

---

# **django-registration-redux** **Documentation**

*Release 1.4*

**James Bennett**

June 02, 2016



<b>1</b>	<b>Quick start guide</b>	<b>3</b>
<b>2</b>	<b>Release notes</b>	<b>7</b>
<b>3</b>	<b>Upgrade guide</b>	<b>9</b>
<b>4</b>	<b>The default backend</b>	<b>11</b>
<b>5</b>	<b>The “simple” (one-step) backend</b>	<b>15</b>
<b>6</b>	<b>Forms for user registration</b>	<b>17</b>
<b>7</b>	<b>Registration views</b>	<b>19</b>
<b>8</b>	<b>Custom signals used by django-registration-redux</b>	<b>21</b>
<b>9</b>	<b>Frequently-asked questions</b>	<b>23</b>
	<b>Python Module Index</b>	<b>27</b>



This documentation covers the 1.4 release of django-registration-redux, a simple but extensible application providing user registration functionality for [Django](#) powered websites.

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, consult the [quick-start guide](#), which describes all the necessary steps to install django-registration-redux and configure it for the default workflow. For more detailed information, including how to customize the registration process (and support for alternate registration systems), read through the documentation listed below.

If you are upgrading from a previous release, please read the [upgrade guide](#) for information on what's changed.

Contents:



---

## Quick start guide

---

Before installing `django-registration-redux`, you'll need to have a copy of `Django` already installed. For the 1.4 release, `Django` 1.7 or newer is required.

For further information, consult the [Django download page](#), which offers convenient packaged downloads and installation instructions.

### 1.1 Installing `django-registration-redux`

There are several ways to install `django-registration-redux`:

- Automatically, via a package manager.
- Manually, by downloading a copy of the release package and installing it yourself.
- Manually, by performing a Git checkout of the latest code.

It is also highly recommended that you learn to use `virtualenv` for development and deployment of Python software; `virtualenv` provides isolated Python environments into which collections of software (e.g., a copy of `Django`, and the necessary settings and applications for deploying a site) can be installed, without conflicting with other installed software. This makes installation, testing, management and deployment far simpler than traditional site-wide installation of Python packages.

#### 1.1.1 Automatic installation via a package manager

Several automatic package-installation tools are available for Python; the recommended one is `pip`.

Using `pip`, type:

```
pip install django-registration-redux
```

It is also possible that your operating system distributor provides a packaged version of `django-registration-redux`. Consult your operating system's package list for details, but be aware that third-party distributions may be providing older versions of `django-registration-redux`, and so you should consult the documentation which comes with your operating system's package.

#### 1.1.2 Manual installation from a downloaded package

If you prefer not to use an automated package installer, you can download a copy of `django-registration-redux` and install it manually. The latest release package can be downloaded from `django-registration-redux`'s [listing on the Python Package Index](#).

Once you've downloaded the package, unpack it (on most operating systems, simply double-click; alternately, type `tar zxvf django-registration-redux-1.1.tar.gz` at a command line on Linux, Mac OS X or other Unix-like systems). This will create the directory `django-registration-redux-1.1`, which contains the `setup.py` installation script. From a command line in that directory, type:

```
python setup.py install
```

Note that on some systems you may need to execute this with administrative privileges (e.g., `sudo python setup.py install`).

### 1.1.3 Manual installation from a Git checkout

If you'd like to try out the latest in-development code, you can obtain it from the `django-registration-redux` repository, which is hosted at [Github](#) and uses [Git](#) for version control. To obtain the latest code and documentation, you'll need to have Git installed, at which point you can type:

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

You can also obtain a copy of a particular release of `django-registration-redux` by specifying the `-b` argument to `git clone`; each release is given a tag of the form `vX.Y`, where "X.Y" is the release number. So, for example, to check out a copy of the 1.4 release, type:

```
git clone -b v1.0 https://github.com/macropin/django-registration.git
```

In either case, this will create a copy of the `django-registration-redux` Git repository on your computer; you can then add the `django-registration-redux` directory inside the checkout your Python import path, or use the `setup.py` script to install as a package.

## 1.2 Basic configuration and use

Once installed, you can add `django-registration-redux` to any Django-based project you're developing. The default setup will enable user registration with the following workflow:

1. A user signs up for an account by supplying a username, email address and password.
2. From this information, a new `User` object is created, with its `is_active` field set to `False`. Additionally, an activation key is generated and stored, and an email is sent to the user containing a link to click to activate the account.
3. Upon clicking the activation link, the new account is made active (the `is_active` field is set to `True`); after this, the user can log in.

