SIMPLEISBETTERTHANCOMPLEX

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.

TUTORIAL

How to Extend Django User Model

Jul 22, 2016 18 minutes read 208 comments 209,260 views

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.

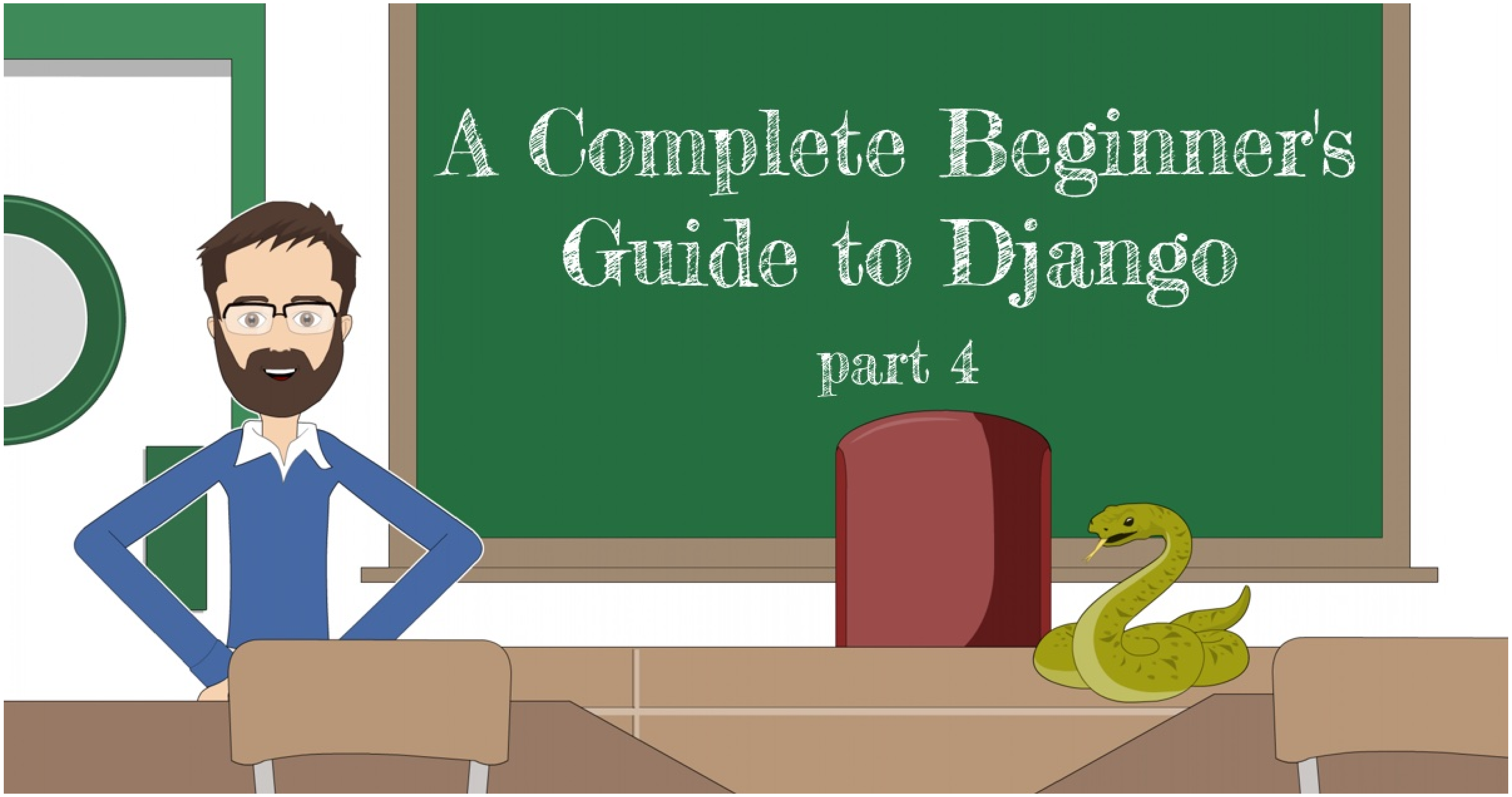
Do NOT hesitate to ask me questions or tell what you think about this post!

