django-authtools Documentation

Release 1.6.0

Fusionbox, Inc.

Jul 21, 2017

Contents

A custom user model app for Django 1.5+. It tries to stay true to the built-in User model for the most part. The main differences between authtools and django.contrib.auth are a User model with email as username and classed-based auth views.

It provides its own custom User model, views, urls, ModelAdmin, and Forms. The Admin classes, views, and forms, however, are all User model agnostic, so they will work with any User model. django-authtools also provides base classes that make creating your own custom User model easier.

Contents:

CHAPTER 1

Introduction

Before you use this, you should probably read the documentation about custom User models.

Installation

1. Install the package:

```
$ pip install django-authtools
```

   Or you can install it from source:

```
$ pip install -e git+http://github.com/fusionbox/django-authtools@master
↪#egg=django-authtools-dev
```

2. Run the authtools migrations:

```
$ python manage.py migrate
```

Quick Setup

If you want to use the User model provided by authtools (a sensible choice), there are three short steps.

1. Add `authtools` to your `INSTALLED_APPS`.
2. Add the following to your settings.py:

```
AUTH_USER_MODEL = 'authtools.User'
```

3. Add `authtools.urls` to your URL patterns:

```
urlpatterns = patterns('',
    # ...
    url(r'^accounts/', include('authtools.urls')),
    # ...
)
```

This will set you up with a custom user that

- uses email as username
- has one `name` field instead of `first_name`, `last_name` (see Falsehoods Programmers Believe About Names)

It also gives you a registered admin class that has a less intimidating *`ReadOnlyPasswordHashWidget`* and the standard auth views (see *Views*).

But it's supposed to be a *custom* User model!

Making a User model that only concerns itself with authentication and authorization just *might* be a good idea. I recommend you read these:

- The User-Profile Pattern in Django
- Williams, Master of the "Come From"

Also, please read this quote from the Django documentation:

> **Warning:** Think carefully before handling information not directly related to authentication in your custom User Model.
>
> It may be better to store app-specific user information in a model that has a relation with the User model. That allows each app to specify its own user data requirements without risking conflicts with other apps. On the other hand, queries to retrieve this related information will involve a database join, which may have an effect on performance.

However, there are many valid reasons for wanting a User model that you can change things on. django-authtools allows you to do that too. django-authtools provides a couple of abstract classes for subclassing.

class `authtools.models.`**`AbstractEmailUser`**
: A no-frills email as username model that satisifes the User contract and the permissions API needed for the Admin site.

class `authtools.models.`**`AbstractNamedUser`**
: A subclass of *`AbstractEmailUser`* that adds a name field.

If want to make your custom User model, you can use one of these base classes.

Tip: If you are just adding some methods to the User model, but not changing the database fields, you should consider using a proxy model.

If you wanted a User model that had `full_name` and `preferred_name` fields instead of just `name`, you could do this:

```
from authtools.models import AbstractEmailUser

class User(AbstractEmailUser):
    full_name = models.CharField('full name', max_length=255, blank=True)
    preferred_name = models.CharField('preferred name',
        max_length=255, blank=True)

    def get_full_name(self):
        return self.full_name

    def get_short_name(self):
        return self.preferred_name
```

CHAPTER 2

Admin

django-authtools provides a couple of Admin classes. The default one is *NamedUserAdmin*, which provides an admin similar to django.contrib.auth. If you are not using the *AbstractNamedUser*, you might want the *UserAdmin* instead.

If you are using your own user model, authtools won't register an Admin class to avoid problems. If you define `REQUIRED_FIELDS` on your custom model, authtools will add those to the first fieldset.

class `authtools.admin.`**NamedUserAdmin**
: This is the default Admin that is used if you use `authtools.models.User` as you `AUTH_USER_MODEL`. Provides an admin for the default `authtools.models.User` model. It includes the default Permissions and Important Date sections.

class `authtools.admin.`**UserAdmin**
: Provides a generic admin class for any User model. It behaves as similarly to the built-in UserAdmin class as possible.

class `authtools.admin.`**StrippedUserAdmin**
: Provides a simpler view on the UserAdmin, it doesn't include the Permission filters or the Important Dates section.

class `authtools.admin.`**StrippedNamedUserAdmin**
: Same as StrippedUserAdmin, but for a User model that has a `name` field.

