Options & Objects: Customizing the Django User Model

Eleanor Stribling · Follow

Published in agatha · 12 min read · Jun 6, 2017

Not long ago, I began working on a web application using the Django web framework where I wanted to sign users in using their phone number instead of the default username. In the process of figuring out the best way to do that, I found a lot of confusing or outdated documentation, which

prompted me to propose a poster session on this topic for PyCon 2017 (and it got in!).

Unfortunately, aside from the awesomeness of my robot collage depicted in the tweet on the left, my poster didn't make the trip home with me to San Francisco. I was also asked by some of the people who stopped by if I had examples of all three to point to someplace, which I didn't at the time. Live and learn.

Atul Varma
@toolness · **Follow**

by far the best poster at the @pycon poster session!

> **Eleanor Stribling** @eleanorstrib
>
> My poster for #pycon mixes #django User model customization with whimsical robots. Just because.

12:19 AM · May 22, 2017

1 Reply Share

Read more on X

Not found

I got some nice compliments on the poster, in person and shown in the tweets on the left, but I thought full examples were a great piece of feedback, and would lead to a broader discussion of the nuances and tradeoffs of each option. In this post, I'll outline what I learned putting together the poster, show some sample code for each possibility, and offer up some recommendations for the s*ign-in-with-something-other-than-username* use case based on my research and on the conversations I had at PyCon.

Orientation

In this Github repo, you'll find a basic, functional project with a simple implementation of each of the three options outlined on the poster. Links to the relevant files for each option inside the repo are included below for easy reference. In each, I've included a separate project and application for each option, all with a working template for front end authentication as well as a working superuser creation method that can be run in the command line.

All of the examples use Django 1.11 and Python 3.5.2. Each one adds a `zip_code` field to the User model as a `CharField` to keep it simple, and in the first two options, I make the `email` field the `username` .

Here I would like to pause and say that if you haven't used Django, I recommend going through one or both of the excellent tutorials from DjangoGirls or the Django docs to get familiar with the framework before trying to follow this post. If you have worked with Django, you're probably familiar with the User model, an out-of-the-box feature that you can make use of to authenticate users (a.k.a. create accounts and sign them in and out of your application) via a Python class that represents a table in the database and comes with a bunch of built in fields and methods for data retrieval and permissions that you can read about in the official docs.

Why this matters beyond the use case

Getting users to sign up for your product often isn't easy or free, and when you finally get them through the funnel to the form, you want to balance

If you're using Django's authentication features, you have three options for customizing your model, shown clockwise on the poster, starting on the top left:

Option 1: Create your own user model by subclassing Django's AbstractBaseUser. This is great if you need a lot of flexibility in the content of the model, and how it is accessed and you don't mind a little extra work at the start of your project.

Option 2: Extend the existing User Model with your own fields by subclassing Django's AbstractUser. This option is ideal if you want to keep the default fields in the user model and you don't need anything too different with

permissions, but you want to change the username to a different field and/or add authentication fields.

Option 3: Add fields to the existing User class using the OneToOneField method. This method doesn't really help if you want to change the username field, but it does enable you to link the User model to another model to store more data. It daisy chains two models together in the database, which would be useful if you want to store user data that doesn't have anything to do with authentication or if some of the user's data is needed by a third party service that you want to cordon off e. This was also the recommended way to customize a User model before version 1.5 Django, so quite a few existing applications use this method.

Of these three, the first two are really solid options for most use cases. At PyCon, I spoke to a few people who were interested in using options 1 or 2 but had Option 3 in place. In those cases, because you're making fundamental changes to the database schema, the only course of action is to back up your data and recreate your models and database. This post is a couple of years old, but contains some solid advice.

This post won't cover implementation of third party authentication services with Django, steps for migrating existing models or sending emails to users. That said, sending emails to authenticate new users with Django will be the subject of a future post!

Let's get started.

Three Options, Step by Step

Option 1 — Subclassing AbstractBaseUser (repo)

This is the highest difficulty, highest flexibility option.

Use this approach if you:

- Want a lot of latitude to customize everything about your model
- Want to minimize the number of fields for data collection and you don't think you'll use the ones that come out of the box
- Have some time to put into writing some of the basic permissions you need to work with Django's permissions model

The first part, covered in the poster, is writing your custom user model itself.

