Learning Objectives

- **Explain why it is better to use a custom User model from the beginning**
- **Create a custom User model**
- **Identify when to use a custom model or a separate model**
- **Change authentication from username to email**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Add a Custom User Model Part 1

Custom User Model

Add a Custom User Model Part 1

Django provides a good default user model, but you might want to use your own to customize how authorization works. One reason for this might be to add extra fields to the User, or allow logging in with an email address instead of an username. We could also use a separate model to store extra fields, for example a `Profile` model that was related to a user with a `OneToOne` field. We'll look at both of these scenarios later.

The recommendation for Django is to use a custom User model right from the start of the Project, even if you don't intend to take full advantage of one. The reason for this is that it's quite a laborious process to change your User model, should you decide to in the future, and Django doesn't have built-in support for migrating data to a new User class.

If you do decide to set up your own User model when you start your project, it's fairly simple. We'll explain the basic process and go through it in detail in the Try It Out section.

- Create a user model that inherits from `django.contrib.auth.model.AbstractUser` (it doesn't need to be called `User`). This can be in any app in your project, but you should consider creating an app just for authentication-related models, templates, forms and views.
- Add the `AUTH_USER_MODEL` setting to point to the new Model. If you've added the model inside a new app, make sure to add that to `INSTALLED_APPS`.
- Register the Model in the `admin.py` for the app in which it was created.

That's it, Django will use your User model instead. The process for migrating data is a bit more involved, and will depend on the status of your project and how much data it has.

Since we have a small project without much data, migration is not too difficult, but takes a few steps.

Try It Out

Let's run through how to add the custom User model and then migrate our data to it. Some of these steps you only need to do as part of data migration, so you would skip these steps if working with a new project.

First we need to take a dump of all the data we want to keep so we can restore it. This wouldn't need to be done with a new Project. Django provides this functionality with the `dumpdata` management command. We only need our `blog` models and `auth.User`.

Run this command in the terminal to dump the data to a file called `data.json` (specified with the `-o` flag).

```
python3 manage.py dumpdata -o data.json blog.Comment blog.Tag
        blog.Post auth.User
```

Then open the `data.json` file in a text editor, and replace all occurrences of `auth.User` with `blango_auth.User`. Use "Find" from the Codio menu bar to help you find `auth.User`. The number of replacements will match the number of users you have. This is so that when we load the data the user information goes into our new table.

Open data.json

Now we'll set up our new User class. Since we're going to do a few authentication related changes, we'll start a new Django app to keep it all together. This will also allow our `blog` app to be self-contained and be reused in other Django projects. Our User model won't have any attributes but it will mean that the users are created in the right table.

First create a new Django app called `blango_auth`.

```
python3 manage.py startapp blango_auth
```

Open `blango_auth/models.py` and add an `AbstractUser` import at the top of the file:

Open blango_auth/models.py

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

Then create a `User` model.

```
class User(AbstractUser):
    pass
```

This is the only model change we need to make. Remember the other models are related to the User model using `settings.AUTH_USER_MODEL` so we don't actually need to change any of their foreign keys. But we do need to update that setting, as well as add `blango_auth` to `INSTALLED_APPS`.

Open `blog/settings.py` and add `AUTH_USER_MODEL = "blango_auth.User"` as an attribute to the `Dev` class. Then add `blango_auth` to `INSTALLED_APPS`, put it before the `blog` entry.

Open blog/settings.py

Next, to be able to edit/create Users in the Django Admin site, we'll need to configure our model in `blango_auth/admin.py`.

Open blango_auth/admin.py

We'll use Django's `django.contrib.auth.admin.UserAdmin` class as the `ModelAdmin` for our `User` model.

To set up the admin is three steps. Import `UserAdmin`:

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

Import the `User` model:

```
from blango_auth.models import User
```

Register the `User` model for the `UserAdmin`.

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

The complete file should look like this:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from blango_auth.models import User
# Register your models here.

