# django-registration Documentation

***Release 3.1.1***

**James Bennett**

**Sep 21, 2020**

# Installation and configuration

django-registration is an extensible application providing user registration functionality for Django-powered Web sites.

Although nearly all aspects of the registration process are customizable, out-of-the-box support is provided for two common use cases:

- Two-phase registration, consisting of initial signup followed by a confirmation email with instructions for activating the new account.
- One-phase registration, where a user signs up and is immediately active and logged in.

To get up and running quickly, *install django-registration*, then read *the quick start guide*, which describes the steps necessary to configure django-registration for the built-in workflows. For more detailed information, including how to customize the registration process (and support for alternate registration systems), read through the documentation listed below.

CHAPTER 1

# Installation guide

The 3.1.1 release of django-registration supports Django 2.2, 3.0, and 3.1 on the following Python versions:

- Django 2.2 supports Python 3.5, 3.6, 3.7, and 3.8.
- Django 3.0 and 3.1 support Python 3.6, 3.7, and 3.8.

## 1.1 Normal installation

The preferred method of installing django-registration is via *pip*, the standard Python package-installation tool. If you don't have *pip*, instructions are available for how to obtain and install it, though if you're using a supported version of Python, *pip* should have come bundled with your installation of Python.

Once you have *pip*, type:

```
pip install django-registration
```

If you don't have a copy of a compatible version of Django, this will also automatically install one for you, and will install a third-party library required by some of django-registration's validation code.

## 1.2 Installing from a source checkout

If you want to work on django-registration, you can obtain a source checkout.

The development repository for django-registration is at <https://github.com/ubernostrum/django-registration>. If you have git installed, you can obtain a copy of the repository by typing:

```
git clone https://github.com/ubernostrum/django-registration.git
```

From there, you can use git commands to check out the specific revision you want, and perform an "editable" install (allowing you to change code as you work on it) by typing:

```
pip install -e .
```

## 1.3 Next steps

To get up and running quickly, check out *the quick start guide*. For full documentation, see *the documentation index*.

CHAPTER 2

# Quick start guide

First you'll need to have Django and django-registration installed; for details on that, see *the installation guide*.

The next steps will depend on which registration workflow you'd like to use. There are two workflows built into django-registration:

- *The two-step activation workflow*, which implements a two-step process: a user signs up, then is emailed an activation link and must click it to activate the account.
- *The one-step workflow*, in which a user signs up and their account is immediately active and logged in.

If you want a signup process other than what's provided by these built-in workflows, please see the documentation for the base *view* and *form* classes, which you can subclass to implement your own preferred user registration flow and rules. The guide below covers use of the built-in workflows.

Regardless of which registration workflow you choose to use, you should add *"django_registration"* to your `INSTALLED_APPS` setting.

**Important: Django's authentication system must be installed**

Before proceeding with either of the recommended built-in workflows, you'll need to ensure *django.contrib.auth* has been installed (by adding it to `INSTALLED_APPS` and running *manage.py migrate* to install needed database tables). Also, if you're making use of a custom user model, you'll probably want to pause and read *the custom user compatibility guide* before using django-registration.

**Note: Additional steps for account security**

While django-registration does what it can to secure the user signup process, its scope is deliberately limited; please read *the security documentation* for recommendations on steps to secure user accounts beyond what django-registration alone can do.

## 2.1 Configuring the two-step activation workflow

The configuration process for using the two-step activation workflow is straightforward: you'll need to specify a couple of settings, connect some URLs and create a few templates.

### 2.1.1 Required settings

Begin by adding the following setting to your Django settings file:

***ACCOUNT_ACTIVATION_DAYS*** This is the number of days users will have to activate their accounts after registering. If a user does not activate within that period, the account will remain permanently inactive unless a site administrator manually activates it.

For example, you might have something like the following in your Django settings:

```
ACCOUNT_ACTIVATION_DAYS = 7 # One-week activation window
```

### 2.1.2 Setting up URLs

Each bundled registration workflow in django-registration includes a Django URLconf which sets up URL patterns for *the views in django-registration*. The URLconf for the two-step activation workflow can be found at *django_registration.backends.activation.urls*. For example, to place the registration URLs under the prefix */accounts/*, you could add the following to your project's root URLconf:

```
from django.urls import include, path

urlpatterns = [
    # Other URL patterns ...
    path('accounts/', include('django_registration.backends.activation.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
    # More URL patterns ...
]
```

Users would then be able to register by visiting the URL */accounts/register/*, log in (once activated) at */accounts/login/*, etc.

The sample URL configuration above also sets up the built-in auth views included in Django (login, logout, password reset, etc.) via the *django.contrib.auth.urls* URLconf.

The following URL names are defined by *django_registration.backends.activation.urls*:

- *django_registration_register* is the account-registration view.
- *django_registration_complete* is the post-registration success message.
- *django_registration_activate* is the account-activation view.
- *django_registration_activation_complete* is the post-activation success message.
- *django_registration_disallowed* is a message indicating registration is not currently permitted.

### 2.1.3 Required templates

You will also need to create several templates required by django-registration, and possibly additional templates required by views in *django.contrib.auth*. The templates required by django-registration are as follows; note that, with

the exception of the templates used for account activation emails, all of these are rendered using a `RequestContext` and so will also receive any additional variables provided by context processors.

#### *django_registration/registration_form.html*

Used to show the form users will fill out to register. By default, has the following context:

***form*** The registration form. This will likely be a subclass of *`RegistrationForm`*; consult Django's forms documentation for information on how to display this in a template.

#### *django_registration/registration_complete.html*

Used after successful completion of the registration form. This template has no context variables of its own, and should inform the user that an email containing account-activation information has been sent.

#### *django_registration/registration_closed.html*

Used when registration of new user accounts is disabled. This template has no context variables of its own.

#### *django_registration/activation_failed.html*

Used if account activation fails. Has the following context:

***activation_error*** A `dict` containing the information supplied to the *`ActivationError`* which occurred during activation. See the documentation for that exception for a description of the keys, and the documentation for *`ActivationView`* for the specific values used in different failure situations.

#### *django_registration/activation_complete.html*

Used after successful account activation. This template has no context variables of its own, and should inform the user that their account is now active.

#### *django_registration/activation_email_subject.txt*

Used to generate the subject line of the activation email. Because the subject line of an email must be a single line of text, any output from this template will be forcibly condensed to a single line before being used. This template has the following context:

***activation_key*** The activation key for the new account, as a string.

***expiration_days*** The number of days remaining during which the account may be activated, as an integer.

***request*** The `HttpRequest` object representing the request in which the user registered.

***scheme*** The protocol scheme used during registration, as a string; will be either *'http'* or *'https'*.

***site*** An object representing the site on which the user registered; depending on whether *django.contrib.sites* is installed, this may be an instance of either `django.contrib.sites.models.Site` (if the sites application is installed) or `django.contrib.sites.requests.RequestSite` (if not). Consult the documentation for the Django sites framework for details regarding these objects' interfaces.

***user*** The newly-created user object.

#### *django_registration/activation_email_body.txt*

Used to generate the body of the activation email. Should display a link the user can click to activate the account. This template has the following context:

***activation_key*** The activation key for the new account, as a string.

***expiration_days*** The number of days remaining during which the account may be activated, as an integer.

***request*** The `HttpRequest` object representing the request in which the user registered.

***scheme*** The protocol scheme used during registration, as a string; will be either *'http'* or *'https'*.

***site*** An object representing the site on which the user registered; depending on whether *django.contrib.sites* is installed, this may be an instance of either `django.contrib.sites.models.Site` (if the sites application is installed) or `django.contrib.sites.requests.RequestSite` (if not). Consult the documentation for the Django sites framework for details regarding these objects.

