Django Improved User Documentation

Release 1.0.1

Russell Keith-Magee, Andrew Pinkham

Dec 13, 2021

Contents:

Welcome! Below you will find the table of contents for Improved User.

If you're in a rush, head over to *Quickstart: Using Improved User*.

If you're new and want to see what your options are, please read *Select a Configuration Method for Improved User*.

CHAPTER 1

Quickstart: Using Improved User

This document provides a quick tutorial for the recommended way to setup Improved User.

See *Select a Configuration Method for Improved User* for an overview of options and tradeoffs.

1.1 Installation

In a Terminal, use `pip` to install the package from PyPI. To use the *`UserFactory`* provided by the package to allow for testing with `factory_boy`, include it in the installation.

```
$ pip install django-improved-user[factory]
```

If `factory_boy` is unnecessary, it can be omitted by installing normally.

```
$ pip install django-improved-user
```

1.2 Configuration and Usage

1. In a Django project, create a new app. For the purposes of this documentation, we will assume the name of your new app is `user_app`, but you could name it whatever you wish.

   ```
   $ python3 manage.py startapp user_app
   ```

2. In your project's settings, add `user_app.apps.UserAppConfig` to `INSTALLED_APPS` (replace `user_app` and `UserAppConfig` as necessary).

3. In `user_app/models.py`, import Improved User's *`AbstractUser`*.

```
from improved_user.model_mixins import AbstractUser
```

4. Create a new `User` model. If you omit comments, you may need to add `pass` to the line below the class.

```
class User(AbstractUser):
    """A User model that extends the Improved User"""
```

Attention: If you add your own fields to the model, you may wish to modify `REQUIRED_FIELDS`.

5. Define or replace `AUTH_USER_MODEL` in your project settings with the new model, as below (replace `user_app` with the name of your own app).

```
AUTH_USER_MODEL='user_app.User'
```

Tip: Remember to use `get_user_model()` to get your new model. Don't import it directly!

6. In Django > 1.9, while still in settings, change `UserAttributeSimilarityValidator` to match correct *`AbstractUser`* fields, as shown below.

```
AUTH_PREFIX = 'django.contrib.auth.password_validation.'
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': AUTH_PREFIX + 'UserAttributeSimilarityValidator',
        'OPTIONS': {
            'user_attributes': ('email', 'full_name', 'short_name')
        },
    },
    # include other password validators here
]
```

7. You're done! Run migrations or go back to programming the rest of your project.

Note: Improved user also comes with forms, test factories, and an admin panel. Take a look at the *Package Reference* for more information.

CHAPTER 2

Quickstart: Contributing

First off, thanks for taking the time to contribute!

This document assumes that you have forked and cloned the repository to work on the package locally. If you are unsure how to do this, please see the *How to Contribute* documentation.

To test the package, start by installing it locally.

```
$ pip install -r requirements.txt
$ python setup.py develop
```

To run the test suite on a single version of Django (assuming you have a version of Django installed), run the `runtests.py` script from the root of the project.

```
$ python runtests.py
```

You can limit tests or pass paramaters as when using `manage.py test`.

```
$ ./runtests.py tests.test_basic -v 3
```

If you have all of the supported Python versions installed (Python 3.4, 3.5, 3.6, and 3.7), you may use `tox` to run all linters and test the package with multiple versions of Python and Django.

```
$ tox
```

You may also limit tests to specific environments or test suites with tox. For instance:

```
$ tox -e py36-django111-unit tests.test_basic
$ tox -e py36-django111-integration user_integration.tests.TestViews.test_home
```

Any change to the code should first be discussed in an issue.

For any changes, please create a new branch, make your changes, and open a pull request on github agains the `development` branch. Refer to the issue you are fixing or building. To make review of the PR easier, please commit small, targeted changes. Multiple small commits with clear messages make reviewing changes easier. Rebasing your branch to help clean up your changes is encouraged. Please remember that this is a volunteer-driven project; we will look at your PR as soon as possible.

CHAPTER 3

Project Rationale

While working together in late 2016, Russell Keith-Magee and Andrew Pinkham— original authors of the project—discussed the repetitive nature of rebuilding a best-practices email-based User model in new Django projects. The two were tired of redoing the same work, and decided to open-source code based on what they'd learned previously.

Russell's *Red User, Blue User, MyUser, auth.User* talk from DjangoCon US 2013 and PyCon AU 2017 (video below) provides a breakdown of the problems with Django's existing approach to identity-handling, as well as an introduction to using custom User models in Django.

In turn, Andrew wished for more modularity in Django's existing `auth` codebase. Having described the process of creating custom User models in his book, Django Unleashed, Andrew felt that developers should be able to import and compose classes to properly integrate with Django's permissions and/or admin. The two set out to build a project that would:

1. provide a User model that authenticates via email (not username)
2. provide a User model with global identity name-fields (full name and short name, rather than the limited anglo-centric first and last name)
3. provide mix-in classes to allow developers to easily compose a new User model.

The project is originally based on the app that Russell had built over a decade of working in Django. Andrew took what he had learned from Django Unleashed and his consulting experience, and integrated it into the project. The result is a `User` model that can be used out of the box (see *Quickstart: Using Improved User* for details) and a set of *mix-in classes* to allow for creation of new User models (notably, the `DjangoIntegrationMixin`).

We hope you find our work useful!