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

Add a Custom User Model Part 2

Add a Custom User Model Part 2

Next we'll create the migration file for `blango_auth.User` with the `makemigrations` management command in the terminal:

```
python3 manage.py makemigrations
```

Your output should be:

```
Migrations for 'blango_auth':
  blango_auth/migrations/0001_initial.py
    - Create model User
```

The next step is a bit scary, and something that only needs to be done for our specific case since we're migrating data. We'll delete our `db.sqlite3`. Since we have a dump of the data, we can restore it again afterward.

```
rm db.sqlite3
```

Then we'll apply our migrations to the database with the `migrate` management command.

```
python3 manage.py migrate
```

Your output should be:

```
Operations to perform:
  Apply all migrations: admin, auth, blango_auth, blog,
      contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
…
```

Then we can reimport the data, another step only necessary in our specific case. We'll use the `loaddata` management command. Specify the JSON file we dumped the data to (`data.json`).

```
python3 manage.py loaddata data.json
```

Your output should look something like:

```
Installed 21 object(s) from 1 fixture(s)
```

Adding a model means the content type IDs have changed, so we'll need to update all the `Comment` objects to point to the new content type of `Post`. This is the last thing that needs to be done as part of the migration process.

We can do this with an `update`. Start a Django Python shell with the `shell` management command.

```
python3 manage.py shell
```

Then you can copy and paste these four lines of code.

```
from django.contrib.contenttypes.models import ContentType
from blog.models import Comment, Post
post_type = ContentType.objects.get_for_model(Post)
Comment.objects.update(content_type=post_type)
```

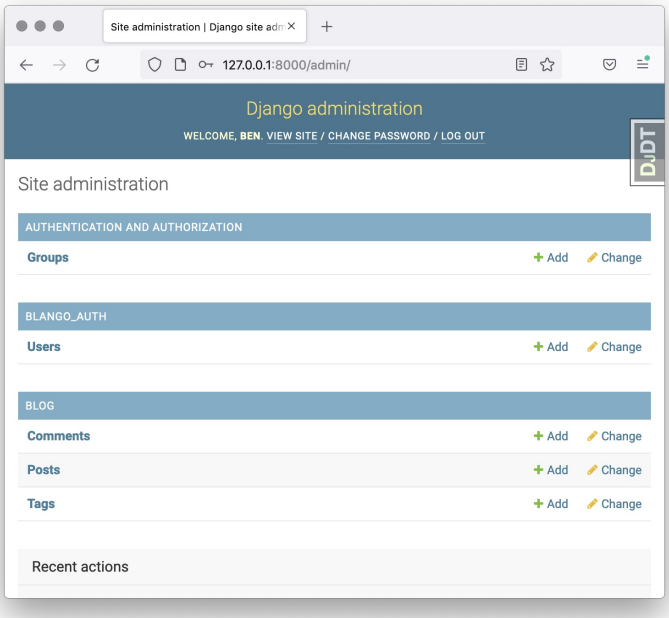
Since all our `Comment` objects are for `Post` objects, we don't need to filter to find any specific ones.

That was just a few steps, and our project is quite simple! You can see why the recommendation is to use a custom User model right from the start.

Close the Python shell with *Ctrl+C*. Then start the Django development server then load up the post list page in your browser. You should see the posts are there and have comments.