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

Related Posts

Django Tips #22 Designing Better Models

How to Implement Multiple User Types with Django



A Complete Beginner's Guide to Django - Part 4

django models auth contrib user

3

Share this post

Search and buy domains from Namecheap. Lowest prices. Buy Now

208 Comments Simple is Better Than Complex 1 Login

Recommend 17 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Ivan Marquez • 8 months ago

somewhy I get this
"AUTH_USER_MODEL refers to model '%s' that has not been installed" % settings.AUTH_USER_MODEL
django.core.exceptions.ImproperlyConfigured: AUTH_USER_MODEL refers to model 'core.User' that has not been installed

21 • Reply • Share ›

Bozhidar Hristov ➦ Ivan Marquez • 4 months ago

substitute 'core' in 'core.User' with your app's name.

1 • Reply • Share ›

Daniel Wise • a year ago

Hi Vitor.

First of all, great work on your website!

First of all, great work on your website.

I am new to Django and i've followed your tutorial about extending the user model.

I used the OneToOneField solution and everything seems to be working perfectly. But I am having trouble with my template's form fields.

I use Django-Registration app for handling registration, and for that (I believe) the form fields are set up on the template's file with {{ form.username }} instead of {{ user.username }}.

The problem that I'm having is that when I try to add the form field in the template for the user to fill out its name, for example, writing {{ user.profile.name }} doesn't return on the template a field. In fact, it renders nothing but the standard fields (username, password1, password2 and email).

I have already tried other things, such as {{ form.profile.name }} or just {{ profile.name }} or {{ myappname.profile.name }} but none of those things seems to work.

Could you give me a hand about that?

P.S: Sorry about my english. It's a bit rusty. :D
Also, don't know if that was the best session to post this. Sorry if it wasn't.

P.S.2: Também sou do Brasil! Só agora vi que você também é.
10 ^ | ˅ • Reply • Share ›

Jack • a year ago
Thanks, Vitor.
Could you also share your experience or thoughts on how social integration (e.g. django-allauth) work or not work with these 4 options of extending user model?
If a user registered the extended user, and later on would like to link with Google+ or facebook account with one click login, how would that change our design.

Cheers again for sharing your knowledge.
8 ^ | ˅ • Reply • Share ›

Barry Melton ➦ Jack • 8 months ago
Not the author, but social authentication is a prime use case for the 'User Profile' approach. The right way to do this is to keep the User instance pristine, and add OneToOne model for storing provider-related data. You *could* extend the

user model, but then you'd end up running migrations on a production instance every time you wanted to add a new authentication provider.

Keeping everything external to the User model allows you a lot of niceties, like being able to more easily adapt to changes by authentication providers (e.g., Google moving from OpenID to oAuth to oAuth2) while being able to support every version of their authentication without having to run a single migration against a production table.

1 ^ | ˅ • Reply • Share ›

Jack ➦ Barry Melton • 8 months ago

Thanks a lot, Barry. To store provider-related data using the OneToOne model, would it then be keeping two versions of password? One for the password comes with the pristine user model, the other one linnked with the social account?

^ | ˅ • Reply • Share ›

Barry Melton ➦ Jack • 8 months ago

Generally speaking, social authentication is a passwordless transaction.

Using OAuth2, for example, all the 'passwords' are handled on the backend in the form of keys. To log a user in with their Google account credentials, step 1 is to create an application in the Google API console. That console gives me a client ID, a secret key, and allows me to specify a redirect_uri.

To get a user token, I request a token using the keys I have, redirect the user to Google to 'allow' my app to have access to their credentials. If they allow it on the Google servers, Google will pass back an authorization code to the redirect_uri I specified above (which is how they prevent others using my keys), and then, behind the scenes, I can exchange the authorization code for an access token, which I can use to perform actions on the user's behalf.