In your application folder, go to the `models.py` folder and add code along the lines of the following (with whatever fields you want, of course!):

models.py

```
1  from django.db import models
2  from django.contrib.auth.models import AbstractBaseUser
3
4
5  class CustomUser(AbstractBaseUser):
6      email = models.EmailField(unique=True)
7      zip_code = models.CharField(max_length=6)
8      name = models.CharField(max_length=30)
9      is_staff = models.BooleanField(default=False)
10     REQUIRED_FIELDS = ['zip_code']
11     USERNAME_FIELD = 'email'
12
13     def get_short_name(self):
14         return self.email
15
16     def __str__(self):
17         return self.email
```

There is a lot going on in this one, so let's break it down a bit.

- We need to add the `unique=True` parameter to whatever field we are using as the `USERNAME_FIELD` on line 6.
- `REQUIRED_FIELDS` is a list of fields that will be mandatory to create a user. Note that including the `USERNAME_FIELD` here will give you an error when you try to run your app. Because I'll create my form by importing and sublassing the `UserCreationForm` in my `forms.py` file, `password` doesn't need to be added to the `REQUIRED_FIELDS` list.

Next, visit the `admin.py` file in your application folder and add the following to register your model:

admin.py

```
1  from django.contrib import admin
2  from abstract_base_user_sample import CustomUser
3
4  admin.site.register('CustomUser')
```

Then add this line to say that `CustomUser` is what you'll use to authenticate people.

```
AUTH_USER_MODEL = 'abstract_base_user_sample.CustomUser'
```

Then migrate your database from your project root:

```
python manage.py makemigrations
```

```
python manage.py migrate
```

So, again, this was as far as my poster went. However, with great flexibility comes greater programming overhead on set up!

If you try to run an app with the code as it is now on your localhost, it will ostensibly work but you won't be able to access the admin or add a user. The reason? You need to explicitly define this functionality via a user manager, and that's where our `CustomUserManager` class comes in.

To create this functionality, you can import `BaseUserManager` from `django.contrib.auth.models` as shown on line 2, then subclass it as seen on line 4 when a `CustomAccountManager` class is created. You'll need to define

methods for `create_user` , `create_superuser` , and `get_by_natural_key` at a minimum, or you will encounter errors. If you want to create more specific permissions groups, this is where you should define those as well.

models.py

```
1  from django.db import models
2  from django.contrib.auth.models import PermissionsMixin, AbstractBaseUser, BaseUserManager
3
4  class CustomAccountManager(BaseUserManager):
5      def create_user(self, email, zip_code, password):
6          user = self.model(email=email, zip_code=zip_code, password=password)
7          user.set_password(password)
8          user.is_staff = False
9          user.is_superuser = False
10         user.save(using=self._db)
11         return user
12
13     def create_superuser(self, email, zip_code, password):
14         user = self.create_user(email=email, zip_code=zip_code, password=password)
15         user.is_active = True
16         user.is_staff = True
17         user.is_superuser = True
18         user.save(using=self._db)
19         return user
20
21     def get_by_natural_key(self, email_):
22         print(email_)
23         return self.get(email=email_)
```

There are some other values you will have to set as well to make the user model work:

- `user.is_staff` as shown on lines 8 & 16 → this controls access to the admin site.
- `user.is_superuser` as shown on lines 9 & 17 → when True, the user has all available permissions
- `user.save()` is saving the data from the form to the database

The `get_by_natural_key` method on line 21 should be set to whatever the id credential will be, effectively the `username`, or whatever you're replacing that value with — in our use case, this is email.

But we're not quite done yet! We need to make some modifications to our `CustomUser`.

You might have noticed on line 2 in the screenshot above, we imported something called `PermissionsMixin`; this is a really helpful model that provides you with the methods you need to work with the Django permissions module with minimal pain and suffering. Make sure you import this before the `AbstractBaseUser` and `BaseUserManager` or you will get errors. Then, add it to the arguments for your `CustomUser` model.