***user*** The newly-created user object.

Note that the templates used to generate the account activation email use the extension *.txt*, not *.html*. Due to widespread antipathy toward and interoperability problems with HTML email, django-registration produces plain-text email, and so these templates should output plain text rather than HTML.

To make use of the views from *django.contrib.auth* (which are set up for you by the example URL configuration above), you will also need to create the templates required by those views. Consult the documentation for Django's authentication system for details regarding these templates.

## 2.2 Configuring the one-step workflow

Also included is a *one-step registration workflow*, where a user signs up and their account is immediately active and logged in.

You will need to configure URLs to use the one-step workflow; the easiest way is to `include()` the URLconf *django_registration.backends.one_step.urls* somewhere in your URL configuration. For example, to place the URLs under the prefix */accounts/* in your URL structure:

```
from django.urls import include, path

urlpatterns = [
    # Other URL patterns ...
    path('accounts/', include('django_registration.backends.one_step.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
    # More URL patterns ...
]
```

Users could then register accounts by visiting the URL */accounts/register/*.

The following URL names are defined by *django_registration.backends.one_step.urls*:

- *django_registration_register* is the account-registration view.
- *django_registration_complete* is the post-registration success message.
- *django_registration_disallowed* is a message indicating registration is not currently permitted.

This URLconf will also configure the appropriate URLs for the rest of the built-in *django.contrib.auth* views (log in, log out, password reset, etc.).

Finally, you will need to create following templates:

- *django_registration/registration_form.html*
- *django_registration/registration_closed.html*

See *the documentation above* for details of these templates.

To make use of the views from *django.contrib.auth* (which are set up for you by the example URL configuration above), you will also need to create the templates required by those views. Consult the documentation for Django's authentication system for details regarding these templates.

CHAPTER 3

# The two-step activation workflow

The two-step activation workflow, found in *django_registration.backends.activation*, implements a two-step registration process: a user signs up, an inactive account is created, and an email is sent containing an activation link which must be clicked to make the account active.

## 3.1 Behavior and configuration

A default URLconf is provided, which you can `include()` in your URL configuration; that URLconf is *django_registration.backends.activation.urls*. For example, to place user registration under the URL prefix */accounts/*, you could place the following in your root URLconf:

```
from django.urls import include, path

urlpatterns = [
    # Other URL patterns ...
    path('accounts/', include('django_registration.backends.activation.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
    # More URL patterns ...
]
```

That also sets up the views from *django.contrib.auth* (login, logout, password reset, etc.).

This workflow makes use of up to three settings (click for details on each):

- *ACCOUNT_ACTIVATION_DAYS*
- *REGISTRATION_OPEN*
- *REGISTRATION_SALT* (see also *note below*)

By default, this workflow uses *RegistrationForm* as its form class for user registration; this can be overridden by passing the keyword argument *form_class* to the registration view.

## 3.2 Views

Two views are provided to implement the signup/activation process. These subclass *the base views of django-registration*, so anything that can be overridden/customized there can equally be overridden/customized here. There are some additional customization points specific to this implementation, which are listed below.

For an overview of the templates used by these views (other than those specified below), and their context variables, see *the quick start guide*.

**class** django_registration.backends.activation.views.**RegistrationView**

A subclass of *django_registration.views.RegistrationView* implementing the signup portion of this workflow.

Important customization points unique to this class are:

**create_inactive_user**(*form*)

Creates and returns an inactive user account, and calls *send_activation_email()* to send the email with the activation key. The argument *form* is a valid registration form instance passed from *register()*.

**Parameters** **form** (django_registration.forms.RegistrationForm) – The registration form.

**Return type** django.contrib.auth.models.AbstractUser

**get_activation_key**(*user*)

Given an instance of the user model, generates and returns an activation key (a string) for that user account.

**Parameters** **user** (*django.contrib.auth.models.AbstractUser*) – The new user account.

**Return type** str

**get_email_context**(*activation_key*)

Returns a dictionary of values to be used as template context when generating the activation email.

**Parameters** **activation_key** (*str*) – The activation key for the new user account.

**Return type** dict

**send_activation_email**(*user*)

Given an inactive user account, generates and sends the activation email for that account.

**Parameters** **user** (*django.contrib.auth.models.AbstractUser*) – The new user account.

**Return type** None

**email_body_template**

A string specifying the template to use for the body of the activation email. Default is *"django_registration/activation_email_body.txt"*.

**email_subject_template**

A string specifying the template to use for the subject of the activation email. Default is *"django_registration/activation_email_subject.txt"*. Note that, to avoid header-injection vulnerabilities, the result of rendering this template will be forced into a single line of text, stripping newline characters.

**class** django_registration.backends.activation.views.**ActivationView**

A subclass of *django_registration.views.ActivationView* implementing the activation portion of this workflow.

Errors in activating the user account will raise *ActivationError*, with one of the following values for the exception's *code*:

***"already_activated"*** Indicates the account has already been activated.

***"bad_username"*** Indicates the username decoded from the activation key is invalid (does not correspond to any user account).

***"expired"*** Indicates the account/activation key has expired.

***"invalid_key"*** Generic indicator that the activation key was invalid.

Important customization points unique to this class are:

**get_user**(*username*)
Given a username (determined by the activation key), looks up and returns the corresponding instance of the user model. If no such account exists, raises `ActivationError` as described above. In the base implementation, checks the `is_active` field to avoid re-activating already-active accounts, and raises `ActivationError` with code *already_activated* to indicate this case.

**Parameters** **username** (*str*) – The username of the new user account.

**Return type** django.contrib.auth.models.AbstractUser

**Raises** ***django_registration.exceptions.ActivationError*** – if no matching inactive user account exists.

**validate_key**(*activation_key*)
Given the activation key, verifies that it carries a valid signature and a timestamp no older than the number of days specified in the setting *ACCOUNT_ACTIVATION_DAYS*, and returns the username from the activation key. Raises `ActivationError`, as described above, if the activation key has an invalid signature or if the timestamp is too old.

**Parameters** **activation_key** (*str*) – The activation key for the new user account.

**Return type** str

**Raises** ***django_registration.exceptions.ActivationError*** – if the activation key has an invalid signature or is expired.

---

**Note: URL patterns for activation**

Although the actual value used in the activation key is the new user account's username, the URL pattern for `ActivationView` does not need to match all possible legal characters in a username. The activation key that will be sent to the user (and thus matched in the URL) is produced by `django.core.signing.dumps()`, which base64-encodes its output. Thus, the only characters this pattern needs to match are those from the URL-safe base64 alphabet, plus the colon ("*:*") which is used as a separator.

The default URL pattern for the activation view in *django_registration.backends.activation.urls* handles this for you.

---

## 3.3 How it works

When a user signs up, the activation workflow creates a new user instance to represent the account, and sets the *is_active* field to `False`. It then sends an email to the address provided during signup, containing a link to activate the account. When the user clicks the link, the activation view sets *is_active* to `True`, after which the user can log in.

The activation key is the username of the new account, signed using Django's cryptographic signing tools (specifically, `dumps()` is used, to produce a guaranteed-URL-safe value). The activation process includes verification of the signature prior to activation, as well as verifying that the user is activating within the permitted window (as specified in the setting *ACCOUNT_ACTIVATION_DAYS*, mentioned above), through use of Django's `TimestampSigner`.

