented a custom user model couldn't be used together. If you need to store per user information in your app, use a `ForeignKey` or `OneToOneField` to `settings.AUTH_USER_MODEL` as described below.

Referencing the `User` model

`get_user_model()` [source]

Instead of referring to `User` directly, you should reference the user model using `django.contrib.auth.get_user_model()`. This method will return the currently active user model – the custom user model if one is specified, or `User` otherwise.

When you define a foreign key or many-to-many relations to the user model, you should specify the custom model using the `AUTH_USER_MODEL` setting. For example:

```
from django.conf import settings
from django.db import models


class Article(models.Model):
    author = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
```

When connecting to signals sent by the user model, you should specify the custom model using the `AUTH_USER_MODEL` setting. For example:

```
from django.conf import settings
from django.db.models.signals import post_save


def post_save_receiver(sender, instance, created, **kwargs):
    pass


post_save.connect(post_save_receiver, sender=settings.AUTH_USER_MODEL)
```

Generally speaking, it's easiest to refer to the user model with the `AUTH_USER_MODEL` setting in code that's executed at import time, however, it's also possible to call `get_user_model()` while Django is importing models, so you could use `models.ForeignKey(get_user_model(), ...)`.

If your app is tested with multiple user models, using `@override_settings(AUTH_USER_MODEL=...)` for example, and you cache the result of `get_user_model()` in a module-level variable, you may need to listen to the `setting_changed` signal to clear the cache. For example:

```
from django.apps import apps
from django.contrib.auth import get_user_model
from django.core.signals import setting_changed
from django.dispatch import receiver


@receiver(setting_changed)
def user_model_swapped(*, setting, **kwargs):
    if setting == "AUTH_USER_MODEL":
        apps.clear_cache()
        from myapp import some_module

        some_module.UserModel = get_user_model()
```

Specifying a custom user model

When you start your project with a custom user model, stop to consider if this is the right choice for your project.

Keeping all user related information in one model removes the need for additional or more complex database queries to retrieve related models. On the other hand, it may be more suitable to store app-specific user information in a model that has a relation with your custom user model. That allows each app to specify its own user data requirements without potentially conflicting or breaking assumptions by other apps. It also means that you would keep your user model as simple as possible, focused on authentication, and following the minimum requirements Django expects custom user models to meet.

If you use the default authentication backend, then your model must have a single unique field that can be used for identification purposes. This can be a username, an email address, or any other unique attribute. A non-unique username field is allowed if you use a custom authentication backend that can support it.

The easiest way to construct a compliant custom user model is to inherit from `AbstractBaseUser`. `AbstractBaseUser` provides the core implementation of a user model, including hashed passwords and tokenized password resets. You must then provide some key implementation details:

class models.CustomUser

USERNAME_FIELD

A string describing the name of the field on the user model that is used as the unique identifier. This will usually be a username of some kind, but it can also be an email address, or any other unique identifier. The field *must* be unique (e.g. have `unique=True` set in its definition), unless you use a custom authentication backend that can support non-unique usernames.

In the following example, the field `identifier` is used as the identifying field:

```
class MyUser(AbstractBaseUser):
    identifier = models.CharField(max_length=40, unique=True)
    ...
    USERNAME_FIELD = "identifier"
```

EMAIL_FIELD

A string describing the name of the email field on the `User` model. This value is returned by `get_email_field_name()`.

REQUIRED_FIELDS

A list of the field names that will be prompted for when creating a user via the `createsuperuser` management command. The user will be prompted to supply a value for each of these fields. It must include any field for which `blank` is `False` or undefined and may include additional fields you want prompted for when a user is created interactively. `REQUIRED_FIELDS` has no effect in other parts of Django, like creating a user in the admin.

For example, here is the partial definition for a user model that defines two required fields - a date of birth and height:

```
class MyUser(AbstractBaseUser):
    ...
    date_of_birth = models.DateField()
    height = models.FloatField()
    ...
    REQUIRED_FIELDS = ["date_of_birth", "height"]
```

Note: `REQUIRED_FIELDS` must contain all required fields on your user model, but should *not* contain the `USERNAME_FIELD` or `password` as these fields will always be prompted for.

`is_active`

A boolean attribute that indicates whether the user is considered "active". This attribute is provided as an attribute on `AbstractBaseUser` defaulting to `True`. How you choose to implement it will depend on the details of your chosen auth backends. See the documentation of the `is_active attribute on the built-in user model` for details.

