

Extending the existing User model VS Custom User Model in Django

Chapter 7

1

Extending the existing User model

- 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)
```

2

Accessing the related information

- 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
```
- 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 are not automatically created when a user is created.
- 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.

3

Adding a profile model's fields to the user page in the admin

- 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.
- Django provides two subclasses of InlineModelAdmin and they are:
 - TabularInline
 - StackedInline

```
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)
```

4

stackedinline vs tabularinline

- Basically, both allow you to edit models on the same page as a parent model.
- Allow to edit a certain model while editing another one instead of having to manually add another instance somewhere else in your interface.
- The difference between these two is merely the Layout used to render them.

The screenshot shows two Django administration forms side-by-side. The top form is titled 'Add user' and contains fields for 'Username', 'Password', and 'Password confirmation'. The bottom form is titled 'EMPLOYEE' and contains fields for 'Employee: #1' and 'Department'. The forms are rendered using the 'stackedinline' layout, allowing them to be edited on the same page as a parent model.

Custom User Model

Using a custom user model when starting a project

- In the previous chapter, we have used the default User Model provided by Django.
- What if all the provided attributes that the User model provides isn't enough?
- Say, we want to include an additional attribute for each user account, e.g. an age field.
- This is achieved through the creation of an additional model in `models.py` file and updating `settings.py` to tell Django to use the new custom user model in place of the built-in User model.

7

Creating our custom user model

Creating our custom user model requires the following steps:

1. Create a new CustomUser model (`models.py`);
2. Update `settings.py` to tell Django to use the new custom user model in place of the built-in User model;
3. Update `admin.py` to use the new CustomUser model;
4. Create a migration record for the model and migrate the change into our database to create a new database that uses the custom user model;
5. Create new forms for UserCreation and UserChangeForm (`forms.py`);
6. Update `views.py` to use the new forms created in step 5.

8

Creating our custom user model – Step 1: Create a new CustomUser model

Edit `models.py` to create a database model called `CustomUser`.

- We added our first extra field for the “age” of our users.
- We used Django’s `PositiveIntegerField` which means the integer must be either positive or zero.
- We extend `AbstractUser`, so our `CustomUser` is basically a copy of the default User model. The only update is our new age field.

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

# Create your models here

class CustomUser (AbstractUser):    # add the model to extend the AbstractUser
    age = models.PositiveIntegerField(default=0)
```

9

Creating our custom user model – Step 2: Update settings.py

Update `settings.py` to tell Django to use the new custom user model in place of the built-in User model.

- At the bottom of `settings.py`, add the following line to use `CustomUser` that we have created in `models.py` in the previous step:

`AUTH_USER_MODEL = 'yourAppNameHere.CustomUser'`

10

Creating our custom user model – Step 3: Update admin.py

Since Django Admin is tightly coupled to the default User model, we will extend the existing `UserAdmin` class (line 7 below) to use our new `CustomUser` model (line 13 below). Then, add the extra custom fields to `fieldsets` (line 8 below) in order to be able to add and edit them in the Django admin.

```
1 # users/admin.py
2 from django.contrib import admin
3 from django.contrib.auth.admin import UserAdmin
4
5 # Register your models here
6 from .models import CustomUser
7 class CustomUserAdmin(UserAdmin):
8     fieldsets = UserAdmin.fieldsets + (
9         (None, {'fields': ('age',)}),
10    )
11
12    list_display = ['email', 'username', 'age']
13    model = CustomUser
14
15 admin.site.register(CustomUser, CustomUserAdmin)
```

Customizing lists – list_display:

By default, the list displays the result of `__str__()` of the object. To add other fields to the list to display, define a `UserAdmin` class for the model.

`fieldsets` now contain both the attributes of the default User model and our added custom attributes of the `CustomUser` model in the “add” and “edit” user page of the Django Admin.

`list_display` is used to customize which fields to be displayed in Django Admin page.

11

Django admin – add user

- As you can see, the add user only shows username and password.

Django administration

Home > Users > Users > Add user

ARTICLES

Articles + Add

AUTHENTICATION AND AUTHORIZATION

Groups + Add

USERS

Users + Add

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation:

Enter the same password as before, for verification.

12

Django admin – Change user

- The Change user interface is governed by our fieldsets in admin.py
- Here, you will see that Age has been added at the bottom after the attributes of the default user model

Result of fieldsets = UserAdmin.fieldsets + ((None, ('fields': ('age',))),)

Django administration

Home · Users · Users · test999

ARTICLES

Articles [+ Add](#)

AUTHENTICATION AND AUTHORIZATION

Groups [+ Add](#)

USERS

Users [+ Add](#)

Change user

test999

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

Raw passwords are not stored, so there is no way to see this user's password, but...

Personal info

First name:

Last name:

Email address:

Important dates

Last login: Date: Today Time: Now

Date joined: Date: Today Time: Now

Age:

Django admin – list_display

Django administration

Home · Users · Users

ARTICLES

Articles [+ Add](#)

AUTHENTICATION AND AUTHORIZATION

Groups [+ Add](#)

USERS

Users [+ Add](#)

Select user to change

Q Search

Action: Go 0 of 8 selected

<input type="checkbox"/>	USERNAME	EMAIL ADDRESS	AGE
<input type="checkbox"/>	test11	test11@123.com	0
<input type="checkbox"/>	test12	test12@123.com	0

- According to list_display that we have used in admin.py, we see 'username', 'email' and 'age' being displayed.

Step 4: Create a migration record and migrate the change that uses the custom user model

- It is NOT recommend to run migrate on new projects until *after* a custom user model has been configured.
- Otherwise Django will bind the database to the built-in User model which is difficult to modify later on in the project.
- What is the difference of running the following 2 sets of commands?

(1) `python manage.py makemigrations`
`python manage.py migrate`

(2) `python manage.py makemigrations users`
`python manage.py migrate users`

- We will **use Case (1)**: If you run the commands without the app name, then all available changes will be applied, including the project level and all apps.
 - Remember that it is the first time we run migrate, the migrations for the built-in project-level “auth”, “admin” have not yet been executed.
 - Using this option will create them as well, together with the model class “CustomUser” we have built in models.py

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying sessions.0001_initial... OK
```