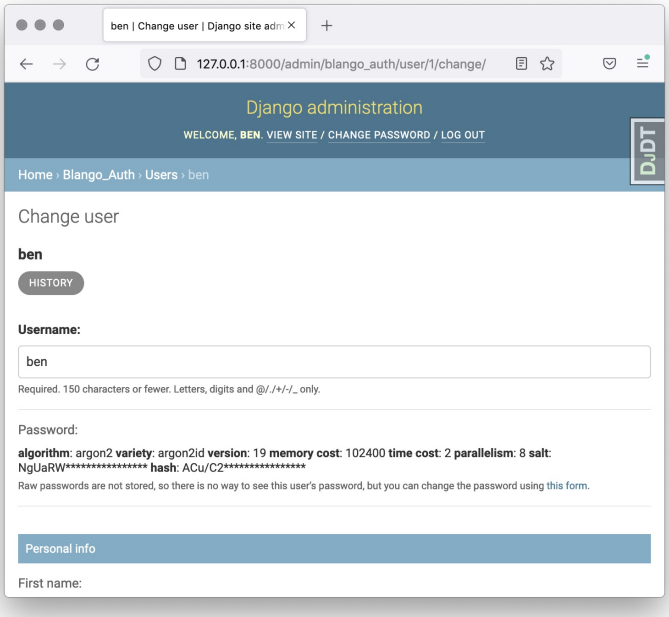
View Blog

Log in to the Django Admin site and it mostly looks the same, however you'll notice `Users` now under the *Blango_Auth* heading, rather than *Authentication and Authorization*.



Blango Auth heading

If you click through to see a User detail page, you'll notice that because we're using `UserAdmin`, we see the same form as we did for the built-in Django user.



same form

Next we'll see how to relate another Model to our User, to store extra information.

Profile Models

Profile Models

As we mentioned at the start of this section, extra data can be stored for a User either by adding attributes to the User model, or by relating a different model to the user with a `OneToOneField`. Which method should you use to add extra data? Well... it depends.

Data that is more fundamental to identifying your user should be added to the `User` model itself. For example, if you needed to store an ID to tie a User back to one of your company's other systems, and it should always be present, add it as a new field on a custom User model.

If you're storing data that could be optional, or is not related to identifying the user, consider a separate model. This also allows for separation of concerns. For example, a User's author's profile is probably only relevant inside a `blog` app, and not as part of authentication. So, we should so create a profile model inside the `blog` app instead of the `blango_auth` app. Furthermore, not all data is relevant to all users. In Blango, only some of our Users are authors, so it doesn't make sense to store author bios for non-authors.

Relating a profile, or similar model, back to a User is easy, just add `OneToOneField` to it. Let's add an `AuthorProfile` model now, to store a user's biography text.

Try It Out

Start by adding a new model in `blog/models.py`, called `AuthorProfile`. It should have two fields `user`, and `bio`. `user` is a `OneToOneField` to `settings.AUTH_USER_MODEL`. Add arguments `on_delete=models.CASCADE` and `related_name="profile"`. `bio` is just a `TextField`. Be sure to run `makemigrations` after adding the new model.

The full class should look like this:

```
class AuthorProfile(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE,
        related_name="profile"
    )
    bio = models.TextField()


    def __str__(self):
        return f"{self.__class__.__name__} object for
        {self.user}"
```

Next we'll need to add `AuthorProfile` to Django admin. Open `blog/admin.py`, and add `AuthorProfile` to your imports:

Open admin.py

```
from blog.models import Tag, Post, Comment, AuthorProfile
```

Then, register the `AuthorProfile` to the admin site.

```
admin.site.register(AuthorProfile)
```

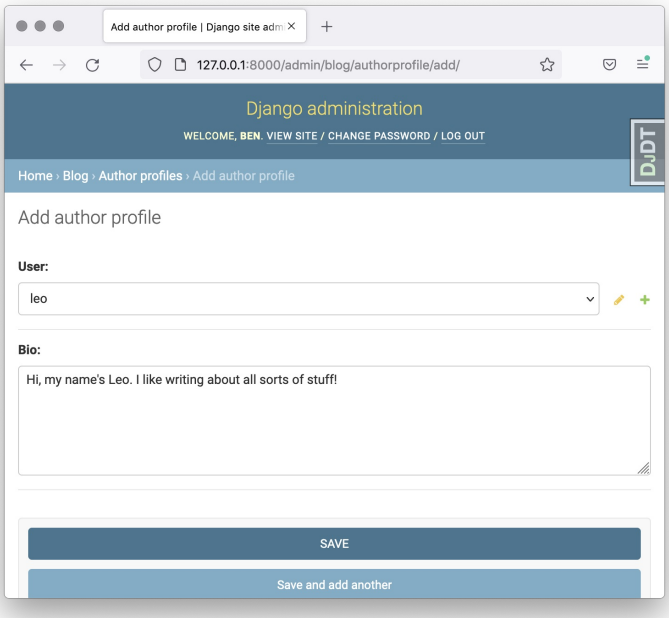
The last change is to add a section in `post-detail.html` to display the biography. Add this code just before the `blog/post-comments.html` is included.