Nowhere in that process do I ever get access to the user's Google password at all, so there's no need to store it. You *do* want to store your client_id and secret key, but you can store those in an environment variable, and read them into your Django settings by using os.environ[]. The authorization codes are temporary, and are generally only good for a single use, or a very limited time, so there's no need to store those anywhere. You *do* want to store the access token and refresh tokens you get after logging the user in, and you would want to store those in the social account model, but they don't give your users any authorization access to *your* site beyond authentication. (Note the distinction between authentication and authorization -- the former is who you are, the latter is what you can do).

In a vanilla Django app, you use the social authentication for Authentication, which you then exchange for a Django Session or Token you create for use with a django-rest-framework API. For Authorization, you use groups and permissions as you ordinarily would.

2 ^ | ˅ • Reply • Share ›

Jack ➦ Barry Melton • 8 months ago

It helps a lot. Cheers, Barry.

^ | ˅ • Reply • Share ›

Derrick Kearney • 7 months ago

Vitor, I recent hit a wall with an issue, that may help others. I used the official doc example to create a custom user class. It was not using the permissions, so I read your article to see how you did it. What I did not realize, was the official doc example MyUser class overrides the has_perm of the PermissionsMixin. My own mistake, for not looking more closely, but it was a simple but painful mistake to make. The comment in the official Django example "# Simplest possible answer: Yes, always", a yes is the opposite of simple. I finally found the answer on SO, after a lot of amendments to the search terms

Disregard, this lead to a far more painful problem. Superusers no longer have permission in the admin. The official documentation is sorely lacking in discussing this scenario of creating custom users and using permissions, imo

5 ^ | ˅ • Reply • Share ›

Mike Ru • a year ago

What should I write to urls.py file?

4 ^ | ˅ • Reply • Share ›

Joseph Marando Daudi • a year ago

Hello Victor, i am having a problem when creating a custom user model to register users with first_name, middle_name, last_name and email only without any password fields; i want the passwords to be auto generated and sent to the user via email, can you help me with that please.

3 ^ | ˅ • Reply • Share ›

Eduardo ➦ Joseph Marando Daudi • a year ago

Hi
I think in your case you should create the pre_save signal wich generate the password and call your send_email method. Another option is create a CustomUserManager (extending from UserManager) and override the create_user method to set the generated password and send mail to user

1 ^ | ⌄ • Reply • Share ›

Dan • a year ago

Hi Vitor - these blogs are really great. I'm referring to them more and more as I go deeper in to the django djungle hehe.
I'm getting an error message on the 'User Profile' option : IntegrityError: UNIQUE constraint failed: myapp_profile.user_id.

I have implemented it almost 1 on 1 with your example and I tried flushing the db. Can you think of a reason why this might happen?

2 ^ | ⌄ • Reply • Share ›

Dan ➦ Dan • a year ago

Investigating a bit further, I can create a user (u = User.objects.create_user('user','password') and update the User attributes (u.password ='1234567') and access the profile attributes through u.profile.addedAttribute. However in the shell, when I save, I always get errors, yet the information is saved to the db. Is this normal behaviour for the user profile method?

2 ^ | ⌄ • Reply • Share ›

Oleg Klimenko • a year ago

Perfect article, shortly, clear and accessibly! Thank you Vitor! :)

2 ^ | ⌄ • Reply • Share ›

Tandavala • 3 months ago

I am almost 3 weeks now trying to solve this problem by my own, in overall this is the solution number 4 that i have found in this wonderful blog, congratulation for your work, i am Angolan since i knew your blog through django Brazil community i have been following your work here and on github.

I am following this tutorial but i am having problem when i run the django server, i made to screenshot one is for the code and other one is for error display, please i need your help to solve this problem. My thanks in advance.

see more

1 ^ | v • Reply • Share ›

firxworx ➦ Tandavala • 2 months ago

In your project's settings.py did you define AUTH_USER_MODEL correctly? It could be trying to load the default and that's the reason for the clashes reported by the system check. Assuming that your custom User is in an app called `core` (per this tutorial) you'd want a line `AUTH_USER_MODEL = 'core.User'` to appear one time in your settings.py.