15

Creating our custom user model – forms.py

Step 5: Create new forms for UserCreation

In the previous chapter, for (**SignUp – (4)** Write the logic for the view SignUpView), we used Django's built-in form class, `UserCreationForm`, to build the signup page to register new users easily.

Now, we need to create forms.py to add the forms to interact with our new `CustomUser` model, for the cases to add a new user and edit user information. One scenario is when a user signs up for a new account on our website.

- So we need to update the two built-in forms for this functionality: `UserCreationForm` and `UserChangeForm` that are tied to `User`.

- For both forms we are setting the model to our `CustomUser` and using the default fields by using `Meta.fields`.
- Our `CustomUser` model contains all the fields of the default `User` model and our additional age field which we set.

```
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from .models import CustomUser

class CustomUserCreationForm(UserCreationForm):
    class Meta:
        model = CustomUser
        fields = UserCreationForm.Meta.fields

class CustomUserChangeForm(UserChangeForm):
    class Meta:
        model = CustomUser
        fields = UserChangeForm.Meta.fields
```

16

Creating our custom user model – forms.py

Step 5: Create new forms for UserCreation

- Under fields we're using Meta.fields which just displays the default settings of username/password.
- We can explicitly set which fields to be displayed, so let's update it to ask for a username/email/password by setting it to ('username', 'email',). We don't need to include the password field because it's required.

```
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from .models import CustomUser

class CustomUserCreationForm(UserCreationForm):
    class Meta:
        model = CustomUser
        fields = UserCreationForm.Meta.fields

class CustomUserChangeForm(UserChangeForm):
    class Meta:
        model = CustomUser
        fields = UserChangeForm.Meta.fields
```

```
class CustomUserCreationForm(UserCreationForm):
    class Meta(UserCreationForm.Meta):
        model = CustomUser
        fields = ('username', 'email', ) # new

class CustomUserChangeForm(UserChangeForm):
    class Meta:
        model = CustomUser
        fields = ('username', 'email', ) # new
```

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Email address:

Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

Creating our custom user model –

Step 6: Update views.py for SignUp

Update views.py to use the new forms created in step 5.

```
from django.contrib.auth.forms import UserCreationForm
from django.urls import reverse_lazy
from django.views import generic

class SignUpView(generic.CreateView):
    form_class = UserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'signup.html'
```

Example of view from SignUp with the built-in User Model

```
from django.urls import reverse_lazy
from django.views import generic

from .forms import CustomUserCreationForm

class SignUp(generic.CreateView):
    form_class = CustomUserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'signup.html'
```

Changing to a custom user model mid-project

- Changing `AUTH_USER_MODEL` after you have created database tables is significantly more difficult since it affects foreign keys and many-to-many relationships, for example.
- This change cannot be done automatically and requires manually fixing your schema, moving your data from the old user table, and possibly manually reapplying some migrations.
- 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.

19

Tips on Custom User Model

- 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)
```

20

Referencing the Custom User model

- If you reference `User` directly (for example, by referring to it in a foreign key), your code will not work in projects where the `AUTH_USER_MODEL` setting has been changed to a different user model.
- 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,
    )
```

21

Summary

- Extending the existing User model
- Customer User Model with **additional User attributes**

22