CHAPTER 4

Select a Configuration Method for Improved User

The goal of this package is to improve your project's User model. To that end, Improved User may be used in three different ways. You may:

1. inherit *AbstractUser* in your own User model (**extension**);
2. use the supplied *User* model directly (**integration**);
3. create your own User model using the supplied model mix-in classes (**replacement**).

Tip: It is generally considered a good idea to change the User model as infrequently and as little as possible, given the possibility of security problems. Creating a Profile model—which has a foreign key to the User model—to store your users' information can help avoid changes to the User model.

4.1 Extension Method

The extension method is the recommended method to use when configuring Improved User. Instructions for this method are found in *Quickstart: Using Improved User*. This method gives the developer the most control and flexibility, at the cost of having slightly extra code. This method is the least likely to cause you problems in the long run, as it grants you control of the model fields and migrations for your User model, and gives you the opportunity of entirely removing Improved User in the future if you need to.

4.2 Integration Method

The integration option is the simplest, and uses the least code. However, it is also the least flexible, as it assumes that you will never change the structure of the *User* model. While this method may work fine for many, the amount of work required to deal with any potential future change is very high. In many ways, it is the most similar to Django's own User: you gain all of the benefits of the class directly, but forgo the ability to control or remove the model in the future without serious work. You may refer to *How To: Integrate Improved User Directly* to use this method.

Warning: It will always be possible to switch between the extension and replacement methods, but is difficult to migrate to or from the integration method.

4.3 Replacement Method

The replacement method comes with the same trade-offs as the extension method, but should be used in the event any of the fields included in the `AbstractUser` are not desired. We recommend this method only to those very familiar with Django. For more information, please refer to *How To: Create a Custom User using Mixins*.

CHAPTER 5

Warning about Email Case-Sensitivity

RFC 5321 states that the `mailbox` in `mailbox@hostname` of an email format is case-sensitive. `ANDREW@example.com` and `andrew@example.com` are therefore different email addresses (the domain is case-insensitive).

Django's `EmailField` follows the RFC, and so, therefore, does Improved User.

Today, many email providers have made their email systems case-insensitive. However, not all providers have done so. As such, if we were to provide a custom case-insensitive `EmailField`, we may be alienating your users without you even knowing!

What's more, we follow the RFC because not doing so can cause obscure security issues.

When creating your project's templates, we recommend reminding your users that their emails *may* be case-sensitive, and that the username on this site is definitely case-sensitive.

Even if email case-sensitivity becomes a problem on your site, we recommend you continue to use case-sensitive email fields so that you retain case-sensitive data. Instead, rely on case-insensitive selection and filtering to find and authenticate users (lowercase database indexes can make this quite fast). These decisions and code are outside the scope of this project and we therefore do not provide any work on this front.

CHAPTER 6

How To: Integrate Improved User Directly

Warning: This configuration method is but one of three, and may not make the most sense for your project. Please read *Select a Configuration Method for Improved User* before continuing, or else follow the instructions in *Quickstart: Using Improved User*.

In a new Django project, perform the following steps in the `settings.py` file or base settings file.

1. Add `improved_user.apps.ImprovedUserConfig` to `INSTALLED_APPS`
2. Define or replace `AUTH_USER_MODEL` with the new model, as below.

```
AUTH_USER_MODEL='improved_user.User'
```

3. In Django > 1.9, change `UserAttributeSimilarityValidator` to match correct `User` fields, as shown below.

```
AUTH_PREFIX = 'django.contrib.auth.password_validation.'
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': AUTH_PREFIX + 'UserAttributeSimilarityValidator',
        'OPTIONS': {
            'user_attributes': ('email', 'full_name', 'short_name')
        },
    },
    # include other password validators here
]
```

Note: Improved user also comes with forms, test factories, and an admin panel. Take a look at the *Package Reference* for more information.

CHAPTER 7

How To: Create a Custom User using Mixins

Warning: This configuration method is but one of three, and may not make the most sense for your project. Please read *Select a Configuration Method for Improved User* before continuing, or else follow the instructions in *Quickstart: Using Improved User*.

The `User` and `AbstractUser` classes supplied by the package are not always what you want. In some cases, they may supply fields you do not need or wish for. This tutorial demonstrates how to create `User` models using the provided mix-in classes, effectively building the model from scratch.

In this tutorial, we will create a new custom User that has an email field and password, but which does not feature either the `short_name` or `full_name` fields.

Warning: Not supplying methods for names on the User model will cause problems with Django's Admin.

Tip: If you're looking to extend the `User` model, rather than replace it as shown in this tutorial, use the following steps:

1. inherit `AbstractUser` (follow the instructions in *Quickstart: Using Improved User* to see how)
2. add new fields as desired
3. override `REQUIRED_FIELDS` if necessary (remembering to put `'short_name', 'full_name'` in the list)

In an existing app, in the `models.py` file, we start by importing the tools we need to build the model. We first import classes from Django.

```
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin
from django.utils.translation import ugettext_lazy as _
```

`AbstractBaseUser` and `PermissionsMixin` will serve as a base for the User (click the classes in this sentence to see Django's official documentation on the subject). We also import `ugettext_lazy()` to enable translation of our strings.

We then import mix-in classes from Improved User.

```
from improved_user.managers import UserManager
from improved_user.model_mixins import DjangoIntegrationMixin, EmailAuthMixin
```

The *DjangoIntegrationMixin* class provides fields that allow the model to integrate with Django's default Authentication Backend as well as a field to allow for integration with Django's Admin.

The *EmailAuthMixin* creates an `EmailField` and sets the field to be used as the username during the authentication process.