^ | v • Reply • Share ›

Mayur Patil • 7 months ago

Hi Vitor, I followed this blog to create a custom User Model but after that I am unable to access my admin pannel.
Here are the errors->
Request URL: http://127.0.0.1:8000/admin/login/?next=/admin/
Django Version: 1.11.2
Exception Type: AttributeError
Exception Value:
'User' object has no attribute 'is_staff'
Exception Location: /home/mayur/Documents/codeMatrix/backend/venv/lib/python3.5/site-packages/django/contrib/admin/forms.py in confirm_login_allowed, line 21

1 ^ | v • Reply • Share ›

Udit Makkar ➦ Mayur Patil • a month ago

Dude i am having the same problem and i cant figure it out
have you found the solution to your problem

^ | v • Reply • Share ›

Phil Spelman ➦ Udit Makkar • 19 days ago

HEY! Check out this other tutorial...it includes the proper code for registering your custom model with Django admin (in a brand new, never-been-migrated project)

admin (in a brand new, never been migrated project)
Go down to: "Register your new User model with Django admin"

https://www.fomfus.com/arti...

^ | v • Reply • Share ›

Subrata Bhadury • 7 months ago
Hello, I have an app where two type of users are there, frontend and backend user. Frontend user will signup and can login into the app. Similarly admin user can login in backend. There is no similarity between these two type of users. In this case new frontend model (separate table) is required to manage (signup/login) frontend users, I am guessing it. so, how to achieve this, any suggestions ?

1 ^ | v • Reply • Share ›

Olaf Schüsler • a year ago
Thank you Vitor for this great guide,

I have a question about implementing your second solution. In my software, we have a project mode, which is managed by a 'manager' model and can be subscribed to by 'professional' models. Both models need be able to log in, for which I opted for the fourth solution. However, am I correct that if I would add the @receiver, both the manager and professional model would be generated when a user would be created? How would you suggest to solve this? I tried skipping this altogether, creating my own user instance and adding this to the manager/professional model, but is there a better way?

Many thanks for regarding my question

1 ^ | v • Reply • Share ›

Musharaf Baig ➦ Olaf Schüsler • 3 months ago
Hey Olaf, as far as i understand may be you used use Signal (post_save) and in signal check if the role is manager or professional and save the instance accordingly.

^ | v • Reply • Share ›

Andrew Artajos • a year ago
I am using this one: Custom User Model from AbstractBaseUser. I created a post_save signal for Custom User Model but it doesn't get called.

1 ^ | v • Reply • Share ›

jing • a year ago

How to login and register with the new custom users? Thanks!

1 • Reply • Share ›

Bernardo Duarte • a year ago

Thanks for the post @Vitor Freitas it helped me alot to understande how to exend django's user model.
There is but a question about the second manner, can it overwrite or add new validators to the base user model, like making the user.email unique?

1 • Reply • Share ›

Vitor Hidalgo • a year ago

Hello, Vitor, how are you? Congratulations on the post, I'm having a problem at the moment of saving the form, it's giving me the following message: "RelatedObjectDoesNotExist at / profile /" "User has no profile." Can you help me? Many thanks and congratulations for the post.

1 • Reply • Share ›

Vitor Freitas Mod → Vitor Hidalgo • a year ago

Hey Vitor :-)
It seems like you already had some Users created in the database before implementing the Profile extension.

If that was the case, I would create some sort of "data migration". Usually I do it directly in the Python shell, like this:

```
$ python manage.py shell
> from django.contrib.auth.models import User
> from mysite.core.models import Profile
> users = User.objects.filter(profile=None)
> for user in users:
>     Profile.objects.create(user=user)
```

But then make sure you Profile instances are being created properly, in the **create_user_profile** Signal