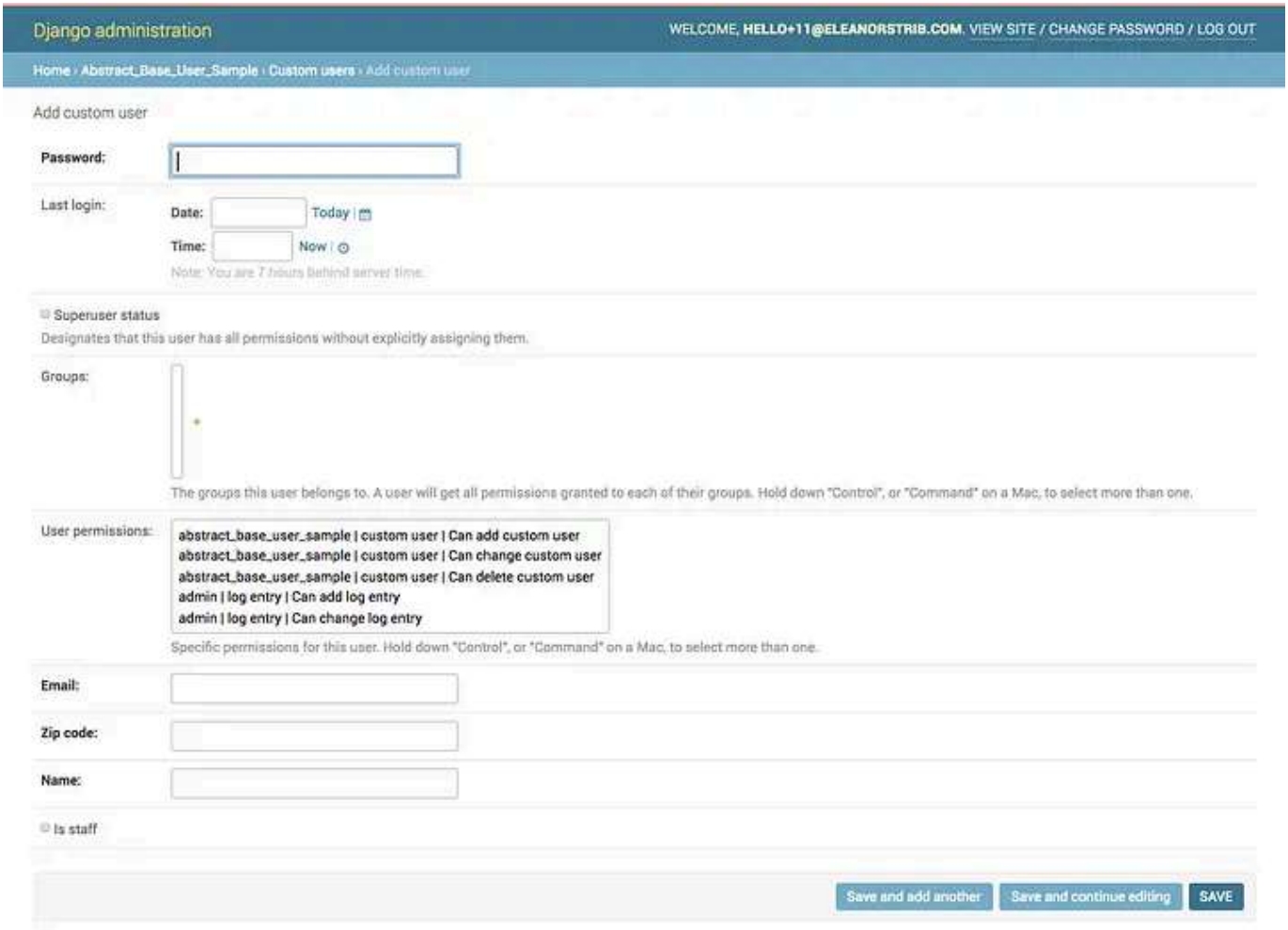
```
26  class CustomUser(AbstractBaseUser, PermissionsMixin):
27      email = models.EmailField(unique=True)
28      zip_code = models.CharField(max_length=6)
29      name = models.CharField(max_length=30)
30      is_staff = models.BooleanField(default=False)
31      REQUIRED_FIELDS = ['zip_code', 'password']
32      USERNAME_FIELD = 'email'
33
34      objects = CustomAccountManager()
35
36      def get_short_name(self):
37          return self.email
38
39      def natural_key(self):
40          return self.email
41
42      def __str__(self):
43          return self.email
```

Note that we also need to add a `get_short_name` method, as Django expects a short name to be available for each user. This can be whatever field you want.