The *UserManager* is a custom model manager that provides the *create_user()* and *create_superuser()* methods used in Django.

Danger: Improved Users' custom *UserManager* is intended to work with subclasses of *EmailAuthMixin*, and will likely not work with your User subclass if you are using a different field for your username. You will, in that case, need to create your own `UserManager`. The source code for Improved Users' *UserManager* as well as Django's `BaseUserManager` and `UserManager` would likely prove helpful.

Note: If you wanted to create a User model with a field other than email for username, you would set the `USERNAME_FIELD` on your User model to the name of the field that should serve as the username. Please take a look at the source of *EmailAuthMixin* for an example of this.

With all our tools in place, we can now create a User model. We start by creating a class that inherits all of the classes we have imported, and then we tie the *UserManager* to the new model.

```
class User(DjangoIntegrationMixin, EmailAuthMixin,
           PermissionsMixin, AbstractBaseUser):
    """A user created using mix-ins from Django and improved-user

    Note that the lack of name methods will cause errors in the Admin
    """
    objects = UserManager()
```

For good measure, we can specify the name and verbose name of the model, making sure to internationalize the strings. Our full and final `models.py` file is shown below.

```
"""A User model created by django-improved-user mixins"""
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin
from django.utils.translation import ugettext_lazy as _

from improved_user.managers import UserManager
from improved_user.model_mixins import DjangoIntegrationMixin, EmailAuthMixin


class User(DjangoIntegrationMixin, EmailAuthMixin,
           PermissionsMixin, AbstractBaseUser):
    """A user created using mix-ins from Django and improved-user

    Note that the lack of name methods will cause errors in the Admin
```

(continues on next page)

(continued from previous page)

```
    """
    objects = UserManager()

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

Tip: Setting `abstract = True` in the `Meta` class would allow the class above to be an AbstractUser model similar to *`AbstractUser`*

For all of the classes you may use to create your own `User` model, please see *`model_mixins`*.

CHAPTER 8

How To: Use Improved User in Data Migrations

Creating users in data migrations is **discouraged** as doing so represents a potential **security risk**, as passwords are stored in plaintext in the migration. However, doing so in proof-of-concepts or in special cases may be necessary, and the steps below will demonstrate how to create and remove new users in a Django data migration.

The `django-improved-user` package intentionally disallows use of *`UserManager`* in data migrations (we forgo the use of model managers in migrations). The *`create_user()`* and *`create_superuser()`* methods are thus both unavailable when using data migrations. Both of these methods rely on *`User`* model methods which are unavailable in Historical models, so we could not use them even if we wanted to (short of refactoring large parts of code currently inherited by Django).

We therefore rely on the standard `Manager`, and supplement the password creation behavior.

In an existing Django project, you will start by creating a new and empty migration file. Replace `APP_NAME` in the command below with the name of the app for which you wish to create a migration.

```
$ python manage.py makemigrations --empty --name=add_user APP_NAME
```

We start by importing the necessary tools

```
from django.conf import settings
from django.contrib.auth.hashers import make_password
from django.db import migrations
```

We will use `RunPython` to run our code. `RunPython` expects two functions with specific parameters. Our first function creates a new user.

```
def add_user(apps, schema_editor):
    User = apps.get_model(*settings.AUTH_USER_MODEL.split('.'))
    User.objects.create(
        email='migrated@jambonsw.com',
        password=make_password('s3cr3tp4ssw0rd!'),
        short_name='Migrated',
        full_name='Migrated Improved User',
    )
```

NB: Due to the lack of *UserManager* or *User* methods, the `email` field is not validated or normalized. What's more, the `password` field is not validated against the project's password validators. **It is up to the developer coding the migration file to provide proper values.**

The second function is technically optional, but providing one makes our lives easier and is considered best-practice. This function undoes the first, and deletes the user we created.

```
def remove_user(apps, schema_editor):
    User = apps.get_model(*settings.AUTH_USER_MODEL.split('.'))
    User.objects.get(email='migrated@jambonsw.com').delete()
```

Finally, we use our migration functions via `RunPython` in a `django.db.migrations.Migration` subclass. Please note the *addition* of the dependency below. If your file already had a dependency, please add the tuple below, but do not remove the existing tuple(s).

```
class Migration(migrations.Migration):

    dependencies = [
        ('improved_user', '0001_initial'),
    ]

    operations = [
        migrations.RunPython(add_user, remove_user),
    ]
```

The final migration file is printed in totality below.

```
from django.conf import settings
from django.contrib.auth.hashers import make_password
from django.db import migrations


def add_user(apps, schema_editor):
    User = apps.get_model(*settings.AUTH_USER_MODEL.split('.'))
    User.objects.create(
        email='migrated@jambonsw.com',
        password=make_password('s3cr3tp4ssw0rd!'),
        short_name='Migrated',
        full_name='Migrated Improved User',
    )


def remove_user(apps, schema_editor):
    User = apps.get_model(*settings.AUTH_USER_MODEL.split('.'))
    User.objects.get(email='migrated@jambonsw.com').delete()


class Migration(migrations.Migration):

    dependencies = [
        ('improved_user', '0001_initial'),
    ]

    operations = [
        migrations.RunPython(add_user, remove_user),
    ]
```

You may wish to read more about Django Data Migrations and `RunPython`.

CHAPTER 9

How To: Use the Django Admin with Improved User

Django Improved User defines an admin panel for the *User* model provided by the package.

The admin panel is used automatically if you are integrating directly with the package (see *Select a Configuration Method for Improved User* for more information about different uses, and *How To: Integrate Improved User Directly* for instructions on direct integration).

If you are extending the User model with **no changes** (as shown in the *Quickstart: Using Improved User*), you can simply import the existing admin panel and use it in your own project.