## 3.4 Security considerations

The activation key emailed to the user in the activation workflow is a value obtained by using Django's cryptographic signing tools. The activation key is of the form:

```
encoded_username:timestamp:signature
```

where *encoded_username* is the username of the new account, *timestamp* is the timestamp of the time the user registered, and *signature* is an HMAC of the username and timestamp. The username and HMAC will be URL-safe base64 encoded; the timestamp will be base62 encoded.

Django's implementation uses the value of the `SECRET_KEY` setting as the key for HMAC; additionally, it permits the specification of a salt value which can be used to "namespace" different uses of HMAC across a Django-powered site.

The activation workflow will use the value (a string) of the setting *`REGISTRATION_SALT`* as the salt, defaulting to the string *"registration"* if that setting is not specified. This value does *not* need to be kept secret (only `SECRET_KEY` does); it serves only to ensure that other parts of a site which also produce signed values from user input could not be used as a way to generate activation keys for arbitrary usernames (and vice-versa).

CHAPTER 4

# The one-step workflow

As an alternative to *the two-step (registration and activation) workflow*, django-registration bundles a one-step registration workflow in *django_registration.backends.one_step*. This workflow consists of as few steps as possible:

1. A user signs up by filling out a registration form.
2. The user's account is created and is active immediately, with no intermediate confirmation or activation step.
3. The new user is logged in immediately.

## 4.1 Configuration

To use this workflow, include the URLconf *django_registration.backends.one_step.urls* somewhere in your site's own URL configuration. For example:

```
from django.urls import include, path

urlpatterns = [
    # Other URL patterns ...
    path('accounts/', include('django_registration.backends.one_step.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
    # More URL patterns ...
]
```

To control whether registration of new accounts is allowed, you can specify the setting *REGISTRATION_OPEN*.

Upon successful registration, the user will be redirected to the site's home page – the URL */*. This can be changed by subclassing `django_registration.backends.one_step.views.RegistrationView` and overriding the method *get_success_url()* or setting the attribute *success_url*. You can also do this in a URLconf. For example:

```
from django.conf.urls import include, url

from django_registration.backends.one_step.views import RegistrationView
```

(continues on next page)

(continued from previous page)

```
urlpatterns = [
    # Other URL patterns ...
    path('accounts/register/',
         RegistrationView.as_view(success_url='/profile/'),
         name='django_registration_register'),
    path('accounts/', include('django_registration.backends.one_step.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
    # More URL patterns ...
]
```

The default form class used for account registration will be *django_registration.forms.RegistrationForm*, although this can be overridden by supplying a custom URL pattern for the registration view and passing the keyword argument *form_class*, or by subclassing `django_registration.backends.one_step.views.RegistrationView` and either overriding *form_class* or implementing *get_form_class()*, and specifying the custom subclass in your URL patterns.

## 4.2 Templates

The one-step workflow uses two templates:

- *django_registration/registration_form.html*.
- *django_registration/registration_closed.html*

See *the quick start guide* for details of these templates.

CHAPTER 5

# Base view classes

In order to allow the utmost flexibility in customizing and supporting different workflows, django-registration makes use of Django's support for class-based views. Included in django-registration are two base classes which can be subclassed to implement many types of registration workflows.

The built-in workflows in django-registration provide their own subclasses of these views, and the documentation for those workflows will indicate customization points specific to those subclasses. The following reference covers useful attributes and methods of the base classes, for use in writing your own custom registration workflows.

**class** django_registration.views.**RegistrationView**

A subclass of Django's `FormView` which provides the infrastructure for supporting user registration.

Standard attributes and methods of `FormView` can be overridden to control behavior as described in Django's documentation, with the exception of *`get_success_url()`*, which must use the signature documented below.

When writing your own subclass, one method is required:

**register**(*form*)

Implement your registration logic here. *form* will be the (already-validated) form filled out by the user during the registration process (i.e., a valid instance of *`RegistrationForm`* or a subclass of it).

This method should return the newly-registered user instance, and should send the signal *`django_registration.signals.user_registered`*. Note that this is not automatically done for you when writing your own custom subclass, so you must send this signal manually.

**Parameters** **form** (`django_registration.forms.RegistrationForm`) – The registration form to use.

**Return type** django.contrib.auth.models.AbstractUser

Useful optional places to override or customize on subclasses are:

**disallowed_url**

The URL to redirect to when registration is disallowed. Can be a hard-coded string, the string resulting from calling Django's `reverse()` helper, or the lazy object produced by Django's `reverse_lazy()` helper. Default value is the result of calling `reverse_lazy()` with the URL name *'registration_disallowed'*.

**form_class**
The form class to use for user registration. Can be overridden on a per-request basis (see below). Should be the actual class object; by default, this class is *django_registration.forms.RegistrationForm*.

**success_url**
The URL to redirect to after successful registration. Can be a hard-coded string, the string resulting from calling Django's `reverse()` helper, or the lazy object produced by Django's `reverse_lazy()` helper. Can be overridden on a per-request basis (see below). Default value is *None*; subclasses must override and provide this.

**template_name**
The template to use for user registration. Should be a string. Default value is *'django_registration/registration_form.html'*.

**get_form_class**()
Select a form class to use on a per-request basis. If not overridden, will use *form_class*. Should be the actual class object.

**Return type** *django_registration.forms.RegistrationForm*

**get_success_url**(*user*)
Return a URL to redirect to after successful registration, on a per-request or per-user basis. If not overridden, will use *success_url*. Should return a value of the same type as *success_url* (see above).

**Parameters** **user** (*django.contrib.auth.models.AbstractUser*) – The new user account.

**Return type** str

**registration_allowed**()
Should indicate whether user registration is allowed, either in general or for this specific request. Default value is the value of the setting *REGISTRATION_OPEN*.

**Return type** bool

**class** django_registration.views.**ActivationView**
A subclass of Django's `TemplateView` which provides support for a separate account-activation step, in workflows which require that.

One method is required:

**activate**(**args*, ***kwargs*)
Implement your activation logic here. You are free to configure your URL patterns to pass any set of positional or keyword arguments to *ActivationView*, and they will in turn be passed to this method.

This method should return the newly-activated user instance (if activation was successful), or raise *ActivationError* (if activation was not successful).

**Return type** django.contrib.auth.models.AbstractUser

**Raises** ***django_registration.exceptions.ActivationError*** – if activation fails.

Useful places to override or customize on an *ActivationView* subclass are:

**success_url**
The URL to redirect to after successful activation. Can be a hard-coded string, the string resulting from calling Django's `reverse()` helper, or the lazy object produced by Django's `reverse_lazy()` helper. Can be overridden on a per-request basis (see below). Default value is *None*; subclasses must override and provide this.

**template_name**
The template to use after failed user activation. Should be a string. Default value is *'django_registration/activation_failed.html'*.

**get_success_url**(*user*)
Return a URL to redirect to after successful activation, on a per-request or per-user basis. If not overridden, will use *success_url*. Should return a value of the same type as *success_url* (see above).

**Parameters** **user** (*django.contrib.auth.models.AbstractUser*) – The activated user account.

**Return type** str

CHAPTER 6

# Base form classes

Several form classes are provided with django-registration, covering common cases for gathering account information and implementing common constraints for user registration. These forms were designed with django-registration's built-in registration workflows in mind, but may also be useful in other situations.

**class** `django_registration.forms.`**RegistrationForm**

A form for registering an account. This is a subclass of Django's built-in `UserCreationForm`, and has the following fields, all of which are required:

***username*** The username to use for the new account.

***email*** The email address to use for the new account.

***password1*** The password to use for the new account.

***password2*** The password to use for the new account, repeated to catch typos.

---

**Note: Validation of usernames**

Django supplies a default regex-based validator for usernames in its base `AbstractBaseUser` implementation, allowing any word character along with the following set of additional characters: ., @, +, and -.