CHAPTER 3

Forms

django-authtools provides several Form classes that mimic the forms in django.contrib.auth.forms, but work better with `USERNAME_FIELD` and `REQUIRED_FIELDS`. These forms don't require the `authtools.models.User` class in order to work, they should work with any User model that follows the `User class contract`.

class `authtools.forms.`**`UserCreationForm`**
Basically the same as django.contrib.auth, but respects `USERNAME_FIELD` and `User.REQUIRED_FIELDS`.

class `authtools.forms.`**`CaseInsensitiveUsernameFieldCreationForm`**
This is the same form as `UserCreationForm`, but with an added method, `clean_username` which lowercases the username before saving. It is recommended that you use this form if you choose to use either the *`CaseInsensitiveUsernameFieldModelBackend`* authentication backend class.

Note: This form is also available sa CaseInsensitiveEmailUserCreationForm for backwards compatibility.

---

class `authtools.forms.`**`UserChangeForm`**
A normal ModelForm that adds a `ReadOnlyPasswordHashField` with the *`BetterReadOnlyPasswordHashWidget`*.

class `authtools.forms.`**`AdminUserChangeForm`**
Same as *`UserChangeForm`*, but adds a link to the admin change password form.

class `authtools.forms.`**`FriendlyPasswordResetForm`**
Basically the same as `django.contrib.auth.forms.PasswordResetForm`, but checks the email address against the database and gives a friendly error message.

> **Warning:** This form leaks user email addresses. Please refer to the view `friendly_password_reset()`.

It also provides a Widget class.

class `authtools.forms.`**`BetterReadOnlyPasswordHashWidget`**
This is basically the same as django's `ReadOnlyPasswordHashWidget`, but it provides a less

intimidating user interface. Whereas django's Widget displays the password hash with it's salt, *BetterReadOnlyPasswordHashWidget* simply presents a string of asterisks.

CHAPTER 4

Views

django-authtools provides the following class-based views, intended to be *mostly* drop-in replacements for their built-in counterparts.

In addition to the built-in views, there is a new *PasswordResetConfirmAndLoginView* that logs in the user and redirects them after they reset their password.

Note: The view functions in Django were wrapped in decorators. The classed-based views provided by django-authtools have the same decorators applied to their view functions. Any subclasses of these views will also have the same decorators applied.

---

class authtools.views.**LoginView**

The view function authtools.views.login() replaces django.contrib.auth.views.login().

disallow_authenticated

When True, authenticated users will be automatically redirected to the success_url when visiting this view. Defaults to True.

class authtools.views.**LogoutView**

The view functions authtools.views.logout() and authtools.views.logout_then_login() replace django.contrib.auth.views.logout() django.contrib.auth.views.logout_then_login() respectively.

url

The URL to redirect to after logging in. This replaces the login_url parameter present in the built-in function.

For the logout_then_login() this is default to LOGIN_REDIRECT_URL.

template_name

If *url* is None and there is no next parameter, *LoginView* will act like a TemplateView and display a template.

class authtools.views.**PasswordChangeView**
The view function authtools.views.password_change() replaces django.contrib.auth.views.password_change().

success_url
This replaces the post_change_redirect parameter present in the built-in function. Uses the next URL parameter or defaults to the 'password_change_done' view.

class authtools.views.**PasswordChangeDoneView**
The view function authtools.views.password_change_done() replaces django.contrib.auth.views.password_change_done().

class authtools.views.**PasswordResetView**
The view function authtools.views.password_reset() replaces django.contrib.auth.views.password_reset().

success_url
The pages which the user should be redirected to after requesting a password reset. This replaces the next_page parameter present in the built-in function. Defaults to the 'password_reset_done' view.

form_class
The form class to present the user. This replaces the password_reset_form parameter present in the built-in function.

Django 1.6 removed the email check from this view in order to avoid leaking user email addresses.

In some cases, this can worsen user experience without providing any extra security. For example, if email addresses are unique, then the registration form will be leaking email addresses.