`get_full_name()`

Optional. A longer formal identifier for the user such as their full name. If implemented, this appears alongside the username in an object's history in `django.contrib.admin`.

`get_short_name()`

Optional. A short, informal identifier for the user such as their first name. If implemented, this replaces the username in the greeting to the user in the header of `django.contrib.admin`.

Importing `AbstractBaseUser`: `AbstractBaseUser` and `BaseUserManager` are importable from `django.contrib.auth.base_user` so that they can be imported without including `django.contrib.auth` in `INSTALLED_APPS`.

The following attributes and methods are available on any subclass of `AbstractBaseUser`:

`class models.AbstractBaseUser`

`get_username()`

Returns the value of the field nominated by `USERNAME_FIELD`.

`clean()`

Normalizes the username by calling `normalize_username()`. If you override this method, be sure to call `super()` to retain the normalization.

`classmethod get_email_field_name()`

Returns the name of the email field specified by the `EMAIL_FIELD` attribute. Defaults to `'email'` if `EMAIL_FIELD` isn't specified.

`classmethod normalize_username(username)`

Applies NFKC Unicode normalization to usernames so that visually identical characters with different Unicode code points are considered identical.

`is_authenticated`

Read-only attribute which is always `True` (as opposed to `AnonymousUser.is_authenticated` which is always `False`). 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 or has a valid session. Even though normally you will check this attribute on `request.user` to find out whether it has been populated by the `AuthenticationMiddleware` (representing the currently logged-in user), you should know this attribute is `True` for any `User` instance.

`is_anonymous`

Read-only attribute which is always `False`. This is a way of differentiating `User` and `AnonymousUser` objects. Generally, you should prefer using `is_authenticated` to this attribute.

`set_password(raw_password)`

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

When the raw_password is `None`, the password will be set to an unusable password, as if `set_unusable_password()` were used.

`check_password(raw_password)`

`acheck_password(raw_password)`

Asynchronous version: `acheck_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.)

Changed in Django 5.0:
`acheck_password()` method was added.

`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 `AbstractBaseUser` 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_session_auth_hash()`

Returns an HMAC of the password field. Used for Session invalidation on password change.

`get_session_auth_fallback_hash()`

Yields the HMAC of the password field using `SECRET_KEY_FALLBACKS`. Used by `get_user()`.

`AbstractUser` subclasses `AbstractBaseUser`:

`class models.AbstractUser`

`clean()`

Normalizes the email by calling `BaseUserManager.normalize_email()`. If you override this method, be sure to call `super()` to retain the normalization.

Writing a manager for a custom user model

You should also define a custom manager for your user model. If your user model defines `username`, `email`, `is_staff`, `is_active`, `is_superuser`, `last_login`, and `date_joined` fields the same as Django's default user, you can install Django's `UserManager`; however, if your user model defines different fields, you'll need to define a custom manager that extends `BaseUserManager` providing two additional methods:

`class models.CustomUserManager`

`create_user(username_field, password=None, **other_fields)`

The prototype of `create_user()` should accept the username field, plus all required fields as arguments. For example, if your user model uses `email` as the username field, and has `date_of_birth` as a required field, then `create_user` should be defined as:

```
def create_user(self, email, date_of_birth, password=None):
    # create user here
    ...
```

`create_superuser(username_field, password=None, **other_fields)`

The prototype of `create_superuser()` should accept the username field, plus all required fields as arguments. For example, if your user model uses `email` as the username field, and has `date_of_birth` as a required field, then `create_superuser` should be defined as:

```
def create_superuser(self, email, date_of_birth, password=None):
    # create superuser here
    ...
```

For a `ForeignKey` in `USERNAME_FIELD` or `REQUIRED_FIELDS`, these methods receive the value of the `to_field` (the `primary_key` by default) of an existing instance.

`BaseUserManager` provides the following utility methods:

class models.BaseUserManager

classmethod normalize_email(email)

Normalizes email addresses by lowercasing the domain portion of the email address.

get_by_natural_key(username)

Retrieves a user instance using the contents of the field nominated by `USERNAME_FIELD`.

Extending Django's default `User`

If you're entirely happy with Django's `User` model, but you want to add some additional profile information, you could subclass `django.contrib.auth.models.AbstractUser` and add your custom profile fields, although we'd recommend a separate model as described in Specifying a custom user model. `AbstractUser` provides the full implementation of the default `User` as an abstract model.

Custom users and the built-in auth forms

Django's built-in forms and views make certain assumptions about the user model that they are working with.