Because it's a subclass of Django's `UserCreationForm`, *`RegistrationForm`* will inherit the base validation defined by Django. It also applies some custom validators to the username: *`ReservedNameValidator`*, and *`validate_confusables()`*.

---

---

**Note: Validation of email addresses**

django-registration applies two additional validators – *`HTML5EmailValidator`* and *`validate_confusables_email()`* – to the email address.

The HTML5 validator uses the HTML5 email-validation rule (as implemented on HTML's *input type="email"*), which is more restrictive than the email RFCs. The purpose of this validator is twofold: to match the behavior of HTML5, and to simplify django-registration's other validators. The full RFC grammar for email addresses is enormously complex despite most of its features rarely if ever being used legitimately, so disallowing those

features allows other validators to interact with a much simpler format, ensuring performance, reliability and safety.

---

**Note: Custom user models**

If you are using a custom user model, you **must** subclass this form and tell it to use your custom user model instead of Django's default user model. If you do not, django-registration will deliberately crash with an error message reminding you to do this. See *the custom user compatibility guide* for details.

---

**class** django_registration.forms.**RegistrationFormCaseInsensitive**
A subclass of *RegistrationForm* which enforces case-insensitive uniqueness of usernames, by applying *CaseInsensitiveUnique* to the username field.

---

**Note: Unicode case handling**

This form will normalize the username value to form NFKC, matching Django's default approach to Unicode normalization. it will then case fold the value, and use a case-insensitive (*iexact*) lookup to determine if a user with the same username already exists; the results of this query may depend on the quality of your database's Unicode implementation, and on configuration details. The results may also be surprising to developers who are primarily used to English/ASCII text, as Unicode's case rules can be quite complex.

---

**class** django_registration.forms.**RegistrationFormTermsOfService**
A subclass of *RegistrationForm* which adds one additional, required field:

***tos*** A checkbox indicating agreement to the site's terms of service/user agreement.

**class** django_registration.forms.**RegistrationFormUniqueEmail**
A subclass of *RegistrationForm* which enforces uniqueness of email addresses in addition to uniqueness of usernames, by applying *CaseInsensitiveUnique* to the email field.

CHAPTER 7

# Custom user models

Django's built-in auth system provides a default model for user accounts, but also supports replacing that default with a custom user model. Many projects choose to use a custom user model from the start of their development, even if it begins as a copy of the default model, in order to avoid the difficulty of migrating to a custom user model later on.

In general, django-registration will work with a custom user model, though at least some additional configuration is always required in order to do so. If you are using a custom user model, please read this document thoroughly *before* using django-registration, in order to ensure you've taken all the necessary steps to ensure support.

The process for using a custom user model with django-registration can be summarized as follows:

1. Compare your custom user model to the assumptions made by the built-in registration workflows.
2. If your user model is compatible with those assumptions, write a short subclass of `RegistrationForm` pointed at your user model, and instruct django-registration to use that form.
3. If your user model is *not* compatible with those assumptions, either write subclasses of the appropriate views in django-registration which will be compatible with your user model, or modify your user model to be compatible with the built-in views.

These steps are covered in more detail below.

## 7.1 Compatibility of the built-in workflows with custom user models

Django provides a number of helpers to make it easier for code to generically work with custom user models, and django-registration makes use of these. However, the built-in registration workflows must still make *some* assumptions about the structure of your user model in order to work with it. If you intend to use one of django-registration's built-in registration workflows, please carefully read the appropriate section to see what it expects from your user model.

### 7.1.1 The two-step activation workflow

*The two-step activation workflow* requires that the following be true of your user model:

- Your user model must set the attribute `USERNAME_FIELD` to indicate the field used as the username.

- Your user model must have a field (of some textual type, ideally `EmailField`) for storing an email address, and it must define the method `get_email_field_name()` to indicate the name of the email field.
- The username and email fields must be distinct. If you wish to use the email address as the username, you will need to write your own completely custom registration form.
- Your user model must have a field named `is_active`, and that field must be a `BooleanField` indicating whether the user's account is active.

If your user model is a subclass of Django's `AbstractBaseUser`, all of the above will be automatically handled for you.

If your custom user model defines additional fields beyond the minimum requirements, you'll either need to ensure that all of those fields are optional (i.e., can be *NULL* in your database, or provide a suitable default value defined in the model), or specify the correct list of fields to display in your *`RegistrationForm`* subclass.

### 7.1.2 The one-step workflow

*The one-step workflow* places the following requirements on your user model:

- Your user model must set the attribute `USERNAME_FIELD` to indicate the field used as the username.
- It must define a textual field named *password* for storing the user's password.

Also note that the base *`RegistrationForm`* includes and requires an email field, so either provide that field on your model and set the `get_email_field_name()` attribute to indicate which field it is, or subclass *`RegistrationForm`* and override to remove the *email* field or make it optional.

If your user model is a subclass of Django's `AbstractBaseUser`, all of the above will be automatically handled for you.

If your custom user model defines additional fields beyond the minimum requirements, you'll either need to ensure that all of those fields are optional (i.e., can be *NULL* in your database, or provide a suitable default value defined in the model), or specify the correct list of fields to display in your *`RegistrationForm`* subclass.

Because the one-step workflow logs in the new account immediately after creating it, you also must either use Django's `ModelBackend` as an authentication backend, or use an authentication backend which accepts a combination of *USERNAME_FIELD* and *password* as sufficient credentials to authenticate a user.

## 7.2 Writing your form subclass

The base *`RegistrationView`* contains code which compares the declared model of your registration form with the user model of your Django installation. If these are not the same model, the view will deliberately crash by raising an `ImproperlyConfigured` exception, with an error message alerting you to the problem.

This will happen automatically if you attempt to use django-registration with a custom user model and also attempt to use the default, unmodified `RegistrationForm`. This is, again, a deliberate design feature of django-registration, and not a bug: django-registration has no way of knowing in advance if your user model is compatible with the assumptions made by the built-in registration workflows (see above), so it requires you to take the explicit step of replacing the default registration form as a way of confirming you've manually checked the compatibility of your user model.

In the case where your user model is compatible with the default behavior of django-registration, you will be able to subclass *`RegistrationForm`*, set it to use your custom user model as the model, and then configure the views in django-registration to use your form subclass. For example, you might do the following (in a *forms.py* module somewhere in your codebase – do **not** directly edit django-registration's code):

```
from django_registration.forms import RegistrationForm

from mycustomuserapp.models import MyCustomUser


class MyCustomUserForm(RegistrationForm):
    class Meta(RegistrationForm.Meta):
        model = MyCustomUser
```

You may also need to specify the fields to include in the form, if the set of fields to include is different from the default set specified by the base *RegistrationForm*.

Then in your URL configuration (example here uses the two-step activation workflow), configure the registration view to use the form class you wrote:

```
from django.urls import include, path

from django_registration.backends.activation.views import RegistrationView

from mycustomuserapp.forms import MyCustomUserForm


urlpatterns = [
    # ... other URL patterns here
    path('accounts/register/',
        RegistrationView.as_view(
            form_class=MyCustomUserForm
        ),
        name='django_registration_register',
    ),
    path('accounts/',
        include('django_registration.backends.activation.urls')
    ),
    # ... more URL patterns
]
```

## 7.3 Incompatible user models

If your custom user model is not compatible with the built-in workflows of django-registration, you have several options.