```
"""Demonstrate use of UserAdmin on extended User model"""
from django.contrib import admin
from django.contrib.auth import get_user_model

from improved_user.admin import UserAdmin

User = get_user_model()  # pylint: disable=invalid-name
# WARNING
# This works, but note that any additional fields do not appear in the
# Admin.  For instance, the User model in this example has a verified
# boolean field added to it, but this field will not appear in the
# admin.  Additionally, if the verified field did not have a default,
# creating the User model via the admin panel would be impossible. As
# such, do not use this method in production applications, and instead
# define your own UserAdmin class.
admin.site.register(User, UserAdmin)
```

As noted in the comment in the file above, this method is not desirable in production contexts. Additionally, it will not work in the event you are replacing existing fields (as shown in *How To: Create a Custom User using Mixins*).

When using the extension method on a real/production site, or when replacing existing fields, you will need to build your own admin panel. Django doesn't supply mechanisms for simple inheritance of other admin panels, and the package maintainers don't know what fields you're using, so it's impossible for us to provide an easily extendable or re-usable admin panel in these scenarios. We encourage you to look at *UserAdmin* for guidance (printed below for your convenience).

```
"""Admin Configuration for Improved User"""
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.utils.translation import ugettext_lazy as _

from .forms import UserChangeForm, UserCreationForm


class UserAdmin(BaseUserAdmin):
    """Admin panel for Improved User, mimics Django's default"""

    fieldsets = (
        (None, {"fields": ("email", "password")}),
        (_("Personal info"), {"fields": ("full_name", "short_name")}),
        (
            _("Permissions"),
            {
                "fields": (
                    "is_active",
                    "is_staff",
                    "is_superuser",
                    "groups",
                    "user_permissions",
                ),
            },
        ),
        (_("Important dates"), {"fields": ("last_login", "date_joined")}),
    )
    add_fieldsets = (
        (
            None,
            {
                "classes": ("wide",),
                "fields": ("email", "short_name", "password1", "password2"),
            },
        ),
    )
    form = UserChangeForm
    add_form = UserCreationForm
    list_display = ("email", "full_name", "short_name", "is_staff")
    search_fields = ("email", "full_name", "short_name")
    ordering = ("email",)
```

Note: To allow the class above to be imported in demo situations, the module is lacking a call to register the `UserAdmin` class. When you create your own class, you will need code similar to the snippet below.

```
from django.contrib import admin
from django.contrib.auth import get_user_model

User = get_user_model()
admin.site.register(User, NewUserAdmin)
```

CHAPTER 10

How to Contribute

First off, thanks for taking the time to contribute!

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. The following is a set of guidelines for contributing to django-improved-user, hosted on Github. These are mostly guidelines, not rules. Use your best judgment, and feel free to propose changes to this document in a pull request.

Please remember that this is a volunteer-driven project. We will look at the issues and pull requests as soon as possible.

10.1 Code of Conduct

This project is subject to a Code of Conduct. By participating, you are expected to uphold this code.

Please be respectful to other developers.

10.2 Types of Contributions

You can contribute in many ways:

10.2.1 Report Bugs

Please report bugs on the Github issue tracker. Search the tracker to make sure someone else hasn't already reported the issue. If you find your the problem has already been reported, feel free to add more information if appropriate. If you don't find the problem reported, please open a new issue, and follow the guidelines set forth in the text field.

10.2.2 Fix Bugs

Look through the Github issue tracker for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it. If someone has been assigned, or notes that it is claimed in the comments, please reach out to them to work together on the issue to avoid duplicating work. Note that, as volunteers, people sometime are unable to complete work they start, and that it is reasonable after a certain amount of time to assume they are no longer working on the issue. Use your best judgment to assess the situation.

10.2.3 Write (or Request) Documentation

The documentation aims to provide reference material, how-to guides, and a general tutorial for getting started with Django and django-improved-user. If you believe the documentation can be expanded or added to, your contribution would be welcomed.

If you are running into a problem, and believe that some documentation could clarify the problem (or the solution!) please feel free to request documentation on the Github issue tracker.

For more about different kinds of documentations and how to think about the differences, please watch Daniele Procida's PyCon US 2017 talk on the subject.

10.3 Your First Contribution

Ready to contribute? Let's get django-improved-user working on your local machine.

This package relies on Python, pip, and Django. Please make sure you have the first two installed.

To get started, fork the git repository to your own account using the fork button on the top right of the Github interface. You now have your own fork of the project! Clone your fork of the repository using the command below, but with your own username.

```
$ git clone git@github.com:YOUR_USERNAME/django-improved-user.git
```

We recommend the use of virtual environments when developing (generally). If you are not familiar with virtual environments, take a look at Python's venv documentation. Virtualenvwrapper is also a favorite.

You can now install all of the dependencies required to develop the project. Use pip to install all dependencies, as demonstrated below.

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

If you are modifying code, keep reading. If you are changing documentation, skip to the next section.

10.3.1 Your First Code Contribution

Before making any changes, let's first make sure all the tests pass. To run the test suite on a single version of Django, you will need to install Django and the package (in development mode). Use the command below to do both.

```
$ python setup.py develop
```

Run the *runtests.py* script from the root of the project to test the django-improved-user project.