8 • Reply • Share ›

moses → Vitor Freitas • 4 months ago

>>> from django.contrib.auth.models import User
>>> from customer.models import Profile
>>> users = User.objects.filter(profile=None)
>>> for user in users:
... Profile.objects.create(user=user)

```
...
Traceback (most recent call last):
File "<console>", line 2, in <module>
File "C:\Users\User\desktop\oneio\Oneioenv\lib\site-packages\django\db\models\manager.py", line 82, in manager_method
return getattr(self.get_queryset(), name)(*args, **kwargs)
File "C:\Users\User\desktop\oneio\Oneioenv\lib\site-packages\django\db\models\query.py", line 415, in create
obj = self.model(**kwargs)
File "C:\Users\User\desktop\oneio\Oneioenv\lib\site-packages\django\db\models\base.py", line 495, in __init__
raise TypeError("'%s' is an invalid keyword argument for this function" % kwarg)
TypeError: 'user' is an invalid keyword argument for this function
>>>
```

^ | ˅ • Reply • Share ›

moses ➦ moses • 4 months ago

hi i'm getting this error above and i don't know how to solve it, im new to django

1 ^ | ˅ • Reply • Share ›

moses ➦ moses • 4 months ago

Never mind, i saw my mistake i had named the onetooneField column customer instead of 'user'. Thanks for this django tutorial series by the way.

^ | ˅ • Reply • Share ›

Thomas Smets ➦ Vitor Freitas • 7 months ago

How can I do when I want to have "normal" users (admins for instance may not have a "Profile") and users with profiles ?

^ | ˅ • Reply • Share ›

Oskar Gmerek ➦ Vitor Freitas • a year ago

Hello. I thinking about if this is possible to do it by django code. If in the future I want to add a field I will be forced to do 'data migration' every time and it is not cool. Is it possible to do by source code?

Regards,
Oskar

^ | ∨ • Reply • Share ›

Chimaobi Emmanuel ➦ Oskar Gmerek • a year ago

I think you can do so using os.popen.

You are not necessarily running away from running it in the terminal but automating it with python.

So you can create a function to help you do it like:

```
def migration:
import os
os.popen("python /path-to-your- project/manage.py migrate")
```

Pardon me! How do I use code display/features in this reply?

^ | ∨ • Reply • Share ›

Vitor Hidalgo ➦ Vitor Freitas • a year ago

Hi Vitor, how are you?
Thanks a lot for the help, that's right I already had a user created in the database, the migration I had already run, I just circled the script in the shell and it worked.
Thanks again for the help.
Abraço ;)

^ | ∨ • Reply • Share ›

Hindsight 20/20 • 2 days ago

Hi! so i'm using python3 and django 2.0.1 -- I use the fourth technique (Using AbstractUser) to create some data - However I'm getting an error - Profile (my User model/class) -- Has no field named password (I declared it as class Profile(AbstractUser):)

Can anyone please help? Thank you!

^ | ∨ • Reply • Share ›

S.H. • 4 days ago

Hi Victor,

Thanks a lot for your great blog.

I followed your instruction (Extending User Model Using a One-To-One Link) to create a Profile model. But there is a weird thing: as I create a new user, the Profile model (via the signal) generate the corresponding profiles for the user. However, it generates TWO profiles for a given user_id (which is the Django assigned id in User model). These two profiles are exactly the same except one being a well-defined integer id and one being None (please see the image below)

see more

^ | ⌄ • Reply • Share ›

Oskar Gmerek • 4 days ago

Hello,

I look at about option #2 OneToOneField. It's one problem with this:
If User is already in db and don't have profile yet than can't save user object before manually creating profile for this user.

What do you think about this code?

```
@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
if created:
Profile.objects.create(user=instance)
else:
try:
instance.profile.save()
except Profile.DoesNotExist:
Profile.objects.create(user=instance)
```

I'm just learning so I think this solution may have some weaknesses or not be correct for another reason.

^ | ˅ • Reply • Share ›

Kartik Dube • 19 days ago

YOU ARE A LEGEND. period

^ | ˅ • Reply • Share ›

Navkant Tyagi • 20 days ago

Hi Vitor,

Great tutorial. I have one doubt

Why do we need this method

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

while we have
Profile.objects.create
defined in create_user_profile?

I tried it, worked well without save_user_profile!!