Open post-detail.html

```
{% if post.author.profile %}
    {% row %}
        {% col %}
            <h4>About the author</h4>
            <p>{{ post.author.profile.bio }}</p>
        {% endcol %}
    {% endrow %}
{% endif %}
```

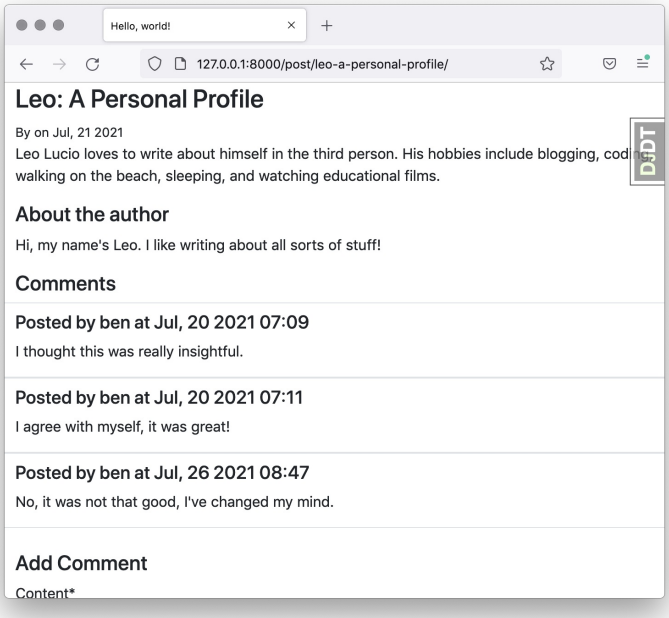
That's all that needs to be done, in terms of code. Now go to the Django Admin and add an `AuthorProfile` to one of your users.

View Blog



author profile form

Then go and view one of the posts for that author. You should see the bio you entered.



vio is now visible

As you can see adding a profile (or similar) model to a User is an easy way to store extra fields, without having to create your own User model: the method we've used would have worked even if we were using the built-in Django User.

But since we have our own User model, we'll make some changes to it to allow users to log in with their email address instead of their username.

Logging in with Email Address Part 1

Logging in with Email Address Part 1

It can be easier for users to log in with their email address to web sites – they don't need to remember their username for each site. Django allows us to set any arbitrary field on the User model as the "username" field, so you could actually use these steps as a guide to set up authentication so the user logs in with a custom `employee_id` field, or similar.

To implement email login we need to do the following:

- Create a new `UserManager` subclass, and override the `create_user()` and `create_superuser()` methods. We need to have the methods require an email address instead of a username.
- Update the `User` model to make the `email` field unique and remove the `username` field.
- Set the `User` model's `objects` attribute to an instance of the new user manager.
- Set the `User` model's `USERNAME_FIELD` to `"email"`.
- Set the `User` model's `REQUIRED_FIELDS` to an empty list. This might seem strange, because by default this value is `["email"]`, however Django assumes the `USERNAME_FIELD` is required, and so doesn't allow it to be listed in `REQUIRED_FIELDS`.
- Change the `__str__()` method of `User` to return the email address.
- Subclass `django.contrib.auth.admin.UserAdmin` and remove any reference to `username` or replace with `email`.

We'll go into these changes in more detail as we make the changes to the Blango project.