The following forms are compatible with any subclass of `AbstractBaseUser`:

- `AuthenticationForm`: Uses the username field specified by `USERNAME_FIELD`.
- `SetPasswordForm`
- `PasswordChangeForm`
- `AdminPasswordChangeForm`

The following forms make assumptions about the user model and can be used as-is if those assumptions are met:

- `PasswordResetForm`: Assumes that the user model has a field that stores the user's email address with the name returned by `get_email_field_name()` (`email` by default) that can be used to identify the user and a boolean field named `is_active` to prevent password resets for inactive users.

Finally, the following forms are tied to `User` and need to be rewritten or extended to work with a custom user model:

- `UserCreationForm`
- `UserChangeForm`

If your custom user model is a subclass of `AbstractUser`, then you can extend these forms in this manner:

```
from django.contrib.auth.forms import UserCreationForm
from myapp.models import CustomUser


class CustomUserCreationForm(UserCreationForm):
    class Meta(UserCreationForm.Meta):
        model = CustomUser
        fields = UserCreationForm.Meta.fields + ("custom_field",)
```

Custom users and `django.contrib.admin`

If you want your custom user model to also work with the admin, your user model must define some additional attributes and methods. These methods allow the admin to control access of the user to admin content:

class models.CustomUser

is_staff

Returns `True` if the user is allowed to have access to the admin site.

is_active

Returns `True` if the user account is currently active.

`has_perm(perm, obj=None):`

Returns `True` if the user has the named permission. If `obj` is provided, the permission needs to be checked against a specific object instance.

`has_module_perms(app_label):`

Returns `True` if the user has permission to access models in the given app.

You will also need to register your custom user model with the admin. If your custom user model extends `django.contrib.auth.models.AbstractUser`, you can use Django's existing `django.contrib.auth.admin.UserAdmin` class. However, if your user model extends `AbstractBaseUser`, you'll need to define a custom `ModelAdmin` class. It may be possible to subclass the default `django.contrib.auth.admin.UserAdmin`; however, you'll need to override any of the definitions that refer to fields on `django.contrib.auth.models.AbstractUser` that aren't on your custom user class.

Note: If you are using a custom `ModelAdmin` which is a subclass of `django.contrib.auth.admin.UserAdmin`, then you need to add your custom fields to `fieldsets` (for fields to be used in editing users) and to `add_fieldsets` (for fields to be used when creating a user). For example:

```
from django.contrib.auth.admin import UserAdmin


class CustomUserAdmin(UserAdmin):
    ...
    fieldsets = UserAdmin.fieldsets + ((None, {"fields": ["custom_field"]}),)
    add_fieldsets = UserAdmin.add_fieldsets + ((None, {"fields": ["custom_field"]}),)
```

See a full example for more details.

Custom users and permissions

To make it easy to include Django's permission framework into your own user class, Django provides `PermissionsMixin`. This is an abstract model you can include in the class hierarchy for your user model, giving you all the methods and database fields necessary to support Django's permission model.

`PermissionsMixin` provides the following methods and attributes:

`class models.PermissionsMixin`

`is_superuser`

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

`get_user_permissions(obj=None)`

Returns a set of permission strings that the user has directly.

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

`get_group_permissions(obj=None)`

Returns a set of permission strings that the user has, through their groups.

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.

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). If `User.is_active` and `is_superuser` are both `True`, this method always returns `True`.

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 `User.is_active` and `is_superuser` are both `True`, this method always returns `True`.

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 `User.is_active` and `is_superuser` are both `True`, this method always returns `True`.

`PermissionsMixin` **and** `ModelBackend`: If you don't include the `PermissionsMixin`, you must ensure you don't invoke the permissions methods on `ModelBackend`. `ModelBackend` assumes that certain fields are available on your user model. If your user model doesn't provide those fields, you'll receive database errors when you check permissions.

Custom users and proxy models

One limitation of custom user models is that installing a custom user model will break any proxy model extending `User`. Proxy models must be based on a concrete base class; by defining a custom user model, you remove the ability of Django to reliably identify the base class.

If your project uses proxy models, you must either modify the proxy to extend the user model that's in use in your project, or merge your proxy's behavior into your `User` subclass.

A full example

Here is an example of an admin-compliant custom user app. This user model uses an email address as the username, and has a required date of birth; it provides no permission checking beyond an `admin` flag on the user account. This model would be compatible with all the built-in auth forms and views, except for the user creation forms. This example illustrates how most of the components work together, but is not intended to be copied directly into projects for production use.