^ | ˅ • Reply • Share ›

Michael Chen • a month ago

Hi,

I just wanted to ask why we need the save_user_profile receiver for the one to one model. Whenever we save an already existing User, why do we need to save the profile as well? Shouldn't only the User have been updated?

^ | ˅ • Reply • Share ›

Navkant Tyagi • a month ago

I have seen people using django.contrib.auth import get_user_model
sometimes seeting.AUTH_USER_MODEL
can u clarify my point?

^ | ˅ • Reply • Share ›

Reply • Share ›

Mark Cananzi • a month ago

Hey Vitor, great tutorial.

Having some difficulty with option three (yes the hairy one) when applying to existing project.
Slight mods to your code are:
App name is "users"
UserManager is CustomUserManager and
User is CustomUser

On migration I get this error:
django.core.exceptions.FieldError: Unknown field(s) (username) specified for CustomUser

settings.py includeds:
AUTH_USER_MODEL = 'users.CustomUser'

users/admin.py includes:
from .models import CustomUserManager, CustomUser

I've looked this through many times. Not sure what I've missed.

Reply • Share ›

guy fawkes • 2 months ago

Hey Vitor, thanks for your informative articles. so i was wondering how do i go about providing password fields? Im only able to display the other fields but it's only logical users need to create passwords along with the other information.

Reply • Share ›

George Lambert • 2 months ago

I'm also having problems with this tutorial with Django 2.0

Reply • Share ›

Dhruv Marwha • 2 months ago

Using the One-To-One link Extension,is it possible for me to associate several addresses with a given user.?

I need this for a E-commerce Project.

Reply • Share ›

Thomas Weholt • 2 months ago

Hi and thanks for another great article! I just got one question. In my project I've created a main package and my custom user model extending AbstractUser in a subpackage and I get an error when I try to reference to it in settings.py like this:

AUTH_USER_MODEL = 'mainpackge.subpackagewithmyusermodel.User'

Is there any way to achive this?

^ | ˅ • Reply • Share ›

Adnan Sheikh • 2 months ago

Perfectly explained and very helpful for beginners. Thank you Vitor. :)

^ | ˅ • Reply • Share ›

Tandavala • 3 months ago

I am one week now trying to solve this problem by my own, fortunately i found the solution here, in overall this is the solution number 4 i have found in this blog.

I am trying to follow the tutorial but i am having some problem when i run the django server i made two screenshot one is for the code and the other one the error display, please i need your help.

see more

Reply • Share ›

Roxanne Apopemptic • 3 months ago

Hello,
I was wondering which option you would recommend when I already have a User model with authentication tokens. I want another user "type" to be able to log in and have separate permissions and extra fields. Right now I'm just using a model for this function, but there is no possibility of login. Basically I want other models to distinguish between the two different models being logged in through the token. Thank you.

Reply • Share ›

Load more comments

ALSO ON SIMPLE IS BETTER THAN COMPLEX

How to Setup Amazon S3 in a Django Project

32 comments • 8 months ago

Avatar xtornasol512 — Cool Thanks Bro!

A Complete Beginner's Guide to Django - Part 6

35 comments • 6 months ago

Avatar Grill Chicken — Great tutorial series! thanks....Can i see the giant pic to response in the create form.

Django Tips #20 Working With Multiple Settings Modules

30 comments • 9 months ago

Avatar Bernardo Gomes — Good Article !

A Complete Beginner's Guide to Django - Part 5

39 comments • 6 months ago

Avatar Pavlo Olshansky — Hi, Vitor.Thanks for your tutorial, it's really great!Wait for your next tutorials. Hope this will be smth like CBV, permissions or DRF. This will be really helpful

Subscribe to our Mailing List

Receive exclusive Django tips every week!

Your email address

SUBSCRIBE

Popular Posts

How to Extend Django User Model

How to Setup a SSL Certificate on Nginx for a Django Application

[How to Deploy a Django Application to Digital Ocean]

© 2015-2018 simple is better than complex cc by-nc-sa 3.0 // about cookies privacy policy