One is to subclass the built-in form and view classes of django-registration and make the necessary adjustments to achieve compatibility with your user model. For example, if you want to use the two-step activation workflow, but your user model uses a completely different way of marking accounts active/inactive (compared to the the assumed *is_active* field), you might write subclasses of that workflow's *RegistrationView* and *ActivationView* which make use of your user model's mechanism for marking accounts active/inactive, and then use those views along with an appropriate subclass of *RegistrationForm*.

Or, if the incompatibilities are relatively minor and you don't mind making the change, you might use Django's migration framework to adjust your custom user model to match the assumptions made by django-registration's built-in workflows, thus allowing them to be used unmodified.

Finally, it may sometimes be the case that a given user model requires a completely custom set of form and view classes to support. Typically, this will also involve an account-registration process far enough from what django-registration's built-in workflows provide that you would be writing your own workflow in any case.

CHAPTER 8

# Validation utilities

To ease the process of validating user registration data, django-registration includes some validation-related data and utilities.

## 8.1 Error messages

Several error messages are available as constants. All of them are marked for translation; most have translations already provided in django-registration.

`django_registration.validators.`**`DUPLICATE_EMAIL`**
: Error message raised by *`RegistrationFormUniqueEmail`* when the supplied email address is not unique.

`django_registration.validators.`**`DUPLICATE_USERNAME`**
: Error message raised by `CaseInsensitiveValidator` when the supplied username is not unique. This is the same string raised by Django's default `User` model for a non-unique username.

`django_registration.validators.`**`RESERVED_NAME`**
: Error message raised by *`ReservedNameValidator`* when it is given a value that is a reserved name.

`django_registration.validators.`**`TOS_REQUIRED`**
: Error message raised by *`RegistrationFormTermsOfService`* when the terms-of-service field is not checked.

## 8.2 Rejecting "reserved" usernames

By default, django-registration treats some usernames as reserved.

**Note: Why reserved names are reserved**

Many Web applications enable per-user URLs (to display account information), and some may also create email addresses or even subdomains, based on a user's username. While this is often useful, it also represents a risk: a user

might register a name which conflicts with an important URL, email address or subdomain, and this might give that user control over it.

django-registration includes a list of reserved names, and rejects them as usernames by default, in order to avoid this issue.

---

**class** django_registration.validators.**ReservedNameValidator**(*reserved_names*)

A callable validator class (see Django's validators documentation) which prohibits the use of a reserved name as the value.

By default, this validator is applied to the username field of *django_registration.forms.RegistrationForm* and all of its subclasses. This validator is attached to the list of validators for the username field, so to remove it (not recommended), subclass *RegistrationForm* and override *__init__()* to change the set of validators on the username field.

If you want to supply your own custom list of reserved names, you can subclass *RegistrationForm* and set the attribute *reserved_names* to the list of values you want to disallow.

The default list of reserved names, if you don't specify one, is *DEFAULT_RESERVED_NAMES*. The validator will also reject any value beginning with the string *".well-known"* (see RFC 5785).

**Parameters** **reserved_names** (*list*) – A list of reserved names to forbid.

**Raises** **django.core.exceptions.ValidationError** – if the provided value is reserved.

Several constants are provided which are used by this validator:

django_registration.validators.**CA_ADDRESSES**

A list of email usernames commonly used by certificate authorities when verifying identity.

django_registration.validators.**NOREPLY_ADDRESSES**

A list of common email usernames used for automated messages from a Web site (such as "noreply" and "mailer-daemon").

django_registration.validators.**PROTOCOL_HOSTNAMES**

A list of protocol-specific hostnames sites commonly want to reserve, such as "www" and "mail".

django_registration.validators.**OTHER_SENSITIVE_NAMES**

Other names, not covered by any of the other lists, which have the potential to conflict with common URLs or subdomains, such as "blog" and "docs".

django_registration.validators.**RFC_2142**

A list of common email usernames specified by RFC 2142.

django_registration.validators.**SENSITIVE_FILENAMES**

A list of common filenames with important meanings, such that usernames should not be allowed to conflict with them (such as "favicon.ico" and "robots.txt").

django_registration.validators.**SPECIAL_HOSTNAMES**

A list of hostnames with reserved or special meaning (such as "autoconfig", used by some email clients to automatically discover configuration data for a domain).

django_registration.validators.**DEFAULT_RESERVED_NAMES**

A list made of the concatenation of all of the above lists, used as the default set of reserved names for *ReservedNameValidator*.

## 8.3 Protecting against homograph attacks

By default, Django permits a broad range of Unicode to be used in usernames; while this is useful for serving a worldwide audience, it also creates the possibility of homograph attacks through the use of characters which are easily visually confused for each other (for example: “pypl” containing a Cyrillic “”, visually indistinguishable in many fonts from a Latin “”).

To protect against this, django-registration applies some validation rules to usernames and email addresses.

`django_registration.validators.`**`validate_confusables`**(*value*)
A custom validator which prohibits the use of dangerously-confusable usernames.

This validator will reject any mixed-script value (as defined by Unicode ‘Script’ property) which also contains one or more characters that appear in the Unicode Visually Confusable Characters file.

This validator is enabled by default on the username field of registration forms.

**Parameters** **`value`** (*str*) – The username value to validate (non-string usernames will not be checked)

**Raises** **`django.core.exceptions.ValidationError`** – if the value is mixed-script confusable

`django_registration.validators.`**`validate_confusables_email`**(*value*)
A custom validator which prohibits the use of dangerously-confusable email address.

This validator will reject any email address where either the local-part of the domain is – when considered in isolation – dangerously confusable. A string is dangerously confusable if it is a mixed-script value (as defined by Unicode ‘Script’ property) which also contains one or more characters that appear in the Unicode Visually Confusable Characters file.

This validator is enabled by default on the email field of registration forms.

**Parameters** **`value`** (*str*) – The email address to validate

**Raises** **`django.core.exceptions.ValidationError`** – if the value is mixed-script confusable

## 8.4 Other validators

**`class`** `django_registration.validators.`**`CaseInsensitiveUnique`**(*model*, *field_name*)
A callable validator class (see Django’s validators documentation) which enforces case-insensitive uniqueness on a given field of a particular model. Used by *`RegistrationFormCaseInsensitive`* for case-insensitive username uniqueness, and *`RegistrationFormUniqueEmail`* for unique email addresses.

**Parameters**

- **`model`** (*`django.db.models.Model`*) – The model class to query against for uniqueness checks.
- **`field_name`** (*`str`*) – The field name to perform the uniqueness check against.

**Raises** **`django.core.exceptions.ValidationError`** – if the value is not unique.

**`class`** `django_registration.validators.`**`HTML5EmailValidator`**
A callable validator class (see Django’s validators documentation) which enforces the HTML5 email address format. The format used by HTML5’s *input type=”email”* is deliberately more restrictive than what is permitted by the latest email RFCs; specifically, HTML5’s validation rule disallows a number of rare and problematic features – such as embedded comments and quoted-string inclusion of otherwise-illegal characters – which are

technically legal to have in an email address but which now mostly serve to confuse or complicate parsers, rather than to provide actual utility.

CHAPTER 9

# Exception classes

django-registration provides two base exception classes to signal errors which occur during the signup or activation processes.

**exception** django_registration.exceptions.**RegistrationError**(*message*, *code*, *params*)

Base exception class for all exceptions raised in django-registration. No code in django-registration will raise this exception directly; it serves solely to provide a distinguishing parent class for other errors. Arguments passed when the exception is raised will be stored and exposed as attributes of the same names on the exception object:

**Parameters**

- **message** (*str*) – A human-readable error message.
- **code** (*str*) – A short but unique identifier used by subclasses to distinguish different error conditions.
- **params** (*dict*) – Arbitrary key-value data to associate with the error.