If you're in this case, and you wish to improve usability of this view informing the user if they did any typo, you can do:

```
# yourproject/urls.py
urlpatterns += patterns( # ...
    # ...
    url('^auth/password_reset/$',
        PasswordResetView.as_view(FriendlyPasswordResetForm),
        name='password_reset'),
    url('^auth/', include('authtools.urls'),
    # ...
)
```

class authtools.views.**PasswordResetDoneView**
The view function authtools.views.password_reset_done() replaces django.contrib.auth.views.password_reset_done().

class authtools.views.**PasswordResetConfirmView**
The view function authtools.views.password_reset_confirm() replaces django.contrib.auth.views.password_reset_confirm().

success_url
Where to redirect the user after resetting their password. This replaces the post_reset_redirect parameter present in the built-in function.

form_class
The form class to present the user when resetting their password. The form class must provide a save method like in the django.contrib.auth.forms.SetPasswordForm This replaces the set_password_form parameter present in the built-in function. Default is django.contrib.auth.forms.SetPasswordForm.

Note: Django 1.6 changed this view to support base-64 encoding the user's pk. Django provides a different view for each type of encoding, but our view works with both, so we only have a single view.

This was a backwards-incompatible change in Django, so be sure to update your urlpatterns and anywhere you reverse the `password_reset_confirm` URL (like the password reset email template, `registration/password_reset_email.html`).

class `authtools.views.`**PasswordResetConfirmAndLoginView**

Available as the view function `authtools.views.password_reset_confirm_and_login()`.

This is like *PasswordResetConfirmView*, but also logs the user in after resetting their password. By default, it will redirect the user to the `LOGIN_REDIRECT_URL`.

If you wanted to use this view, you could have a url config that looks like:

```
urlpatterns = patterns('',
    url('^reset/(?P<uidb36>[0-9A-Za-z]{1,13})-(?P<token>[0-9A-Za-z]{1,13}-[0-9A-
↪Za-z]{1,20})/$',
        'authtools.views.password_reset_confirm_and_login', name='password_reset_
↪confirm'),
    url('^', include('authtools.urls')),
)
```

Note: In Django 1.6, the `uidb36` kwarg was changed to `uidb64`, so your url will look like:

```
url(r'^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>[0-9A-Za-z]{1,13}-[0-9A-Za-z]
↪{1,20})/$',
    'authtools.views.password_reset_confirm_and_login',
    name='password_reset_confirm'),
```

Like *PasswordResetConfirmView*, this view supports both `uid36` and `uidb64`.

class `authtools.views.`**PasswordResetCompleteView**

The view function `authtools.views.password_reset_complete()` replaces `django.contrib.auth.views.password_reset_complete()`.

CHAPTER 5

Authentication Backends

django-authtools provides two authentication backend classes. These backends offer more customization for authentication.

class `authtools.backends.`**`CaseInsensitiveUsernameFieldModelBackend`**
Enables case-insensitive logins for the User model. It works by simply lowercasing usernames before trying to authenticate.

There is also a *`CaseInsensitiveUsernameFieldBackendMixin`* if you need more flexibility.

To use this backend class, add it to your settings:

```
# settings.py
AUTHENTICATION_BACKENDS = [
    'authtools.backends.CaseInsensitiveUsernameFieldModelBackend',
]
```

> **Warning:** Use of this mixin assumes that all usernames are stored in their lowercase form, and that there is no way to have usernames differing only in case. If usernames can differ in case, this authentication backend mixin could cause errors in user authentication. It is advised that you use this mixin in conjuction with the *`CaseInsensitiveUsernameFieldCreationForm`* form.

class `authtools.backends.`**`CaseInsensitiveUsernameFieldBackendMixin`**
Mixin enabling case-insensitive logins.

CHAPTER 6

Tutorials

Here is a list of tutorials for dealing with custom User models.

How To Migrate to a Custom User Model

If you are using the built-in Django User model and you want to switch to an authtools-based User model, there are certain steps you have to take in order to keep all of your data. These are steps that have worked for me in the past, maybe they will help to inform your journey.

This tutorial assumes that you are using South for migrations. If you aren't you probably should be using it. Unless of course, it's the future and the schema-alteration of Django has been completed and merged.

It also assumes that you already have users in your database and that you need to preserve that data. If you don't already have users in your database, you switch easily already.

