How to Extend Django User Model

By **Vitor Freitas**

I'm a passionate software developer and researcher from Brazil, currently living in Finland. I write about Python, Django and Web Development on a weekly basis. Read more.

The Django's built-in authentication system is great. For the most part we can use it out-of-the-box, saving a lot of development and testing effort. It fits most of the use cases and is very safe. But sometimes we need to do some fine adjustment so to fit our Web application.

Commonly we want to store a few more data related to our User. If your Web application have an social appeal, you might want to store a short bio, the location of the user, and other things like that.

In this tutorial I will present the strategies you can use to

simply extend the default Django User Model, so you don't need to implement everything from scratch.

Ways to Extend the Existing User Model

Generally speaking, there are four different ways to extend the existing User model. Read below why and when to use them.

Option 1: Using a Proxy Model

What is a Proxy Model?
It is a model inheritance without creating a new table in the database. It is used to change the behaviour of an existing model (e.g. default ordering, add new methods, etc.) without affecting the existing database schema.

When should I use a Proxy Model?
You should use a Proxy Model to extend the existing User model when you don't need to store extra information in the database, but simply add extra methods or change the model's query Manager.

That's what I need! Take me to the instructions.

Option 2: Using One-To-One Link With a User Model (Profile)

What is a One-To-One Link?
It is a regular Django model that's gonna have it's own database table and will hold a One-To-One relationship

with the existing User Model through a `OneToOneField`.

When should I use a One-To-One Link?
You should use a One-To-One Link when you need to store extra information about the existing User Model that's not related to the authentication process. We usually call it a User Profile.

That's what I need! Take me to the instructions.

Option 3: Creating a Custom User Model Extending AbstractBaseUser

What is a Custom User Model Extending AbstractBaseUser?
It is an entirely new User model that inherit from `AbstractBaseUser`. It requires a special care and to update some references through the `settings.py`. Ideally it should be done in the begining of the project, since it will dramatically impact the database schema. Extra care while implementing it.

When should I use a Custom User Model Extending AbstractBaseUser?
You should use a Custom User Model when your application have specific requirements in relation to the authentication process. For example, in some cases it makes more sense to use an email address as your identification token instead of a username.

That's what I need! Take me to the instructions.

Option 4: Creating a Custom User Model Extending AbstractUser

What is a Custom User Model Extending AbstractUser?

It is a new User model that inherit from `AbstractUser`. It requires a special care and to update some references through the `settings.py`. Ideally it should be done in the begining of the project, since it will dramatically impact the database schema. Extra care while implementing it.

When should I use a Custom User Model Extending AbstractUser?

You should use it when you are perfectly happy with how Django handles the authentication process and you wouldn't change anything on it. Yet, you want to add some extra information directly in the User model, without having to create an extra class (like in the **Option 2**).

That's what I need! Take me to the instructions.

Extending User Model Using a Proxy Model

This is the less intrusive way to extend the existing User model. You won't have any drawbacks with that strategy. But it is very limited in many ways.

Here is how you do it:

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

```
class Person(User):
    objects = PersonManager()

    class Meta:
        proxy = True
        ordering = ('first_name', )

    def do_something(self):
        ...
```

In the example above we have defined a Proxy Model named `Person`. We tell Django this is a Proxy Model by adding the following property inside the Meta class: `proxy = True`.

In this case I've redefined the default ordering, assigned a custom `Manager` to the model, and also defined a new method `do_something`.

It is worth noting that `User.objects.all()` and `Person.objects.all()` will query the same database table. The only difference is in the behavior we define for the Proxy Model.

If that's all you need, go for it. Keep it simple.

Extending User Model Using a One-To-One Link

There is a good chance that this is what you want. Personally that is the method I use for the most part. We will be creating a new Django Model to store the extra information that relates to the User Model.

Bear in mind that using this strategy results in additional queries or joins to retrieve the related data. Basically all the time you access an related data, Django will fire an additional query. But this can be avoided for the most cases. I will get back to that later on.