In addition, I've added a `natural_key` field which returns the email — as above, the credential the user needs to sign in — and a simple method to return the email as the string to represent the account.

Finally, Instantiation of the `CustomAccountManager` object on line 34 that we created a moment ago will enable us to access the methods we wrote for managing permissions and accessing data when we created the `CustomAccountManager` class.

Don't forget to migrate your database when you've made all of these changes.



When you create a superuser (in terminal, `python manage.py createsuperuser`), the screenshot on the left is what you'll see when you sign and sign into the admin and create a user. Notice that we don't have many of the standard Django user fields like `first_name`, `last_name`, etc.

Option 2 — Subclassing AbstractUser (repo)

This is a great option if you want to:

- Use default fields from the Django User model
- Control what the `username` variable will be with minimal overhead

- Skip creating a custom user manager and leverage methods built into Django

However, this option does have some limitations around customizing the `username`, which we will see in a minute.

The first step is to write your model.

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

class CustomUser(AbstractUser):
    email = models.EmailField(unique=True)
    zip_code = models.CharField(max_length=6)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['zip_code']

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

Notice that we are importing and subclassing `AbstractUser` this time, but we don't need the `is_staff` field because we're effectively adding to the default User model which already has this attribute.

The `USERNAME_FIELD` resets the username to email, so that is required on sign in. The model already has an email field, but it needs to be added here as well to add the `unique = True` parameter . Another important note: the username field and password will be required, so they don't need to go into the `REQUIRED FIELDS` list — in fact if you reset the `USERNAME_FIELD` the app will throw an error.

Add the `AUTH_USER_MODEL` variable to the project's `settings.py` file, so the application knows what Users should look like.

```
60  AUTH_USER_MODEL = 'abstract_user_sample.CustomUser'
```

Register your model in the project's `admin.py` file to make it visible in the admin screen.

admin.py

```
1  from django.contrib import admin
2  from abstract_user_sample.models import CustomUser
3
4  admin.site.register(CustomUser)
```

At this point, remember to run your migrations:

- `python manage.py makemigrations`
- `python manage.py migrate`

Then create a superuser (`python manage.py createsuperuser`) to test it. This is where you run into a problem if you want to use `email` as the username but not require an actual `username` when the person signs up: the source code for the `AbstractUser` model *requires* `email`, `password` AND `username`.

This means that:

- If you don't add username to `REQUIRED_FIELDS` when creating a superuser, you'll get an error along the lines of `TypeError: create_superuser()`

`missing 1 required positional argument: 'username'` because the `AbstractUser` model is expecting that argument, even though you've set `USERNAME_FIELD = 'email'`

- If you try to create a user via a form in the browser that doesn't include username, you'll get an error like this: `NOT NULL constraint failed: abstract_user_sample_customuser.username`

There are a couple of options here, depending on your requirements:

1. Use Option 1 and subclass `AbstractBaseUser` for your custom model instead. It does require more code, but it would be cleaner in some ways — for example, you could skip having a username entirely if it doesn't make sense for your application
2. Override the username field parameters in your `models.py` file, the same way you overrode the email parameters if you want to keep the field but don't want to require it when a new user is created. Here's a bare bones example that will wipe out the helper text, error message, etc. that are out of the box, but enables that field to be blank or set to null.