This tutorial shows the easy way to migrate custom Users, keeping the same database table. If you want to move to your own database table, there is an excellent answer on StackOverflow.

Step 1: Backup your database

There are several commands for doing this depending on your RDBMS (`pg_dump`, `mysqldump`, `cp`). If you don't want to worry about those, you could also look for a solution like django-backupdb. You *do not* want to start this process without having a backup of your database.

Steps 2 and 3 are actually completely safe. They don't actually affect the database. What they do accomplish is moving the authoritative source of control over the User model class from django to your code.

Step 2: Make a new app

This is the app where your custom User model will live. I usually call this app `accounts`.

```
$ python manage.py startapp accounts
```

In your new app, edit the models file and add the following:

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

class User(AbstractUser):
    class Meta:
        db_table = 'auth_user'
```

This will put the User model in the same database table as the old one. This is not ideal, but it is the easiest way to do this migration.

Add your `accounts` app to `INSTALLED_APPS`.

Set the `AUTH_USER_MODEL` setting to point to your new User model.

```
AUTH_USER_MODEL = 'accounts.User'
```

Step 3: Seize control

Generate an initial migration for the `accounts` app.

```
$ python manage.py schemamigration --initial accounts
```

If you are working on a new database and are running the migrations from scratch, you can run that migration normally. However, if you are working on an existing database, this migration will fail because the tables it attempts to create already exist. You will have to fake run this migration.

```
$ python manage.py migrate --fake accounts 0001
```

Note: If you are very certain that these migrations will *never* be run on an empty database, you can replace the bodies `forwards` and `backwards` migrations with `pass`. This is not a good idea though.

Step 4: Conquer

Your `accounts` app is now the authoritative source for the User model. You are in charge now.

Go build stuff.

Optional Step 5: Customize

Warning: There is a potential unique constraint failure here. If you don't have emails for all of your users, you won't be able to migrate. If you don't have emails for all of your users, they won't be able to log in either, so you should make sure that you have all of those email addresses first.

Now that you have control of the User model, there are tons of customizations that you can do. One thing that I like to do is treat `email` as the username and get rid of `first_name`/`last_name` in favor of a single `name` field. Here's how I've done it in the past.

1. Install django-authtools.

```
$ pip install django-authtools
```

2. Add the fields that I want to User. In this case, all I want to add is `name`. `email` already exists on User, but I do need to make it unique if I'm going to treat it as a username.

   Here is an implementation of the User model using *`authtools.models.AbstractNamedUser`* as a base. It preserves all of the fields that are on the built-in User model, but adds `name` and treats `email` as the username.

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

from authtools.models import AbstractNamedUser


class User(AbstractNamedUser):
    username = models.CharField(_('username'), max_length=30, unique=True)
    first_name = models.CharField(_('first name'), max_length=30, blank=True)
    last_name = models.CharField(_('last name'), max_length=30, blank=True)

    class Meta:
        db_table = 'auth_user'
```

   I still have `first_name` and `last_name` because I have to preserve that data, I will get rid of those fields in step 5. When you are altering the schema and migrating data, the South tutorial on data migrations recommends that you split it up into 3 steps.

3. Make a schema migration to add those fields.

```
$ python manage.py schemamigration --auto accounts
```

4. Make a data migration to copy `first_name`/`last_name` into `name`.

```
$ python manage.py datamigration accounts consolidate_name_field
```

   Here is an example of a migration that does this:

```
class Migration(DataMigration):
    def forwards(self, orm):
        for user in orm['accounts.User'].objects.all():
            user.name = user.first_name + ' ' + user.last_name
            user.save()

    def backwards(self, orm):
        for user in orm['accounts.User'].objects.all():
            # If there are more than two names, assume that the rest
            # are their last names.
            user.first_name, _, user.last_name = user.name.partition(' ')
            user.save()
```

   The backwards migration does make some assumptions about how names work, but those are the assumptions you are forced to make when using a system that assumes people have two names.

5. Delete the columns you don't want on your User model. For me, that's `username`, `first_name`, and `last_name`. My User model now looks like this:

```
class User(AbstractNamedUser):
    class Meta:
        db_table = 'auth_user'
```

6. Generate a migration that deletes those extra fields.

```
$ python manage.py schemamigration --auto accounts
```

   You will be presented with a question about what to do in the backwards migration. The `username` field was non-nullable, which means it's impossible to go back. I would select to disable backwards migrations.