I usually name the Django Model as `Profile`:

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

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(max_length=500, blank=True)
    location = models.CharField(max_length=30, blank=True)
    birth_date = models.DateField(null=True, blank=True)
```

Now this is where the magic happens: we will now define **signals** so our `Profile` model will be automatically created/updated when we create/update User instances.

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(max_length=500, blank=True)
    location = models.CharField(max_length=30, blank=True)
    birth_date = models.DateField(null=True, blank=True)

@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)

@receiver(post_save, sender=User)
```

```
def save_user_profile(sender, instance, **kwargs):
    instance.profile.save()
```

Basically we are hooking the `create_user_profile` and `save_user_profile` methods to the User model, whenever a **save** event occurs. This kind of signal is called `post_save`.

Great stuff. *Now, tell me how can I use it*.

Piece of cake. Check this example in a Django Template:

```
<h2>{{ user.get_full_name }}</h2>
<ul>
  <li>Username: {{ user.username }}</li>
  <li>Location: {{ user.profile.location }}</li>
  <li>Birth Date: {{ user.profile.birth_date }}</li>
</ul>
```

How about inside a view method?

```
def update_profile(request, user_id):
    user = User.objects.get(pk=user_id)
    user.profile.bio = 'Lorem ipsum dolor sit amet, consectetur adipisicing elit...
    user.save()
```

Generally speaking, you will never have to call the Profile's save method. Everything is done through the User model.

What if I'm using Django Forms?

Did you know that you can process more than one form at once? Check out this snippet:

forms.py

```
class UserForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ('first_name', 'last_name', 'email')

class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ('url', 'location', 'company')
```

views.py

```
@login_required
@transaction.atomic
def update_profile(request):
    if request.method == 'POST':
        user_form = UserForm(request.POST, instance=request.user)
        profile_form = ProfileForm(request.POST, instance=request.user.profile)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
            messages.success(request, _('Your profile was successfully updated!'))
            return redirect('settings:profile')
        else:
            messages.error(request, _('Please correct the error below.'))
    else:
        user_form = UserForm(instance=request.user)
        profile_form = ProfileForm(instance=request.user.profile)
    return render(request, 'profiles/profile.html', {
        'user_form': user_form,
        'profile_form': profile_form
    })
```

profile.html

```
<form method="post">
  {% csrf_token %}
  {{ user_form.as_p }}
  {{ profile_form.as_p }}
  <button type="submit">Save changes</button>
</form>
```

And the extra database queries you were talking about?

Oh, right. I've addressed this issue in another post named "Optimize Database Queries". You can read it clicking here.

But, long story short: Django relationships are lazy. Meaning Django will only query the database if you access one of the related properties. Sometimes it causes some undesired effects, like firing hundreds or thousands of queries. This problem can be mitigated using the `select_related` method.

Knowing beforehand you will need to access a related data, you can prefetch it in a single database query:

```
users = User.objects.all().select_related('profile')
```

Extending User Model Using a Custom Model Extending AbstractBaseUser

The hairy one. Well, honestly I try to avoid it at all costs. But sometimes you can't run from it. And it is perfectly fine. There is hardly such a thing as best or worst solution. For the most part there is a more or less appropriate solution. If this is the most appropriate solution for you case, go ahead.

I had to do it once. Honestly I don't know if this is the cleaner way to do it, but, here goes nothing:

I needed to use email address as auth token and in the scenario the `username` was completly useless for me. Also there was no need for the `is_staff` flag, as I wasn't using the Django Admin.

Here is how I defined my own user model:

```
from __future__ import unicode_literals

from django.db import models
from django.core.mail import send_mail
from django.contrib.auth.models import PermissionsMixin
from django.contrib.auth.base_user import AbstractBaseUser
from django.utils.translation import ugettext_lazy as _

from .managers import UserManager


class User(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(_('email address'), unique=True)
    first_name = models.CharField(_('first name'), max_length=30, blank=True)
    last_name = models.CharField(_('last name'), max_length=30, blank=True)
    date_joined = models.DateTimeField(_('date joined'), auto_now_add=True)
    is_active = models.BooleanField(_('active'), default=True)
    avatar = models.ImageField(upload_to='avatars/', null=True, blank=True)

    objects = UserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []

    class Meta:
        verbose_name = _('user')
        verbose_name_plural = _('users')

    def get_full_name(self):
        '''
        Returns the first_name plus the last_name, with a space in between.
        '''
        full_name = '%s %s' % (self.first_name, self.last_name)
        return full_name.strip()
```

```
    def get_short_name(self):
        '''
        Returns the short name for the user.
        '''
        return self.first_name

    def email_user(self, subject, message, from_email=None, **kwargs):
        '''
        Sends an email to this User.
        '''
        send_mail(subject, message, from_email, [self.email], **kwargs)
```