**exception** django_registration.exceptions.**ActivationError**(*message*, *code*, *params*)

Exception class to indicate errors during account activation. Subclass of *RegistrationError* and inherits its attributes.

# CHAPTER 10

# Custom settings

Although the choice of registration workflow does not necessarily require changes to your Django settings (as registration workflows are selected by including the appropriate URL patterns in your root URLconf), the built-in workflows of django-registration make use of several custom settings.

`django.conf.settings.`**`ACCOUNT_ACTIVATION_DAYS`**
: An `int` indicating how long (in days) after signup an account has in which to activate.

  Used by:

  - *The two-step activation workflow*

`django.conf.settings.`**`REGISTRATION_OPEN`**
: A `bool` indicating whether registration of new accounts is currently permitted.

  A default of *True* is assumed when this setting is not supplied, so specifying it is optional unless you want to temporarily close registration (in which case, set it to *False*).

  Used by:

  - *The two-step activation workflow*
  - *The one-step workflow*

  Third-party workflows wishing to use an alternate method of determining whether registration is allowed should subclass *`django_registration.views.RegistrationView`* (or a subclass of it from an existing workflow) and override *`registration_allowed()`*.

`django.conf.settings.`**`REGISTRATION_SALT`**
: A `str` used as an additional "salt" in the process of generating signed activation keys.

  This setting is optional, and a default of *"registration"* will be used if not specified. The value of this setting does not need to be kept secret; see *the note about this salt value and security* for details.

  Used by:

  - *The two-step activation workflow*

CHAPTER 11

# Signals used by django-registration

Much of django-registration's customizability comes through the ability to write and use different workflows for user registration. However, there are many cases where only a small bit of additional logic needs to be injected into the registration process, and writing a custom workflow to support this represents an unnecessary amount of work. A more lightweight customization option is provided through two custom signals which the built-in registration workflows send, and custom workflows are encouraged to send, at specific points during the registration process; functions listening for these signals can then add whatever logic is needed.

For general documentation on signals and the Django dispatcher, consult Django's signals documentation. This documentation assumes that you are familiar with how signals work and the process of writing and connecting functions which will listen for signals.

`django_registration.signals.`**`user_activated`**

Sent when a user account is activated (not applicable to all workflows). Provides the following arguments:

***sender*** The *`ActivationView`* subclass used to activate the user.

***user*** A user-model instance representing the activated account.

***request*** The `HttpRequest` in which the account was activated.

This signal is automatically sent for you by the base *`ActivationView`*, so unless you've overridden its `get()` method in a subclass you should not need to explicitly send it.

`django_registration.signals.`**`user_registered`**

Sent when a new user account is registered. Provides the following arguments:

***sender*** The *`RegistrationView`* subclass used to register the account.

***user*** A user-model instance representing the new account.

***request*** The `HttpRequest` in which the new account was registered.

This signal is **not** automatically sent for you by the base *`RegistrationView`*. It is sent by the subclasses implemented for the three included registration workflows, but if you write your own subclass of *`RegistrationView`*, you'll need to send this signal as part of the implementation of the *`register()`* method.

CHAPTER 12

# Feature and API deprecation cycle

This document will list any features or APIs of django-registration which are deprecated and scheduled to be removed in future releases.

As of 3.1.1, no features or APIs are currently deprecated.

CHAPTER 13

# Security guide

**Important: Reporting security issues**

If you believe you have found a security issue in django-registration, please do *not* use the public GitHub issue tracker to report it. Instead, you can contact the author privately to report the issue.

Anything related to users or user accounts has security implications and represents a large source of potential security issues. This document is not an exhaustive guide to those implications and issues, and makes no guarantees that your particular use of Django or django-registration will be safe; instead, it provides a set of recommendations, and explanations for why django-registration does certain things or recommends particular approaches. Using this software responsibly is, ultimately, up to you.

Before continuing with this document, you should ensure that you've read and understood Django's security documentation. Django provides a good overview of common security issues in the general field of web development, and an explanation of how it helps to protect against them or provides tools to help you do so.

You should also ensure you're following Django's security recommendations. You can check for many common issues by running:

```
python manage.py check --tag security
```

on your codebase.

## 13.1 Recommendation: use the two-step activation workflow

Two user-signup workflows are included in django-registration, along with support for writing your own. If you choose to use one of the included workflows, *the two-step activation workflow* is the recommended default.

The activation workflow provides a verification step – the user must click a link sent to the email address they used to register – which can serve as an impediment to automated account creation for malicious purposes.

The activation workflow generates an activation key which consists of the new account's username and the timestamp of the signup, verified using Django's cryptographic signing tools which in turn use the HMAC implementation from

the Python standard library. Thus, django-registration is not inventing or building any new cryptography, but only using existing/vetted implementations in an approved and standard manner.

Additionally, the activation workflow takes steps to ensure that its use of HMAC does not act as an oracle. Several parts of Django use the signing tools, and third-party applications are free to use them as well, so django-registration makes use of the ability to supply a salt value for the purpose of "namespacing" HMAC usage. Thus an activation token generated by django-registration's activation workflow should not be usable for attacks against other HMAC-carrying values generated by the same installation of Django.

## 13.2 Restrictions on user names: reserved names

By default, django-registration applies a list of reserved names, and does not permit users to create accounts using those names (see `ReservedNameValidator`). The default list of reserved names includes many names that could cause confusion or even inappropriate access. These reserved names fall into several categories:

- Usernames which could allow a user to impersonate or be seen as a site administrator. For example, *'admin'* or *'administrator'*.
- Usernames corresponding to standard/protocol-specific email addresses (relevant for sites where creating an account also creates an email address with that username). For example, *'webmaster'*.
- Usernames corresponding to standard/sensitive subdomain names (relevant for sites where creating an account also creates a subdomain corresponding to the username). For example, *'ftp'* or *'autodiscover'*.
- Usernames which correspond to sensitive URLs (relevant for sites where user profiles appear at a URL containing the username). For example, *'contact'* or *'buy'*.

It is strongly recommended that you leave the reserved-name validation enabled.

## 13.3 Restrictions on user names and email addresses: Unicode

By default, django-registration permits the use of a wide range of Unicode in usernames and email addresses. However, to prevent some types of Unicode-related attacks, django-registration will not permit certain specific uses of Unicode characters.

For example, while the username *'admin'* cannot normally be registered (see above), a user might still attempt to register a name that appears visually identical, by substituting a Cyrillic 'a' or other similar-appearing character for the first character. This is a homograph attack.

To prevent homograph attacks, django-registration applies the following rule to usernames, and to the local-part and the domain of email addresses:

- If the submitted value is mixed-script (contains characters from multiple different scripts, as in the above example which would mix Cyrillic and Latin characters), and
- If the submitted value contains characters appearing in the Unicode Visually Confusable Characters file,
- Then the value will be rejected.

See `validate_confusables()` and `validate_confusables_email()`.

This should not interfere with legitimate use of Unicode, or of non-English/non-Latin characters in usernames and email addresses. To avoid a common false-positive situation, the local-part and domain of an email address are checked independently of each other.

It is strongly recommended that you leave this validation enabled.

## 13.4 Additional steps to secure user accounts

The scope of django-registration is solely the implementation of user-signup workflows, which limits the ways in which django-registration alone can protect your users. Other features of Django itself, or of other third-party applications, can provide significant increases in protection.