```
$ python runtests.py
```

You can limit the tests or pass paramaters as if you had called Django's *manage.py test*.

```
$ ./runtests.py tests.test_basic -v 3
```

If you have Python 3.4, 3.5, and 3.6 installed on your system, you will be able to test the package under all required conditions. The project uses *tox* to make this easy. This will use all the linters and test the package with multiple Python and Django versions.

```
$ tox
```

Note that any change made to this project must meet the linting rules and tests run by tox. These rules are double-checked by TravisCI and AppVeyor. Furthermore, changes in code must maintain or increase code-coverage unless this is unreasonable.

If your tests all pass, you are ready to make changes! If not, please open an issue in Github detailing the test failure you are seeing.

Create a new branch in the repository. Name the branch descriptively, and reference the the github issue if applicable. Below are a few examples of what that command might look like.

```
$ git checkout -b add_how_to_subclass_abstract_user_guide
$ git checkout -b issue_45_allow_whitespace_in_passwords
```

Please note that all pull requests that feature code changes are expected to reference github issues, as discussion is required for any change.

Make your changes! We recommend a test-driven approach to development. Please remember to update any relevant documentation. Make your commits small, and target each commit to do a single thing. If you are comfortable rebasing git commits, please do so at the end - providing small, targeted, organized commits can make reviewing code radically easier, and we will be grateful for it.

Once you are done, push your changes to github, and open a pull request via the interface. Please follow all of the instructions in the pull request textfield when doing so, as it will help us understand and review your code.

Congratulations on opening a pull request!

10.3.2 Your First Documentation Contribution

> If it isn't documented, it doesn't exist.
>
> —Mike Pope

Documentation is crucial, and I am thrilled to get your help writing it!

All of the documentation is written in reStructuredText, sometimes called *rst*. Some of the documents (such as this one!) are in the root of the Github project, but the vast majority exist in the `docs` directory. The documents found in this directory are compiled to HTML by Sphinx (which has a primer on rst).

You may use the `Makefile` in the `docs` directory to run Sphinx.

```
$ cd docs
$ make clean && make html
```

If you browse to `_build/html` (within the `docs` directory), you'll find a local build of all the documentation! Open any of the HTML files in a browser to read the documentation.

Alternatively, you can use `tox` to build the documentation (requires that Python 3.6 be installed). This is more of a check, as navigating to the built files is less easy.

```
$ tox -e docs
```

The documentation automatically builds reference documentation for the project. To update these reference documents, you will need to change the Python docstrings in the code itself. Corrections and expansions to existing docs, as well as new tutorials and how-to guides are welcome additions. If you had a pain point while using this project, and you would like to add to an existing document or else to write a new one, you are encouraged to do it!

If you run into an problems or have a question, please ask it on the Github issue tracker (after making sure someone hasn't already asked and answered the question!).

Once you have made changes to the documents in question, you'll want to make sure that Sphinx builds the documentation without any errors.

Commit your changes, and push them to your local branch. Using the Github interface, open a pull request to the development branch in the main repository! Please follow all of the instructions in the pull request textfield when doing so, as it will help us understand and review your code.

Congratulations on opening a pull request!

CHAPTER 11

Package Reference

In this Document

11.1 Overview

Django Improved User is organized like a regular Django app.

class improved_user.apps.**ImprovedUserConfig**
Reference this class in INSTALLED_APPS to use the package.

The package provides both a concerete *User* model, as well as mix-in and abstract model classes to be used to extend the model or replace it entirely. Please refer to *Select a Configuration Method for Improved User* for more information about how to configure these models to best suit your purposes.

The package also provides forms, test factories, and an admin panel. Please see the reference documentation for these items below.

Finally, the actual code on Github has three example projects that may be helpful if this documentation was not.

11.2 Reference Documents

11.2.1 Improved User Model

class improved_user.models.**User**(*email*, *password*, *short_name=None*, *full_name=None*)
Bases: *improved_user.model_mixins.AbstractUser*

The Improved User Model is intended to be used out-of-the-box.

Do **not** import this model directly: use `get_user_model()`.

Parameters

- **id** (*AutoField*) – Id
- **date_joined** (*DateTimeField*) – Date joined
- **email** (*EmailField*) – Email address
- **full_name** (*CharField*) – Full name
- **groups** (*ManyToManyField*) – The groups this user belongs to. A user will get all permissions granted to each of their groups.
- **is_active** (*BooleanField*) – Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
- **is_staff** (*BooleanField*) – Designates whether the user can log into the admin site.
- **is_superuser** (*BooleanField*) – Designates that this user has all permissions without explicitly assigning them.
- **last_login** (*DateTimeField*) – Last login
- **password** (*CharField*) – Password
- **short_name** (*CharField*) – Short name
- **user_permissions** (*ManyToManyField*) – Specific permissions for this user.

check_password(*raw_password*)
Return a boolean of whether the raw_password was correct. Handles hashing formats behind the scenes.

clean()
Override default clean method to normalize email.

Call `super().clean()` if overriding.

email_user(*subject*, *message*, *from_email=None*, ***kwargs*)
Sends an email to this User.

get_full_name()
Returns the full name of the user.

get_short_name()
Returns the short name for the user.

get_username()
Return the identifying username for this User

has_module_perms(*app_label*)
Return True if the user has any permissions in the given app label. Use similar logic as has_perm(), above.

has_perm(*perm*, *obj=None*)
Return True if the user has the specified permission. Query all available auth backends, but return immediately if any backend returns True. Thus, a user who has permission from a single auth backend is assumed to have permission in general. If an object is provided, check permissions for that object.

has_perms(*perm_list*, *obj=None*)
Return True if the user has each of the specified permissions. If object is passed, check if the user has all required perms for it.

is_anonymous
Always return False. This is a way of comparing User objects to anonymous users.

is_authenticated
Always return True. This is a way to tell if the user has been authenticated in templates.

refresh_from_db(*using=None*, *fields=None*)
Reload field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

11.2.2 Managers

class improved_user.managers.**UserManager**
Manager for Users; overrides create commands for new fields

Meant to be interacted with via the user model.