Try It Out

Before making these changes you should be sure that every user in your system has an email address, otherwise they won't be able to log in. While it's not impossible to fix this afterwards, it's easier to do it now.

In the Django Admin, go through each user and add a unique email address.

View Blog

▼ **When to implement email login**

Making this kind of change to the login system is something you should do at the start of a project, before you have production users, as it's a much bigger task to undertake then.

After you've done that, let's begin on the code change, starting in `blango_auth/models.py`. First we'll create the `UserManager` subclass, which we'll call `BlangoUserManager`. Import `UserManager` at the top of the file.

```
from django.db import models
from django.contrib.auth.models import AbstractUser, UserManager
```

Essentially we'll use `django.contrib.auth.models.UserManager` as a guide to implementing our methods, generally just replacing the use of `username` with `email`. If you'd like to try writing the class yourself, you can refer to the base class as a guide, here it is on Github. You should override the `_create_user()`, `create_user()` and `create_superuser()` methods. Make sure you write the class above the `User` class, since that will need to reference it.

You might come up with something like this:

```
class BlangoUserManager(UserManager):
    def _create_user(self, email, password, **extra_fields):
        if not email:
            raise ValueError("Email must be set")
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        extra_fields.setdefault("is_staff", False)
        extra_fields.setdefault("is_superuser", False)
        return self._create_user(email, password,
        **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault("is_staff", True)
        extra_fields.setdefault("is_superuser", True)

        if extra_fields.get("is_staff") is not True:
            raise ValueError("Superuser must have
        is_staff=True.")
        if extra_fields.get("is_superuser") is not True:
            raise ValueError("Superuser must have
        is_superuser=True.")

        return self._create_user(email, password,
        **extra_fields)
```

If yours is different, you can always copy and paste the above.

Next the changes to the `User` class. Make these changes:

- The `username` field should be set to `None`.
- The `email` field should be the same on `AbstractUser`, except it should be unique and required.
- Set the `objects` attribute to an instance of `BlangoUserManager`
- Update `USERNAME_FIELD`, `REQUIRED_FIELDS` and the `__str__()` method as we described earlier.

Remember, the `User` class should come *after* the `BlangoUserManager` class. You should come up with something like this:

```
class User(AbstractUser):
    username = None
    email = models.EmailField(
        _("email address"),
        unique=True,
    )

    objects = BlangoUserManager()

    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = []

    def __str__(self):
        return self.email
```

Notice that we're applying a translation to the "email address" text, the same as the base class does, by wrapping it in the `django.utils.translation.gettext_lazy` function. As a shortcut, this function is imported as _, so make sure to add this import:

```
from django.utils.translation import gettext_lazy as _
```

Logging in with Email Address Part 2

Logging in with Email Address Part 2

The last thing to do is build a custom user `UserAdmin` subclass (`BlangoUserAdmin`) in `blango_auth/admin.py`. Once again, you can refer to the base class if you like, here it is on Github. You should override the `fieldsets`, `add_fieldsets`, `list_display`, `search_fields` and `ordering` attributes and replace `username` with `email`, and/or remove `username`, where appropriate.

Open blango_auth/admin.py

The class should end up looking like this:

```
class BlangoUserAdmin(UserAdmin):
    fieldsets = (
        (None, {"fields": ("email", "password")}),
        (_("Personal info"), {"fields": ("first_name",
        "last_name")}),
        (
            _("Permissions"),
            {
                "fields": (
                    "is_active",
                    "is_staff",
                    "is_superuser",
                    "groups",
                    "user_permissions",
                )
            },
        ),
        (_("Important dates"), {"fields": ("last_login",
        "date_joined")}),
    )
    add_fieldsets = (
        (
            None,
            {
                "classes": ("wide",),
                "fields": ("email", "password1", "password2"),
            },
        ),
    )
    list_display = ("email", "first_name", "last_name",
        "is_staff")
    search_fields = ("email", "first_name", "last_name")
    ordering = ("email",)
```