In particular, it is recommended that you:

- Prevent the use of common passwords. You can catch some common passwords by enabling Django's `CommonPasswordValidator`, which uses a list of twenty thousand common passwords. A more comprehensive option is the password validator and other utilities from pwned-passwords-django, which checks against a database containing (as of mid-2018) over half a billion passwords found in data breaches.
- Use two-factor authentication via authenticator applications or hardware security keys (*not* SMS). The package django-two-factor provides integration for two-factor authentication into Django's auth framework.

CHAPTER 14

# Upgrading from previous versions

The current release series of django-registration is the 3.x series, which is not backwards-compatible with the django-registration 2.x release series.

## 14.1 Changes within the 3.x series

Within the 3.x release series, there have been several minor changes and improvements, documented here along with the version in which they occurred.

### 14.1.1 django-registration 3.1

- When an attempt was made to use django-registration with a custom user model, but *without* explicitly subclassing `RegistrationForm` to point to that user model, previously the result would be a cryptic exception and error message raised from within Django, complaining about trying to work with the swapped-out user model. `RegistrationView` now explicitly raises `ImproperlyConfigured` with an informative error message to make it clear what has happened, and directs the developer to the documentation for using custom user models in django-registration.
- A new validator, `HTML5EmailValidator`, is included and is applied by default to the email field of `RegistrationForm`. The HTML5 email address grammar is more restrictive than the RFC grammar, but primarily in disallowing rare and problematic features.
- Support for Python 2 was dropped, as Python 2 is EOL as of 2020-01-01. As a result, support for Django 1.11 (EOL April 2020) was also dropped; the minimum supported Django version is now 2.2.

### 14.1.2 django-registration 3.0.1

- The *custom validators* are now serializable.
- Although no code changes were required, this release officially marks itself compatible with Python 3.7 and with django 2.2.

## 14.2 Changes between django-registration 2.x and 3.x

### 14.2.1 Module renaming

Prior to 3.x, django-registration installed a Python module named *registration*. To avoid silent incompatibilities, and to conform to more recent best practices, django-registration 3.x now installs a module named *django_registration*. Attempts to import from the *registration* module will immediately fail with `ImportError`.

Many installations will be able to adapt by replacing references to *registration* with references to *django_registration*.

### 14.2.2 Removal of model-based workflow

The two-step model-based signup workflow, which has been present since the first public release of django-registration in 2007, has now been removed. In its place, it is recommended that you use *the two-step activation workflow* instead, as that workflow requires no server-side storage of additional data beyond the user account itself.

### 14.2.3 Renaming of two-step activation workflow

*The two-step activation workflow* was previously found at *registration.backends.hmac*; it has been renamed and is now found at *registration.backends.activation*.

### 14.2.4 Renaming of one-step workflow

*The one-step workflow* was previously found at *registration.backends.simple*; it has been renamed and is now found at *registration.backends.one_step*.

### 14.2.5 Removal of auth URLs

Prior to 3.x, django-registration's default URLconf modules for its built-in workflows would attempt to include the Django auth views (login, logout, password reset, etc.) for you. This became untenable with the rewrite of Django's auth views to be class-based, as it required detecting the set of auth views and choosing a set of URL patterns at runtime.

As a result, auth views are no longer automatically configured for you; if you want them, `include()` the URLconf *django.contrib.auth.urls* at a location of your choosing.

### 14.2.6 Distinguishing activation failure conditions

Prior to 3.x, failures to activate a user account (in workflows which use activation) all simply returned *None* in place of the activated account. This meant it was not possible to determine, from inspecting the result, what exactly caused the failure.

In django-registration 3.x, activation failures raise an exception – `ActivationError` – with a message and code (such as *"expired"*), to indicate the cause of failure. This exception is caught by `ActivationView` and turned into the template context variable *activation_error*.

### 14.2.7 Changes to custom user support

Support for custom user models has been brought more in line with the features Django offers. This affects compatibility of custom user models with django-registration's default forms and views. In particular, custom user models should now provide, in addition to `USERNAME_FIELD`, the `get_username()` and `get_email_field_name()` methods. See *the custom user documentation* for details.

### 14.2.8 Changes to *success_url*

Both the registration and activation views mimic Django's own generic views in supporting a choice of ways to specify where to redirect after a successful registration or activation; you can either set the attribute `success_url` on the view class, or implement the method `get_success_url()` . However, there is a key difference between the base Django generic-view version of this, and the version in django-registration: when calling a `get_success_url()` method, django-registration passes the user account as an argument.

This is incompatible with the behavior of Django's base `FormMixin`, which expects `get_success_url()` to take zero arguments.

Also, earlier versions of django-registration allowed `success_url` and `get_success_url()` to provide either a string URL, or a tuple of *(viewname, args, kwargs)* to pass to Django's `reverse()` helper, in order to work around issues caused by calling `reverse()` at the level of a class attribute.

In django-registration 3.x, the *user* argument to `get_success_url()` is now optional, meaning `FormMixin`'s default behavior is now compatible with any `get_success_url()` implementation that doesn't require the user object; as a result, implementations which don't rely on the user object should either switch to specifying `success_url` as an attribute, or change their own signature to *get_success_url(self, user=None)*.

Also, the ability to supply the 3-tuple of arguments for `reverse()` has been removed; both `success_url` and `get_success_url()` now *must* be/return either a string, or a lazy object that resolves to a string. To avoid class-level calls to `reverse()`, use *django.urls.reverse_lazy()* instead.

### 14.2.9 Removed "no free email" form

Earlier versions of django-registration included a form class, *RegistrationFormNoFreeEmail*, which attempted to forbid user signups using common free/throwaway email providers. Since this is a pointless task (the number of possible domains of such providers is ever-growing), this form class has been removed.

### 14.2.10 Template names

Since django-registration's Python module has been renamed from *registration* to *django_registration*, its default template folder has also been renamed, from *registration* to *django_registration*. Additionally, the following templates have undergone name changes:

- The default template name for the body of the activation email in the two-step activation workflow is now *django_registration/activation_email_body.txt* (previously, it was *registration/activation_email.txt*)
- The default template name for `ActivationView` and its subclasses is now *django_registration/activation_failed.html* (previously, it was *registration/activate.html*).

### 14.2.11 Renaming of URL patterns

Prior to 3.x, django-registration's included URLconf modules provided URL pattern names beginning with *"registration"*. For example: *"registration_register"*. In 3.x, these are all renamed to begin with *"django_registration"*. For example: *"django_registration_register"*.

### 14.2.12 Other changes

The URLconf *registration.urls* has been removed; it was an alias for the URLconf of the model-based workflow, which has also been removed.

The compatibility alias *registration.backends.default*, which also pointed to the model-based workflow, has been removed.

## 14.3 Changes during the 2.x release series

One major change occurred between django-registration 2.0 and 2.1: the addition in version 2.1 of the `ReservedNameValidator`, which is now used by default on `RegistrationForm` and its subclasses.

This is technically backwards-incompatible, since a set of usernames which previously could be registered now cannot be registered, but was included because the security benefits outweigh the edge cases of the now-disallowed usernames. If you need to allow users to register with usernames forbidden by this validator, see its documentation for notes on how to customize or disable it.

In 2.2, the behavior of the *RegistrationProfile.expired()* method was clarified to accommodate user expectations; it does *not* return (and thus, *RegistrationProfile.delete_expired_users()* does not delete) profiles of users who had successfully activated.