```
User.objects  # the UserManager
User.objects.all()  # has normal Manager/UserManager methods
User.objects.create_user  # overrides methods for Improved User
```

Set to `objects` by `AbstractUser`

create_superuser(*email*, *password*, ***extra_fields*)
Save new User with is_staff and is_superuser set to True

create_user(*email=None*, *password=None*, ***extra_fields*)
Save new User with email and password

11.2.3 Mix-in Model Classes

These classes are provided as tools to help build your own User models.

- *AbstractUser*
- *DjangoIntegrationMixin*
- *EmailAuthMixin*
- *FullNameMixin*
- *ShortNameMixin*

AbstractUser

class improved_user.model_mixins.**AbstractUser**(**args*, ***kwargs*)
Bases: *improved_user.model_mixins.DjangoIntegrationMixin*, *improved_user.model_mixins.FullNameMixin*, *improved_user.model_mixins.ShortNameMixin*, *improved_user.model_mixins.EmailAuthMixin*, django.contrib.auth.models.PermissionsMixin, django.contrib.auth.base_user.AbstractBaseUser

An abstract base class meant to be inherited (do not instantiate this). The class provides a fully featured User model with admin-compliant permissions. Differs from Django's `AbstractUser`:

1. Login occurs with an email and password instead of username.
2. Provides short_name and full_name instead of first_name and last_name.

All fields other than email and password are optional.

Sets `objects` to *`UserManager`*.

Documentation about Django's `AbstractBaseUser` may be helpful in understanding this class.

Parameters

- **`date_joined`** (*`DateTimeField`*) – Date joined
- **`email`** (*`EmailField`*) – Email address
- **`full_name`** (*`CharField`*) – Full name
- **`groups`** (*`ManyToManyField`*) – The groups this user belongs to. A user will get all permissions granted to each of their groups.
- **`is_active`** (*`BooleanField`*) – Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
- **`is_staff`** (*`BooleanField`*) – Designates whether the user can log into the admin site.
- **`is_superuser`** (*`BooleanField`*) – Designates that this user has all permissions without explicitly assigning them.
- **`last_login`** (*`DateTimeField`*) – Last login
- **`password`** (*`CharField`*) – Password
- **`short_name`** (*`CharField`*) – Short name
- **`user_permissions`** (*`ManyToManyField`*) – Specific permissions for this user.

`check_password`(*raw_password*)
Return a boolean of whether the raw_password was correct. Handles hashing formats behind the scenes.

`clean`()
Override default clean method to normalize email.

Call `super().clean()` if overriding.

`clean_fields`(*exclude=None*)
Clean all fields and raise a ValidationError containing a dict of all validation errors if any occur.

`email_user`(*subject*, *message*, *from_email=None*, ***kwargs*)
Sends an email to this User.

`full_clean`(*exclude=None*, *validate_unique=True*)
Call clean_fields(), clean(), and validate_unique() on the model. Raise a ValidationError for any errors that occur.

`get_deferred_fields`()
Return a set containing names of deferred fields for this instance.

`get_full_name`()
Returns the full name of the user.

`get_group_permissions`(*obj=None*)
Return a list of permission strings that this user has through their groups. Query all available auth backends. If an object is passed in, return only permissions matching this object.

get_session_auth_hash()
Return an HMAC of the password field.

get_short_name()
Returns the short name for the user.

get_username()
Return the identifying username for this User

has_module_perms(*app_label*)
Return True if the user has any permissions in the given app label. Use similar logic as has_perm(), above.

has_perm(*perm*, *obj=None*)
Return True if the user has the specified permission. Query all available auth backends, but return immediately if any backend returns True. Thus, a user who has permission from a single auth backend is assumed to have permission in general. If an object is provided, check permissions for that object.

has_perms(*perm_list*, *obj=None*)
Return True if the user has each of the specified permissions. If object is passed, check if the user has all required perms for it.

has_usable_password()
Return False if set_unusable_password() has been called for this user.

is_anonymous
Always return False. This is a way of comparing User objects to anonymous users.

is_authenticated
Always return True. This is a way to tell if the user has been authenticated in templates.

refresh_from_db(*using=None*, *fields=None*)
Reload field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

save(**args*, ***kwargs*)
Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Handle the parts of saving which should be done only once per save, yet need to be done in raw saves, too. This includes some sanity checks and signal sending.

The 'raw' argument is telling save_base not to save any parent models and not to do any changes to the values before save. This is used by fixture loading.

serializable_value(*field_name*)
Return the value of the field name for this instance. If the field is a foreign key, return the id value instead of the object. If there's no Field object with this name on the model, return the model attribute's value.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

validate_unique(*exclude=None*)
Check unique constraints on the model and raise ValidationError if any failed.

DjangoIntegrationMixin

class improved_user.model_mixins.**DjangoIntegrationMixin**(**args*, ***kwargs*)
Mixin provides fields for Django integration to work correctly

Provides permissions for Django Admin integration, as well as date field used by authentication code.

Parameters