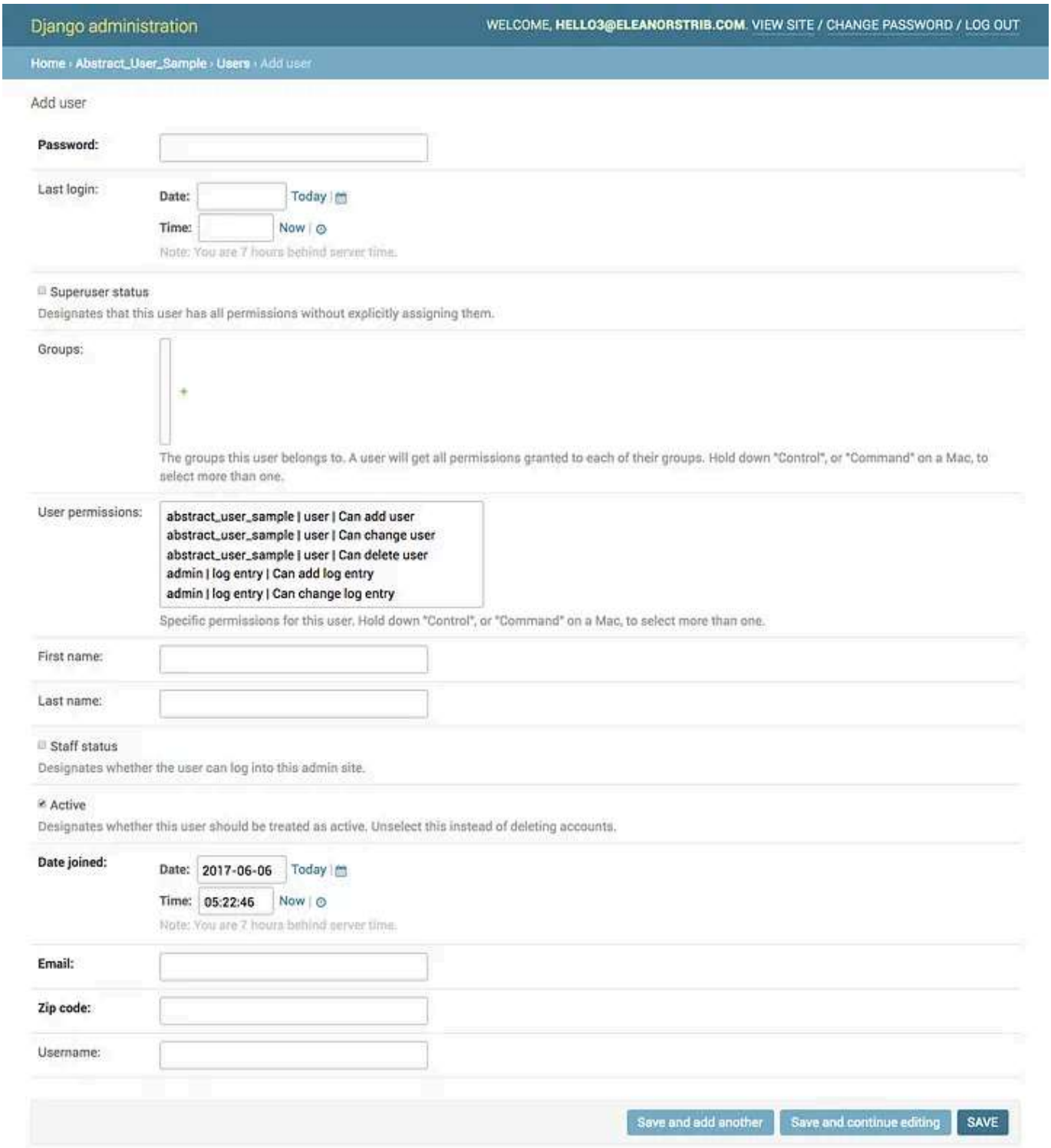
models.py

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

class CustomUser(AbstractUser):
    email = models.EmailField(unique=True)
    zip_code = models.CharField(max_length=6)
    username = models.CharField(blank=True, null=True, max_length=150)
    REQUIRED_FIELDS = ['zip_code', 'username']
    USERNAME_FIELD = 'email'

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

Because we've added `username` to `REQUIRED_FIELDS` on line 8, it will show up when you create a superuser in the command line, however you can specify fields in the forms.py file, so if you really don't want end users adding a username on sign up, you can drop that completely and they can still complete the process, albeit with a blank username field in the database.

Make sure to run your migrations before testing!

When you sign into the admin panel and create a new user, you'll see that — unlike in Option 1 — the default Django fields appear that weren't explicitly in the model, as well as the `zip_code` field we added. Notice that `Username` appears but is not required.



Option 3 — The OneToOneField (repo)

This option doesn't really match the original use case — to sign the user in with the email as a password — but it would make a lot of sense if you

wanted to store user data that isn't directly related to authentication in a way that's still connected to the user model.