In django-registration 2.3, the new validators `validate_confusables()` and `validate_confusables_email()` were added, and are applied by default to the username field and email field, respectively, of registration forms. This may cause some usernames which previously were accepted to no longer be accepted, but like the reserved-name validator this change was made because its security benefits significantly outweigh the edge cases in which it might disallow an otherwise-acceptable username or email address. If for some reason you need to allow registration with usernames or email addresses containing potentially dangerous use of Unicode, you can subclass the registration form and remove these validators, though doing so is not recommended.

## 14.4 Versions prior to 2.0

A 1.0 release of django-registration existed, but the 2.x series was compatible with it.

Prior to 1.0, the most widely-adopted version of django-registration was 0.8; the changes from 0.8 to 2.x were large and significant, and if any installations on 0.8 still exist and wish to upgrade to more recent versions, it is likely the most effective route will be to discard all code using 0.8 and start over from scratch with a 3.x release.

# CHAPTER 15

# Frequently-asked questions

The following are miscellaneous common questions and answers related to installing/using django-registration, culled from bug reports, emails and other sources.

## 15.1 General

### 15.1.1 This doesn't work with custom user models! It crashes as soon as I try to use one!

django-registration can work perfectly well with a custom user model, but this does require you to do a bit more work. Please thoroughly read *the documentation for how to use custom user models* before filing a bug.

Please also note that, as explained in that documentation, by default django-registration will crash if you try to use a custom user model without following the documentation. This is not a bug; it is done deliberately to ensure you read and follow the documentation.

### 15.1.2 How can I support social-media and other auth schemes, like Facebook or GitHub?

By using django-allauth. No single application can or should provide a universal API for every authentication system ever developed; django-registration sticks to making it easy to implement typical signup workflows using Django's default model-based authentication system, while apps like *django-allauth* handle integration with third-party authentication services far more effectively.

### 15.1.3 What license is django-registration under?

django-registration is offered under a three-clause BSD-style license; this is an OSI-approved open-source license, and allows you a large degree of freedom in modifying and redistributing the code. For the full terms, see the file *LICENSE* which came with your copy of django-registration; if you did not receive a copy of this file, you can view it online at <https://github.com/ubernostrum/django-registration/blob/master/LICENSE>.

### 15.1.4 What versions of Django and Python are supported?

As of django-registration 3.1.1, Django 2.2, 3.0, and 3.1 are supported, on Python 3.5 (Django 2.2 only), 3.6, 3.7, and 3.8.

### 15.1.5 I found a bug or want to make an improvement!

**Important: Reporting security issues**

If you believe you have found a security issue in django-registration, please do *not* use the public GitHub issue tracker to report it. Instead, you can contact the author privately to report the issue.

The canonical development repository for django-registration is online at <https://github.com/ubernostrum/django-registration>. Issues and pull requests can both be filed there.

If you'd like to contribute to django-registration, that's great! Just please remember that pull requests should include tests and documentation for any changes made, and that following PEP 8 is mandatory. Pull requests without documentation won't be merged, and PEP 8 style violations or test coverage below 100% are both configured to break the build.

### 15.1.6 How secure is django-registration?

Over the decade-plus history of django-registration, there have been no security issues reported which required new releases to remedy. This doesn't mean, though, that django-registration is perfectly secure: much will depend on ensuring best practices in deployment and server configuration, and there could always be security issues that just haven't been recognized yet.

django-registration does, however, try to avoid common security issues:

- django-registration 3.1.1 only supports versions of Django which were receiving upstream security support at the time of release.
- django-registration does not attempt to generate or store passwords, and does not transmit credentials which could be used to log in (the only "credential" ever sent out by django-registration is an activation key used in the two-step activation workflow, and that key can only be used to make an account active; it cannot be used to log in).
- django-registration works with Django's own security features (including cryptographic features) where possible, rather than reinventing its own.

For more details, see *the security guide*.

### 15.1.7 How do I run the tests?

django-registration's tests are run using tox, but typical installation of django-registration (via *pip install django-registration*) will not install the tests.

To run the tests, download the source (*.tar.gz*) distribution of django-registration 3.1.1 from its page on the Python Package Index, unpack it (*tar zxvf django-registration-|release|.tar.gz* on most Unix-like operating systems), and in the unpacked directory run *tox*.

Note that you will need to have *tox* installed already (*pip install tox*), and to run the full test matrix you will need to have each supported version of Python available. To run only the tests for a specific Python version and Django version, you can invoke *tox* with the *-e* flag. For example, to run tests for Python 3.6 and Django 2.0: *tox -e py36-django20*.

## 15.2 Installation and setup

### 15.2.1 How do I install django-registration?

Full instructions are available in *the installation guide*. For configuration, see *the quick start guide*.

### 15.2.2 Does django-registration come with any sample templates I can use right away?

No, for two reasons:

1. Providing default templates with an application is ranges from hard to impossible, because different sites can have such wildly different design and template structure. Any attempt to provide templates which would work with all the possibilities would probably end up working with none of them.
2. A number of things in django-registration depend on the specific registration workflow you use, including the variables which end up in template contexts. Since django-registration has no way of knowing in advance what workflow you're going to be using, it also has no way of knowing what your templates will need to look like.

Fortunately, however, django-registration has good documentation which explains what context variables will be available to templates, and so it should be easy for anyone who knows Django's template system to create templates which integrate with their own site.

## 15.3 Configuration

### 15.3.1 Do I need to rewrite the views to change the way they behave?

Not always. Any behavior controlled by an attribute on a class-based view can be changed by passing a different value for that attribute in the URLconf. See Django's class-based view documentation for examples of this.

For more complex or fine-grained control, you will likely want to subclass `RegistrationView` or `ActivationView`, or both, add your custom logic to your subclasses, and then create a URLconf which makes use of your subclasses.

### 15.3.2 I don't want to write my own URLconf because I don't want to write patterns for all the auth views!

You're in luck, then; Django provides a URLconf for this, at *django.contrib.auth.urls*.

### 15.3.3 I don't like the names you've given to the URL patterns!

In that case, you should feel free to set up your own URLconf which uses the names you want.

### 15.3.4 I'm using a custom user model; how do I make that work?

See *the custom user documentation*.

## 15.4 Tips and tricks

### 15.4.1 How do I close user signups?

If you haven't modified the behavior of the *registration_allowed()* method in *RegistrationView*, you can use the setting *REGISTRATION_OPEN* to control this; when the setting is *True*, or isn't supplied, user registration will be permitted. When the setting is *False*, user registration will not be permitted.

### 15.4.2 How do I log a user in immediately after registration or activation?

Take a look at the implementation of *the one-step workflow*, which logs a user in immediately after registration.

### 15.4.3 How do I manually activate a user?

In *the two-step activation workflow*, toggle the *is_active* field of the user in the admin.

### 15.4.4 How do I delete expired unactivated accounts?

Perform a query for those accounts, and call the *delete()* method of the resulting *QuerySet*. Since django-registration doesn't know in advance what your definition of "expired" will be, it leaves this step to you.

### 15.4.5 How do I allow Unicode in usernames?

Use Python 3. Django's username validation allows any word character plus some additional characters, but the definition of "word character" depends on the Python version in use. On Python 2, only ASCII will be permitted; on Python 3, usernames containing word characters matched by a regex with the `re.UNICODE` flag will be accepted.

### 15.4.6 How do I tell why an account's activation failed?

If you're using *the two-step activation workflow*, the template context will contain a variable *activation_error* containing the information passed when the *ActivationError* was raised. This will indicate what caused the failure.

**See also:**

- Django's authentication documentation. Django's authentication system is used by django-registration's default configuration.

# Python Module Index

## d

# Index

## N



## O



## P



## R



## S



## T



## U



## V