Note the use of the _ (`gettext_lazy`) function throughout, so make sure it's imported.

```
from django.utils.translation import gettext_lazy as _
```

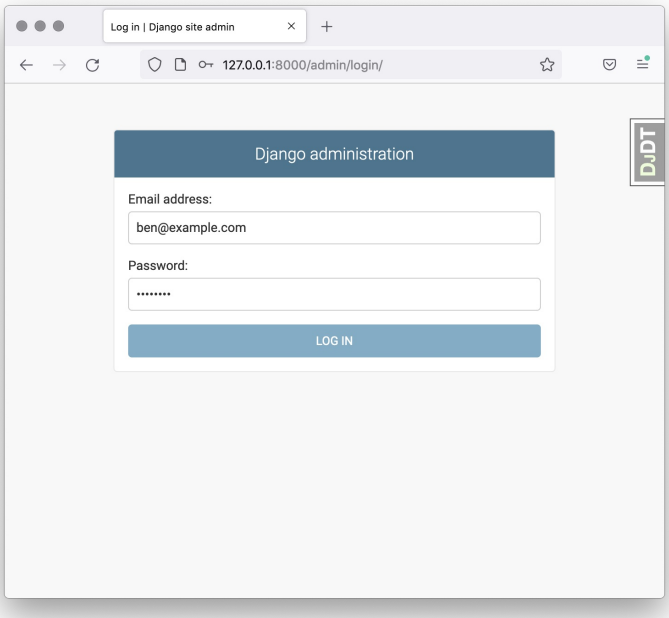
The last change to the file is to register `BlangoUserAdmin` as the admin class for `User`. Change the line of code that says `admin.site.register(User, UserAdmin` to:

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

Finally, you'll need to run `makemigrations` and `migrate` to change the database table.

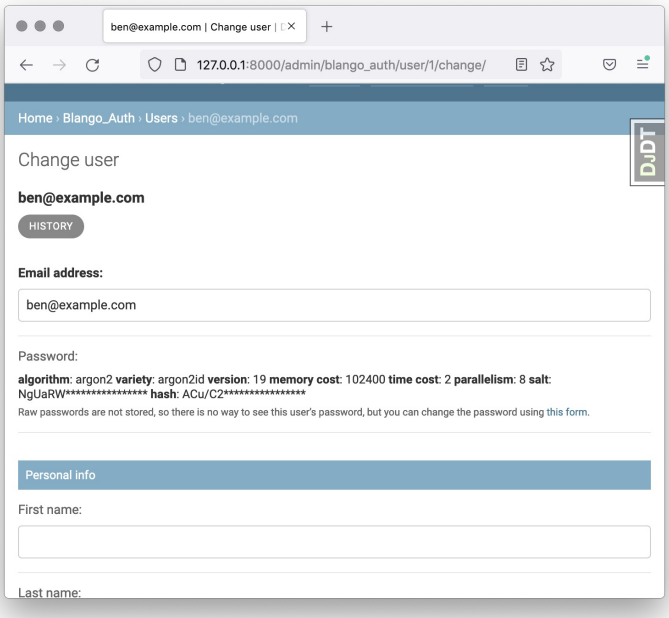
All done! Now you can test it out. Visit the Django Admin in a browser, and log in – this time you'll need to use your email address and password.

View Blog



email login

Try looking at the details for a user, and you'll notice no username anywhere.



user form with email

You can validate that all your changes are working correctly by adding and editing a user through the Django Admin, and also creating a superuser with the `createsuperuser` command. Everything should work fine.

That's the end of this section. We saw how to add a custom user model, and why it's important to make these kinds of authentication decisions at the start of a project, to save lots of extra work. We also learned about when to prefer an external profile model to extra fields on the user class.

One thing that's been missing from Blango is the ability for users to register for the site. We'll fix that up in the next section.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish custom User model"
```

- Push to GitHub:

```
git push
```