I wanted to keep it as close as possible to the existing User model. Since we are inheriting from the `AbstractBaseUser` we have to follow some rules:

- **USERNAME_FIELD**: A string describing the name of the field on the User model that is used as the unique identifier. The field must be unique (i.e., have `unique=True` set in its definition);
- **REQUIRED_FIELDS**: A list of the field names that will be prompted for when creating a user via the `createsuperuser` management command;
- **is_active**: A boolean attribute that indicates whether the user is considered “active”;
- **get_full_name():** A longer formal identifier for the user. A common interpretation would be the full name of the user, but it can be any string that identifies the user.
- **get_short_name():** A short, informal identifier for the user. A common interpretation would be the first name of the user.

Okay, let's move forward. I had also to define my own `UserManager`. That's because the existing manager define the `create_user` and `create_superuser` methods.

So, here is what my `UserManager` looks like:

```
from django.contrib.auth.base_user import BaseUserManager

class UserManager(BaseUserManager):
    use_in_migrations = True

    def _create_user(self, email, password, **extra_fields):
        """
        Creates and saves a User with the given email and password.
        """
        if not email:
            raise ValueError('The given 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_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_superuser', 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)
```

Basically I've done a clean up of the existing `UserManager`, removing the `username` and the `is_staff` property.

Now the final move. We have to update our settings.py.

More specifically the `AUTH_USER_MODEL` property.

```
AUTH_USER_MODEL = 'core.User'
```

This way we are telling Django to use our custom model instead the default one. In the example above, I've created the custom model inside an app named `core`.

How should I reference this model?

Well, there are two ways. Consider a model named `Course`:

```
from django.db import models
from testapp.core.models import User

class Course(models.Model):
    slug = models.SlugField(max_length=100)
    name = models.CharField(max_length=100)
    tutor = models.ForeignKey(User, on_delete=models.CASCADE)
```

This is perfectly okay. But if you are creating a reusable app, that you want to make available for the public, it is strongly advised that you use the following strategy:

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

class Course(models.Model):
    slug = models.SlugField(max_length=100)
    name = models.CharField(max_length=100)
    tutor = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
```

Extending User Model Using a Custom Model Extending AbstractUser

This is pretty straighforward since the class `django.contrib.auth.models.AbstractUser` provides the full implementation of the default User as an abstract model.

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

class User(AbstractUser):
    bio = models.TextField(max_length=500, blank=True)
    location = models.CharField(max_length=30, blank=True)
    birth_date = models.DateField(null=True, blank=True)
```

Then we have to update our settings.py defining the `AUTH_USER_MODEL` property.

```
AUTH_USER_MODEL = 'core.User'
```

In a similar way as the previous method, this should be done ideally in the begining of a project and with an extra care. It will change the whole database schema. Also, prefer to create foreign keys to the User model importing the settings `from django.conf import settings` and referring to the `settings.AUTH_USER_MODEL` instead of referring directly to the custom User model.

Conclusions

Alright! We've gone through four different ways to extend the existing User Model. I tried to give you as much details as possible. As I said before, there is no *best solution*. It will

really depend on what you need to achieve. Keep it simple and choose wisely.

- **Proxy Model:** You are happy with everything Django User provide and don't need to store extra information.
- **User Profile:** You are happy with the way Django handles the auth and need to add some non-auth related attributes to the User.
- **Custom User Model from AbstractBaseUser:** The way Django handles auth doesn't fit your project.
- **Custom User Model from AbstractUser:** The way Django handles auth is a perfect fit for your project but still you want to add extra attributes without having to create a separate Model.

Do NOT hesitate to ask me questions or tell what you think about this post!

You can also join my mailing list. I send exclusive tips directly to your email every week! :-)