7. Run the migrations.

```
$ python manage.py migrate accounts
```

8. Watch YouTube. You are done.

How To Create Users Without Setting Their Password

When creating a new user through Django's admin interface, you are asked to enter the new user's password. This is less than ideal, because it requires the admin to think of a password for someone else, communicate it to them somehow, and then the user must remember to change it. A better way would be to send a password-reset email to the new user, allowing them to enter their own password.

To implement this, we need to provide a user-creation form that has an optional (instead of required, like the built-in form) password field and a User admin that uses the form and sends the password-reset email when creating a new user.

We'll subclass *UserCreationForm* to create a form with optional password fields:

```
from django import forms
from authtools.forms import UserCreationForm

class UserCreationForm(UserCreationForm):
    """
    A UserCreationForm with optional password inputs.
    """

    def __init__(self, *args, **kwargs):
        super(UserCreationForm, self).__init__(*args, **kwargs)
        self.fields['password1'].required = False
        self.fields['password2'].required = False
        # If one field gets autocompleted but not the other, our 'neither
        # password or both password' validation will be triggered.
        self.fields['password1'].widget.attrs['autocomplete'] = 'off'
        self.fields['password2'].widget.attrs['autocomplete'] = 'off'

    def clean_password2(self):
        password1 = self.cleaned_data.get("password1")
        password2 = super(UserCreationForm, self).clean_password2()
        if bool(password1) ^ bool(password2):
            raise forms.ValidationError("Fill out both fields")
        return password2
```

Then an admin class that uses our form and sends the email:

```
from django.contrib.auth import get_user_model
from django.contrib.auth.forms import PasswordResetForm
from django.utils.crypto import get_random_string
from authtools.admin import NamedUserAdmin

User = get_user_model()

class UserAdmin(NamedUserAdmin):
    """
    A UserAdmin that sends a password-reset email when creating a new user,
    unless a password was entered.
    """
    add_form = UserCreationForm
    add_fieldsets = (
        (None, {
            'description': (
                "Enter the new user's name and email address and click save."
                " The user will be emailed a link allowing them to login to"
                " the site and set their password."
            ),
            'fields': ('email', 'name',),
        }),
        ('Password', {
            'description': "Optionally, you may set the user's password here.",
            'fields': ('password1', 'password2'),
            'classes': ('collapse', 'collapse-closed'),
        }),
    )

    def save_model(self, request, obj, form, change):
        if not change and (not form.cleaned_data['password1'] or not obj.has_usable_
↪password()):
            # Django's PasswordResetForm won't let us reset an unusable
            # password. We set it above super() so we don't have to save twice.
            obj.set_password(get_random_string())
            reset_password = True
        else:
            reset_password = False

        super(UserAdmin, self).save_model(request, obj, form, change)

        if reset_password:
            reset_form = PasswordResetForm({'email': obj.email})
            assert reset_form.is_valid()
            reset_form.save(
                request=request,
                use_https=request.is_secure(),
                subject_template_name='registration/account_creation_subject.txt',
                email_template_name='registration/account_creation_email.html',
            )
```

Using `django.contrib.auth.forms.PasswordResetForm` allows us to share the email-sending code with Django. If you wanted to change the template the email uses, `email_template_name` would be the place to do it.

Now we can replace the installed UserAdmin with our own.

```
from django.contrib import admin
admin.site.unregister(User)
```

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

You can view the `complete admin.py file here.`

CHAPTER 7

Talks

2013 August 27 - Boulder Django

You can see the slides for "django-authtools, Custom User Model for Everyone!" given at Boulder Django on Speaker Deck.

CHAPTER 8

Contributing

We welcome contributions of all sizes, whether it be a small text change or a large new feature. Here are are some steps for getting started contributing.

Getting Started

1. Install the development requirements:

```
$ pip install -r requirements-dev.txt
```

Running Tests

The best way to run the tests is using tox. You can run the tests on all of our supported Python and Django versions by running:

```
$ tox
```

You can also run specific targets using the `-e` flag.

```
$ tox -e py33-dj18
```

A full list of available tox environments is in the `tox.ini` configuration file.

django-authtools comes with a test suite that inherits from the built-in Django auth test suite. This helps us ensure compatibility with Django and that we can get a little bit of code reuse. The tests are run three times against three different User models.