It's pretty straightforward to implement. In the `models.py` file in your app, you'll need to import the User model (shown on line 2) and then link it to the model with the `OneToOneField` method on line 5.

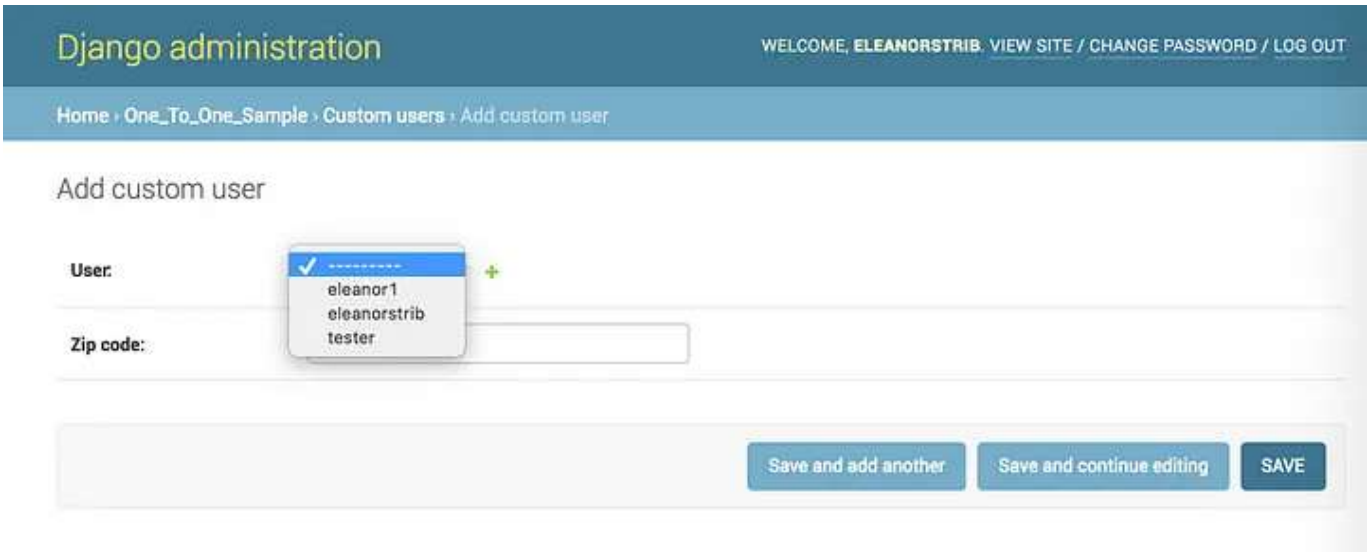
models.py

```
1  from django.db import models
2  from django.contrib.auth.models import User
3
4  class CustomUser(models.Model):
5      user = models.OneToOneField(User, on_delete=models.CASCADE)
6      zip_code = models.CharField(max_length=6)
7
8      def __str__(self):
9          return self.user.username
```

In this case, you don't need to add anything to the `settings.py` file because you are still using the Django User model, however you do still need to register your model in `admin.py` as we saw in the other two examples.

The way that the `User` model and our `CustomUser` model are connected is very clear in the Admin. When you look at the admin for this version, you'll see that Users and CustomUsers are separately listed.

Creating a user is the same as usual, but if you try to add a custom user, the admin will prompt you for the username of the record you want to attach our custom model to, and displays the `zip_code` field we added.



Resources for this post

- Official Django Documentation on customizing the User Model and source code for the auth models
- Two Scoops of Django

- Simple is Better than Complex —How to Extend the Django User Model

Summing up

While this post was long on detail (and thanks for sticking with it!), Django does have some pretty great features that can be confusing the first time you use them, so I hope this post was useful to you!

If you have any comments on this post, please leave them in the comments. Questions or requests on the examples can be added as Github issues.

Django | Authentication | Software Development | Python | Web Development

Written by Eleanor Stribling

1.2K Followers · Editor for agatha

Product & people manager, writer. Group PM @ Google, frmr TubeMogul (now Adobe), Microsoft, & Zendesk. MIT MBA. Building productmavens.io.