- **date_joined** (*DateTimeField*) – Date joined
- **is_active** (*BooleanField*) – Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
- **is_staff** (*BooleanField*) – Designates whether the user can log into the admin site.

EmailAuthMixin

class improved_user.model_mixins.**EmailAuthMixin**(**args*, ***kwargs*)
A mixin to use email as the username

Parameters **email** (*EmailField*) – Email address

clean()
Override default clean method to normalize email.

Call `super().clean()` if overriding.

email_user(*subject*, *message*, *from_email=None*, ***kwargs*)
Sends an email to this User.

FullNameMixin

class improved_user.model_mixins.**FullNameMixin**(**args*, ***kwargs*)
A mixin to provide an optional full name field

Parameters **full_name** (*CharField*) – Full name

get_full_name()
Returns the full name of the user.

ShortNameMixin

class improved_user.model_mixins.**ShortNameMixin**(**args*, ***kwargs*)
A mixin to provide an optional short name field

Parameters **short_name** (*CharField*) – Short name

get_short_name()
Returns the short name for the user.

11.2.4 Forms

Abstract forms meant to be inherited or concrete forms meant to be used direction in your views.

Note: These forms are unnecessary starting in Django 2.1, as Django now supports custom user models in its own forms.

UserCreationForm

class improved_user.forms.**UserCreationForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Bases: *improved_user.forms.AbstractUserCreationForm*

A concrete implementation of AbstractUserCreationForm that uses an e-mail address as a user's identifier.

Parameters

- **email** (*EmailField*) – Email address
- **full_name** (*CharField*) – Full name
- **short_name** (*CharField*) – Short name
- **password1** (*CharField*) – Password
- **password2** (*CharField*) – Enter the same password as above, for verification.

clean_email()

Clean email; set nice error message

Since User.email is unique, this check is redundant, but it sets a nicer error message than the ORM. See #13147.

https://code.djangoproject.com/ticket/13147

UserChangeForm

class improved_user.forms.**UserChangeForm**(**args*, ***kwargs*)

Bases: *improved_user.forms.AbstractUserChangeForm*

Form to update user, but not their password

Parameters

- **password** (*ReadOnlyPasswordHashField*) – Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

- **last_login** (*DateTimeField*) – Last login
- **is_superuser** (*BooleanField*) – Designates that this user has all permissions without explicitly assigning them.
- **groups** (*ModelMultipleChoiceField*) – The groups this user belongs to. A user will get all permissions granted to each of their groups.
- **user_permissions** (*ModelMultipleChoiceField*) – Specific permissions for this user.
- **is_staff** (*BooleanField*) – Designates whether the user can log into the admin site.
- **is_active** (*BooleanField*) – Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
- **date_joined** (*DateTimeField*) – Date joined
- **full_name** (*CharField*) – Full name
- **short_name** (*CharField*) – Short name
- **email** (*EmailField*) – Email address

AbstractUserCreationForm

class improved_user.forms.**AbstractUserCreationForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Bases: django.forms.models.ModelForm

A form that creates a user, with no privileges, from the given username and password.

Parameters

- **password1** (*CharField*) – Password
- **password2** (*CharField*) – Enter the same password as above, for verification.

_post_clean()

Run password validaton after clean methods

When clean methods are run, the user instance does not yet exist. To properly compare model values agains the password (in the UserAttributeSimilarityValidator), we wait until we have an instance to compare against.

https://code.djangoproject.com/ticket/28127 https://github.com/django/django/pull/8408

Has no effect in Django prior to 1.9 May become unnecessary in Django 2.0 (if this superclass changes)

clean_password2()

Check wether password 1 and password 2 are equivalent

While ideally this would be done in clean, there is a chance a superclass could declare clean and forget to call super. We therefore opt to run this password mismatch check in password2 clean, but to show the error above password1 (as we are unsure whether password 1 or password 2 contains the typo, and putting it above password 2 may lead some users to believe the typo is in just one).

save(*commit=True*)
Save the user; use password hasher to set password

AbstractUserChangeForm

class improved_user.forms.**AbstractUserChangeForm**(**args*, ***kwargs*)
Bases: django.forms.models.ModelForm

Base form update User, but not their password

Parameters **password** (*ReadOnlyPasswordHashField*) – Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

clean_password()
Change user info; not the password

We seek to change the user, but not the password. Regardless of what the user provides, return the initial value. This is done here, rather than on the field, because the field does not have access to the initial value

get_local_password_path()
Method to return relative path to password form

Will return rel_password_url attribute on form or else '../password/'. If subclasses cannot simply replace rel_password_url, then they can override this method instead of __init__.

11.2.5 Test Factories

Factories to make testing with Improved User easier

class improved_user.factories.**UserFactory**
Bases: factory.django.DjangoModelFactory

Factory Boy factory for Improved User

Generates a user with a default password of password!. The user is active, but is not staff or a superuser. Any value can be overridden by passing in a value, as shown below.