This code would all live in a `models.py` file for a custom authentication app:

```
from django.db import models
from django.contrib.auth.models import BaseUserManager, AbstractBaseUser


class MyUserManager(BaseUserManager):
    def create_user(self, email, date_of_birth, password=None):
        """
        Creates and saves a User with the given email, date of
        birth and password.
        """
        if not email:
            raise ValueError("Users must have an email address")

        user = self.model(
            email=self.normalize_email(email),
            date_of_birth=date_of_birth,
        )

        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, date_of_birth, password=None):
        """
        Creates and saves a superuser with the given email, date of
        birth and password.
        """
        user = self.create_user(
            email,
            password=password,
            date_of_birth=date_of_birth,
        )
        user.is_admin = True
        user.save(using=self._db)
        return user


class MyUser(AbstractBaseUser):
    email = models.EmailField(
        verbose_name="email address",
        max_length=255,
        unique=True,
    )
```

```
    date_of_birth = models.DateField()
    is_active = models.BooleanField(default=True)
    is_admin = models.BooleanField(default=False)

    objects = MyUserManager()

    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ["date_of_birth"]

    def __str__(self):
        return self.email

    def has_perm(self, perm, obj=None):
        "Does the user have a specific permission?"
        # Simplest possible answer: Yes, always
        return True

    def has_module_perms(self, app_label):
        "Does the user have permissions to view the app `app_label`?"
        # Simplest possible answer: Yes, always
        return True

    @property
    def is_staff(self):
        "Is the user a member of staff?"
        # Simplest possible answer: All admins are staff
        return self.is_admin
```

Then, to register this custom user model with Django's admin, the following code would be required in the app's `admin.py` file:

```
from django import forms
from django.contrib import admin
from django.contrib.auth.models import Group
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.forms import ReadOnlyPasswordHashField
from django.core.exceptions import ValidationError

from customauth.models import MyUser


class UserCreationForm(forms.ModelForm):
    """A form for creating new users. Includes all the required
    fields, plus a repeated password."""

    password1 = forms.CharField(label="Password", widget=forms.PasswordInput)
    password2 = forms.CharField(
        label="Password confirmation", widget=forms.PasswordInput
    )

    class Meta:
        model = MyUser
        fields = ["email", "date_of_birth"]

    def clean_password2(self):
        # Check that the two password entries match
        password1 = self.cleaned_data.get("password1")
        password2 = self.cleaned_data.get("password2")
        if password1 and password2 and password1 != password2:
            raise ValidationError("Passwords don't match")
        return password2

    def save(self, commit=True):
        # Save the provided password in hashed format
        user = super().save(commit=False)
        user.set_password(self.cleaned_data["password1"])
        if commit:
            user.save()
        return user


class UserChangeForm(forms.ModelForm):
    """A form for updating users. Includes all the fields on
    the user, but replaces the password field with admin's
    disabled password hash display field.
    """

    password = ReadOnlyPasswordHashField()
```

```
    class Meta:
        model = MyUser
        fields = ["email", "password", "date_of_birth", "is_active", "is_admin"]


class UserAdmin(BaseUserAdmin):
    # The forms to add and change user instances
    form = UserChangeForm
    add_form = UserCreationForm

    # The fields to be used in displaying the User model.
    # These override the definitions on the base UserAdmin
    # that reference specific fields on auth.User.
    list_display = ["email", "date_of_birth", "is_admin"]
    list_filter = ["is_admin"]
    fieldsets = [
        (None, {"fields": ["email", "password"]}),
        ("Personal info", {"fields": ["date_of_birth"]}),
        ("Permissions", {"fields": ["is_admin"]}),
    ]
    # add_fieldsets is not a standard ModelAdmin attribute. UserAdmin
    # overrides get_fieldsets to use this attribute when creating a user.
    add_fieldsets = [
        (
            None,
            {
                "classes": ["wide"],
                "fields": ["email", "date_of_birth", "password1", "password2"],
            },
        ),
    ]
    search_fields = ["email"]
    ordering = ["email"]
    filter_horizontal = []


# Now register the new UserAdmin...
admin.site.register(MyUser, UserAdmin)
# ... and, since we're not using Django's built-in permissions,
# unregister the Group model from admin.
admin.site.unregister(Group)
```

Finally, specify the custom model as the default user model for your project using the `AUTH_USER_MODEL` setting in your `settings.py`:

```
AUTH_USER_MODEL = "customauth.MyUser"
```