You can get a test coverage report by running `make coverage`. We do not strive for 100% coverage on django-authtools, but it is still a useful metric.

Building Documentation

You can build the documentation by running

```
$ make docs
```

If you are editing the `README.rst` file, please make sure that it compiles correctly using the `longtest` command that is provided by `zest.releaser`.

```
$ longtest
```

CHAPTER 9

CHANGES

1.6.0 (2017-06-14)

- Add support for Django 1.9, 1.10, 1.11 (Jared Proffitt #82)
- Remove old conditional imports dating as far back as Django 1.5
- Update readme

1.5.0 (2016-03-26)

- Update various help_text fields to match Django 1.9 (Wenze van Klink #51, Gavin Wahl #64, Jared Proffitt #67, Ivan VenOsdel #69)
- Documentation fixes (Yuki Izumi #52, Piët Delport #60, Germán Larraín #65)
- Made case-insensitive tooling work with more than just USERNAME_FIELD='username' (Jared Proffitt, Rocky Meza #72, #73)

1.4.0 (2015-11-02)

- Dropped Django 1.7 compatibility (Antoine Catton)
- Add Django 1.8 compatibility (Antoine Catton, Gavin Wahl, #56)
- **Backwards Incompatible:** Remove 1.6 URLs (Antoine Catton)
- **Backwards Incompatible:** Remove view functions

1.3.0 (unreleased)

- Added Django 1.7 compatibility (Antoine Catton, Rocky Meza, #35)
- `LoginView.disallow_authenticated` was changed to `LoginView.allow_authenticated`
- `LoginView.disallow_authenticated` was deprecated.
- **Backwards Incompatible:** `LoginView.allow_authenticated` is now `True` by default (which is the default behavior in Django)
- Create migrations for authtools.

  If updating from an older authtools, these migrations must be run on your apps:

```
$ python manage.py migrate --fake authtools 0001_initial

$ python manage.py migrate
```

1.2.0 (2015-04-02)

- Add CaseInsensitiveEmailUserCreationForm for creating users with lowercased email address usernames (Bradley Gordon, #31, #11)
- Add CaseInsensitiveEmailBackendMixin, CaseInsensitiveEmailModelBackend for authenticating case-insensitive email address usernames (Bradley Gordon, #31, #11)
- Add tox support for test running (Piper Merriam, #25)

1.1.0 (2015-02-24)

- PasswordChangeView now handles a `next` URL parameter (#24)

1.0.0 (released August 16, 2014)

- Add friendly_password_reset view and FriendlyPasswordResetForm (Antoine Catton, #18)
- **Bugfix** Allow LOGIN_REDIRECT_URL to be unicode (Alan Johnson, Gavin Wahl, Rocky Meza, #13)
- **Backwards Incompatible** Dropped support for Python 3.2

0.2.2 (released July 21, 2014)

- Update safe urls in tests
- Give the ability to restrain which users can reset their password
- Add send_mail to AbstractEmailUser. (Jorge C. Leitão)

0.2.1

- Bugfix: UserAdmin was expecting a User with a *name* field.

0.2.0

- Django 1.6 support.

  Django 1.6 broke backwards compatibility of the `password_reset_confirm` view. Be sure to update any references to this URL. Rather than using a separate view for each encoding, authtools uses a single view that works with both.

- Bugfix: if LOGIN_URL was a URL name, it wasn't being reversed in the PasswordResetConfirmView.

0.1.2 (released July 01, 2013)

- Use `prefetch_related` in the UserChangeForm to avoid doing hundreds of `ContentType` queries. The form from Django has the same feature, it wasn't copied over correctly in our original form.

0.1.1 (released May 30, 2013)

- some bugfixes:
- Call `UserManager.normalize_email` on an instance, not a class.
- `authtools.models.User` should inherit its parent's `Meta`.

0.1.0 (released May 28, 2013)

- django-authtools

CHAPTER 10

Development

Development for django-authtools happens on GitHub. Pull requests are welcome. Continuous integration is hosted on Travis CI.

Index