```
UserFactory(
    password='mys3cr3tp4ssw0rd!',
    is_superuser=True,
)
```

11.2.6 Django Admin Panel

Admin Configuration for Improved User

class improved_user.admin.**UserAdmin**(*model*, *admin_site*)
Admin panel for Improved User, mimics Django's default

add_form
alias of *improved_user.forms.UserCreationForm*

form
alias of *improved_user.forms.UserChangeForm*

CHAPTER 12

History

12.1 Next Release

- Nothing Yet!

12.2 1.0.1 (2020-02-16)

- Add flexibility to admin panel usage; document usage

12.3 1.0.0 (2018-07-28)

- Django 1.8, 1.11, 2.0, 2.1 officially supported.
- Django 1.9 and 1.10 are not tested against, as Django does not support them, but they likely work.
- **Breaking change**: Model mix-in classes now exist in their own module! Import from *model_mixins* instead of *models*. (#46, #96)
- Fix issue #49: allow form classes to be imported without requiring project to be in `INSTALLED_APPS` (See #36 and #46 below for associated error and reasoning) (#50)
- Fix issue #36: refactor package to allow for mix-in classes to be imported into third-party project without requiring project to be in `INSTALLED_APPS` (which would unnecessarily create unused tables in the project). Add documentation/tutorial on subject. (#46)
- Django 2.0, 2.1 compatibility. (#43, #93)

12.4 0.5.3 (2017-08-29)

- Include history of changes in online documentation. (#34)

- Write documentation about why and how the project was built. (#34)
- Add section about contributing documentation. (#34)

12.5 0.5.2 (2017-08-27)

- Change package PyPI license identifier for better information on djangopackages.org detail page. See djangopackages/djangopackages#483 for more information.

12.6 0.5.1 (2017-08-27)

- Docfix: Remove links to ReadTheDocs Stable version from ReadMe, as we are unable to build that version until v1.0.0 release. See rtfd/readthedocs.org#2032 for more information. (#31)

12.7 0.5.0 (2017-08-26)

- Provide documentation for the package. This includes Sphinx documentation hosted on ReadTheDocs.org, (#26, #29), but also documents to help contribute to github more easily (#26) as well as a code of conduct (#26). The Read Me includes badges (#26).
- In the event the documentation isn't enough, the project now includes an example project demonstrating integration of django-improved-user with Django as well as django-registration. (#28) This content is used to create some of the documentation (#29).
- Bugfix: The `UserManager` was setting the `last_login` attribute of new users at creation time. Reported in #25, fixed in #27 (`last_login` is `None` until the user actually logs in).

12.8 0.4.0 (2017-08-14)

Warning: This is a **breaking change**, and migrations will conflict with v0.3.0 due to PR #23

- Add `UserFactory` to make testing easier for developers using the package; requires factory_boy (PR #20)
- Split the `ImprovedIdentityMixin` class into atomic parts: `DjangoIntegrationMixin`, `FullNameMixin`, `ShortNameMixin`, `EmailAuthMixin`. This allows developers to create their own custom `AbstractUsers` if needed. (PR #22)
- Change `blank` to `True` on `short_name` field of User model. (**Breaking change!** PR #23).

12.9 0.3.0 (2017-08-10)

- Integrate coverage and codecov service (PR #16)
- Make TravisCI test builds public (first seen in PR #16)
- Merge appropriate tests from Django master (1.11.3 is current release at time of writing). This increases test coverage across the board and updates the test suite to check for parity between Django's User API and Improved User's API as well as check for the same security issues. (PR #18)
- UserManager raises a friendly error if the developer tries to pass a username argument (PR #18)

- Password errors are shown above both password fields (PR #18)
- Bugfix: UserManager handles is_staff, is_active, and is_superuser correctly (PR #18)
- Bugfix: User has email normalized during Model.clean phase (PR #18)
- Bugfix: UserAdmin requires short_name in both add and change (previously only in change; PR #18)
- Bugfix: UserAdmin uses correct relative path URL for password change in all versions of Django (was not working in Django 1.9+) (PR #18)
- Bugfix: Runtests correctly handles test specification (PR #18)

12.10 0.2.0 (2017-07-30)

- Reorganize project to follow best practices (PR #9)
- Allow setup.py to run tests by overriding test command (PR #9)
- Test locally with Tox (PR #10)
- Remove Django 1.9 from supported versions (PR #10)
- Enforce styleguide with flake8, isort, and pylint. Use flake8-commas and flake8-quotes to enhance flake8. Override default distutils check command to check package metadata. Use check-manifest to check contents of MANIFEST.in (PR #11)
- Integrate https://pyup.io/ into project (PR #12)
- Upgrade flake8 to version 3.4.1 (PR #13)
- Make release and distribution less painful with bumpversion package and a Makefile (PR #15)
- Add HISTORY.rst file to provide change log (PR #15)

12.11 0.1.1 (2017-06-28)

- Fix metadata in setup.py for warehouse (see https://github.com/pypa/warehouse/issues/2155 and PR #8)

12.12 0.1.0 (2017-06-28)

- Add tests for Django 1.11 (PR #5)
- Allow for integration with UserAttributeSimilarityValidator (see https://code.djangoproject.com/ticket/28127, https://github.com/django/django/pull/8408, and PR #5)
- Rename project django-improved-user (from django-simple-user)
- Make development default branch (PR #6)
- Initial public release (PR #7)
- Use Simplified BSD License instead of Revised BSD License (#7)

12.13 0.0.1 (2016-10-26)

- Simplified User model for better international handling. Includes forms and admin configuration (PR #1)
- All tests run on TravisCI (PR #3)
- **Compatible with:**
  - Python 3.4, 3.5, 3.6
  - Django 1.8 through 1.10 (PR #3 and #4)

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

Symbols

A

C

D

E

F

G

H

I

R

S

U

V