Note that the default workflow requires `django.contrib.auth` to be installed, and it is recommended that `django.contrib.sites` be installed as well. You will also need to have a working mail server (for sending activation emails), and provide Django with the necessary settings to make use of this mail server (consult [Django's email-sending documentation](#) for details).

### 1.2.1 Settings

Begin by adding `registration` to the `INSTALLED_APPS` setting of your project, and specifying one additional setting:

**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 and may be deleted by maintenance scripts provided in `django-registration-redux`.



**REGISTRATION\_DEFAULT\_FROM\_EMAIL** Optional. If set, emails sent through the registration app will use this string. Falls back to using Django’s built-in `DEFAULT_FROM_EMAIL` setting.

**REGISTRATION\_EMAIL\_HTML** Optional. If this is *False*, registration emails will be send in plain text. If this is *True*, emails will be sent as HTML. Defaults to *True*.

**REGISTRATION\_AUTO\_LOGIN** Optional. If this is *True*, your users will automatically log in when they click on the activation link in their email. Defaults to *False*.

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

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.sites',
    'registration',
    # ...other installed applications...
)

ACCOUNT_ACTIVATION_DAYS = 7 # One-week activation window; you may, of course, use a different value.
REGISTRATION_AUTO_LOGIN = True # Automatically log the user in.
```

Once you’ve done this, run `python manage.py migrate` to install the model used by the default setup.

## 1.2.2 Setting up URLs

The *default backend* includes a Django URLconf which sets up URL patterns for *the views in django-registration-redux*, as well as several useful views in `django.contrib.auth` (e.g., login, logout, password change/reset). This URLconf can be found at `registration.backends.default.urls`, and so can simply be included in your project’s root URL configuration. For example, to place the URLs under the prefix `/accounts/`, you could add the following to your project’s root URLconf:

```
(r'^accounts/', include('registration.backends.default.urls')),
```

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

Another URLConf is also provided – at `registration.auth_urls` – which just handles the Django auth views, should you want to put those at a different location.

## 1.2.3 Templates

The templates in `django-registration-redux` assume you have a *base.html* template in your project’s template directory. This base template should include a `title` block and a `content` block. Other than that, every template needed is included. You can extend and customize the included templates as needed. Some of the templates you’ll probably want to customize are covered here:

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`.

### 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 be an instance of some subclass of `django.forms.Form`; consult [Django’s forms documentation](#) for information on how to display this in a template.

### registration/registration\_complete.html

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

### **registration/activate.html**

Used if account activation fails. With the default setup, has the following context:

**activation\_key** The activation key used during the activation attempt.

### **registration/activation\_complete.html**

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

### **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.

**expiration\_days** The number of days remaining during which the account may be activated.

**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.models.RequestSite` (if not). Consult [the documentation for the Django sites framework](#) for details regarding these objects' interfaces.

### **registration/activation\_email.txt**

**IMPORTANT:** If you override this template, you must also override the HTML version (below), or disable HTML emails by adding `REGISTRATION_EMAIL_HTML = False` to your `settings.py`.

Used to generate the text 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.

**expiration\_days** The number of days remaining during which the account may be activated.

**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.models.RequestSite` (if not). Consult [the documentation for the Django sites framework](#) for details regarding these objects' interfaces.

**user** The new user account

### **registration/activation\_email.html**

This template is used to generate the html body of the activation email. Should display the same content as the text version of the activation email.

The context available is the same as the text version of the template.

---

**Release notes**

---

The 1.4 release of django-registration-redux supports Python 2.7, 3.4, 3.5 and Django 1.7, 1.8, and 1.9.



---

## Upgrade guide

---

The 1.4 release of `django-registration-redux` is not compatible with the legacy `django-registration` (previously maintained by James Bennett).

### 3.1 Django version requirement

As of 1.4, `django-registration-redux` requires Django 1.7 or newer; older Django releases may work, but are officially unsupported. Additionally, `django-registration-redux` officially supports Python 2.7, 3.3, 3.4, and 3.5.

### 3.2 Backwards-incompatible changes

#### 3.2.1 Version 1.4

- Remove unnecessary `_RequestPassingFormView`. See #56. Please ensure that you update any subclassed views to reference `self.request` instead of accepting `request` as an argument.

#### 3.2.2 Version 1.3

- Django 1.7 or newer is required. Please ensure you upgrade your Django version before upgrading.

#### 3.2.3 Version 1.2

- **Native migration support breaks South compatibility:** An initial native migration for Django > 1.7 has been provided. South users will need to configure a null migration with (`SOUTH_MIGRATION_MODULES`) in `settings.py` as shown below:

```
SOUTH_MIGRATION_MODULES = {  
    'registration': 'registration.south_migrations',  
}
```

- **register method in RegistrationView has different parameters:** The parameters of the `register` method in `RegistrationView` have changed.

### 3.2.4 Version 1.1

- **base.html template required:** A *base.html* template is now assumed to exist. Please ensure that your project provides one for django-registration-redux to inherit from.
- **HTML email templates:** django-registration-redux now uses HTML email templates. If you previously customized text email templates, you need to do the same with the new HTML templates.

---

## The default backend

---

A default registration backend' is bundled with `django-registration-redux`, as the module `registration.backends.default`, and implements a simple two-step workflow in which a new user first registers, then confirms and activates the new account by following a link sent to the email address supplied during registration.

### 4.1 Default behavior and configuration

To make use of this backend, simply include the `URLConf` `registration.backends.default.urls` at whatever location you choose in your URL hierarchy.

This backend makes use of the following settings:

**ACCOUNT\_ACTIVATION\_DAYS** This is the number of days users will have to activate their accounts after registering. Failing to activate during that period will leave the account inactive (and possibly subject to deletion). This setting is required, and must be an integer.

**REGISTRATION\_OPEN** A boolean (either `True` or `False`) indicating whether registration of new accounts is currently permitted. This setting is optional, and a default of `True` will be assumed if it is not supplied.

**INCLUDE\_AUTH\_URLS** A boolean (either `True` or `False`) indicating whether auth urls (mapped to `django.contrib.auth.views`) should be included in the `urlpatterns` of the application backend.

**INCLUDE\_REGISTER\_URL** A boolean (either `True` or `False`) indicating whether the view for registering accounts should be included in the `urlpatterns` of the application backend.

**REGISTRATION\_FORM** A string dotted path to the desired registration form.

By default, this backend uses `registration.forms.RegistrationForm` as its form class for user registration; this can be overridden by passing the keyword argument `form_class` to the `register()` view.

Two views are provided: `registration.backends.default.views.RegistrationView` and `registration.backends.default.views.ActivationView`. These views subclass `django-registration-redux`'s base `RegistrationView` and `ActivationView`, respectively, and implement the two-step registration/activation process.

Upon successful registration – not activation – the default redirect is to the URL pattern named `registration_complete`; this can be overridden in subclasses by changing `success_url` or implementing `get_success_url()`

Upon successful activation, the default redirect is to the URL pattern named `registration_activation_complete`; this can be overridden in subclasses by implementing `get_success_url()`.

## 4.2 How account data is stored for activation

During registration, a new instance of `django.contrib.auth.models.User` is created to represent the new account, with the `is_active` field set to `False`. An email is then sent to the email address of the account, containing a link the user must click to activate the account; at that point the `is_active` field is set to `True`, and the user may log in normally.

Activation is handled by generating and storing an activation key in the database, using the following model:

### `class registration.models.RegistrationProfile`

A simple representation of the information needed to activate a new user account. This is **not** a user profile; it simply provides a place to temporarily store the activation key and determine whether a given account has been activated.

Has the following fields:

#### **user**

A `ForeignKey` to `django.contrib.auth.models.User`, representing the user account for which activation information is being stored.

#### **activation\_key**

A 40-character `CharField`, storing the activation key for the account. The activation key is the hexdigest of a SHA1 hash.

#### **activated**

A `BooleanField`, storing whether or not the the `User` has activated their account. Storing this independent from `self.user.is_active` allows accounts to be deactivated and prevent being reactivated without authorization.

And the following methods:

#### **activation\_key\_expired()**

Determines whether this account's activation key has expired, and returns a boolean (`True` if expired, `False` otherwise). Uses the following algorithm:

- 1.If `activated` is `True`, the account has already been activated and so the key is considered to have expired.
- 2.Otherwise, the date of registration (obtained from the `date_joined` field of `user`) is compared to the current date; if the span between them is greater than the value of the setting `ACCOUNT_ACTIVATION_DAYS`, the key is considered to have expired.

**Return type** `bool`

#### **send\_activation\_email** (`site`[, `request`])

Sends an activation email to the address of the account.

The activation email will make use of two templates: `registration/activation_email_subject.txt` and `registration/activation_email.txt`, which are used for the subject of the email and the body of the email, respectively. Each will receive the following context:

**activation\_key** The value of `activation_key`.

**expiration\_days** The number of days the user has to activate, taken from the setting `ACCOUNT_ACTIVATION_DAYS`.

**site** An object representing the site on which the account was 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.models.RequestSite` (if not). Consult the [documentation for the Django sites framework](#) for details regarding these objects' interfaces.

**request** Django's `HttpRequest` object for better flexibility. When provided, it will also provide all the data via installed template context processors which can provide even more flexibility by using many Django's provided and custom template context processors to provide more variables to the template.

Because email subjects must be a single line of text, the rendered output of `registration/activation_email_subject.txt` will be forcibly condensed to a single line.

#### Parameters

- **site** (`django.contrib.sites.models.Site` or `django.contrib.sites.models.RequestSite`) – An object representing the site on which account was registered.
- **request** (`django.http.request.HttpRequest`) – Optional Django's `HttpRequest` object from view which if supplied will be passed to the template via `RequestContext`. As a consequence, all installed `TEMPLATE_CONTEXT_PROCESSORS` will be used to populate context.

**Return type** None

Additionally, `RegistrationProfile` has a custom manager (accessed as `RegistrationProfile.objects`):

**class** `registration.models.RegistrationManager`

This manager provides several convenience methods for creating and working with instances of `RegistrationProfile`:

**activate\_user** (*activation\_key*)

Validates *activation\_key* and, if valid, activates the associated account by setting its `is_active` field to `True`. To prevent re-activation of accounts, the *activated* of the `RegistrationProfile` for the account will be set to `True` after successful activation.

Returns the `User` instance representing the account if activation is successful, `False` otherwise.

**Parameters** *activation\_key* (*string, a 40-character SHA1 hexdigest*) – The activation key to use for the activation.

**Return type** `User` or `bool`

**delete\_expired\_users** ()

Removes expired instances of `RegistrationProfile`, and their associated user accounts, from the database. This is useful as a periodic maintenance task to clean out accounts which registered but never activated.

Accounts to be deleted are identified by searching for instances of `RegistrationProfile` with expired activation keys and with associated user accounts which are inactive (have their `is_active` field set to `False`). To disable a user account without having it deleted, simply delete its associated `RegistrationProfile`; any `User` which does not have an associated `RegistrationProfile` will not be deleted.

A custom management command is provided which will execute this method, suitable for use in cron jobs or other scheduled maintenance tasks: `manage.py cleanupregistration`.

**Return type** None

**create\_inactive\_user** (*site*[, *new\_user*=None, *send\_email*=True, *request*=None, *\*\*user\_info* ])

Creates a new, inactive user account and an associated instance of *RegistrationProfile*, sends the activation email and returns the new *User* object representing the account.

**Parameters**

- **new\_user** (*django.contrib.auth.models.AbstractBaseUser*) – The user instance.
- **user\_info** (*dict*) – The fields to use for the new account.
- **site** (*django.contrib.sites.models.Site* or *django.contrib.sites.models.RequestSite*) – An object representing the site on which the account is being registered.
- **send\_email** (*bool*) – If True, the activation email will be sent to the account (by calling *RegistrationProfile.send\_activation\_email()*). If False, no email will be sent (but the account will still be inactive)
- **request** (*django.http.request.HttpRequest*) – If *send\_email* parameter is True and if *request* is supplied, it will be passed to the email templates for better flexibility. Please take a look at the sample email templates for better explanation how it can be used.

**Return type** *User*

**create\_profile** (*user*)

Creates and returns a *RegistrationProfile* instance for the account represented by *user*.

The *RegistrationProfile* created by this method will have its *activation\_key* set to a SHA1 hash generated from a combination of the account's username and a random salt.

**Parameters** *user* (*User*) – The user account; an instance of *django.contrib.auth.models.User*.

**Return type** *RegistrationProfile*

---

## The “simple” (one-step) backend

---

As an alternative to *the default backend*, and an example of writing alternate workflows, `django-registration-redux` bundles a one-step registration system in `registration.backend.simple`. This backend’s workflow is deliberately as simple 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.

### 5.1 Configuration

To use this backend, simply include the URLconf `registration.backends.simple.urls` somewhere in your site’s own URL configuration. For example:

```
(r'^accounts/', include('registration.backends.simple.urls')),
```

No additional settings are required, but one optional setting is supported:

**REGISTRATION\_OPEN** A boolean (either `True` or `False`) indicating whether registration of new accounts is currently permitted. A default of `True` will be assumed if this setting is not supplied.

Upon successful registration, the default redirect is to the `registration_complete` view (at `accounts/register/complete/`).

The default form class used for account registration will be `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 `registration.backends.simple.views.RegistrationView` and either overriding `form_class` or implementing `get_form_class()`.



---

## Forms for user registration

---

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

**class** `registration.forms.RegistrationForm`

A simple form for registering an account. Has the following fields, all of which are required:

**username** The username to use for the new account. This is represented as a text input which validates that the username is unique, consists entirely of alphanumeric characters and underscores and is at most 30 characters in length.

**email** The email address to use for the new account. This is represented as a text input which accepts email addresses up to 75 characters in length.

**password1** The password to use for the new account. This represented as a password input (`input type="password"` in the rendered HTML).

**password2** The password to use for the new account. This represented as a password input (`input type="password"` in the rendered HTML). Password mismatches are recorded as errors of `password2`.

The constraints on usernames and email addresses match those enforced by Django's default authentication backend for instances of `django.contrib.auth.models.User`. The repeated entry of the password serves to catch typos.

**class** `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** `registration.forms.RegistrationFormUniqueEmail`

A subclass of `RegistrationForm` which enforces uniqueness of email addresses in addition to uniqueness of usernames.

**class** `registration.forms.RegistrationFormNoFreeEmail`

A subclass of `RegistrationForm` which disallows registration using addresses from some common free email providers. This can, in some cases, cut down on automated registration by spambots.

By default, the following domains are disallowed for email addresses:

- aim.com
- aol.com
- email.com
- gmail.com

- googlemail.com
- hotmail.com
- hushmail.com
- msn.com
- mail.ru
- mailinator.com
- live.com
- yahoo.com

To change this, subclass this form and set the class attribute `bad_domains` to a list of domains you wish to disallow.

## 6.1 Multiple Form Inheritance

Multiple *RegistrationForm* subclasses can be inherited into one class. For instance, if your project requires a terms of service and a unique email upon registration, those subclasses can be inherited into a single class. That would look like this:

```
class CustomForm(RegistrationFormTermsOfService, RegistrationFormUniqueEmail): pass
```

NOTE: If inheriting both *RegistrationFormNoFreeEmail* and *RegistrationFormUniqueEmail*, *RegistrationFormNoFreeEmail* must be inherited first, like this:

```
class CustomForm(RegistrationFormNoFreeEmail, RegistrationFormUniqueEmail): pass
```

---

## Registration views

---

In order to allow the utmost flexibility in customizing and supporting different workflows, django-registration-redux makes use of Django's support for [class-based views](#). Included in django-registration-redux are two base classes which can be subclassed to implement whatever workflow is required.

**class** `registration.views.RegistrationView`

A subclass of Django's [FormView](#), which provides the infrastructure for supporting user registration.

Useful places to override or customize on a `RegistrationView` subclass are:

**disallowed\_url**

The URL to redirect to when registration is disallowed. Should be a string, [the name of a URL pattern](#). Default value is `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 `registration.forms.RegistrationForm`.

**success\_url**

The URL to redirect to after successful registration. Should be a string, the name of a URL pattern, or a 3-tuple of arguments suitable for passing to Django's [redirect shortcut](#). Can be overridden on a per-request basis (see below). Default value is `None`, so that per-request customization is used instead.

**template\_name**

The template to use for user registration. Should be a string. Default value is `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.

**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 be a string, the name of a URL pattern, or a 3-tuple of arguments suitable for passing to Django's [redirect shortcut](#).

**registration\_allowed()**

Should return a boolean indicating whether user registration is allowed, either in general or for this specific request.

**register(*form*)**

Actually perform the business of registering a new user. Receives the registration `form`. Should return the new user who was just registered.

### **class** `registration.views.ActivationView`

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

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

#### **template\_name**

The template to use for user activation. Should be a string. Default value is `registration/activate.html`.

#### **activate** (*\*args, \*\*kwargs*)

Actually perform the business of activating a user account. Receives any positional or keyword arguments passed to the view. Should return the activated user account if activation is successful, or any value which evaluates `False` in boolean context if activation is unsuccessful.

#### **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 be a string, the name of a URL pattern, or a 3-tuple of arguments suitable for passing to Django's `redirect` shortcut.



---

## Custom signals used by django-registration-redux

---

Much of `django-registration-redux`'s customizability comes through the ability to write and use registration backends implementing 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 backend to support this represents an unnecessary amount of work. A more lightweight customization option is provided through two custom signals which backends are required 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.

### `registration.signals.user_activated`

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

**sender** The backend class used to activate the user.

**user** An instance of `django.contrib.auth.models.User` representing the activated account.

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

### `registration.signals.user_registered`

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

**sender** The backend class used to register the account.

**user** An instance of `django.contrib.auth.models.User` representing the new account.

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



---

## Frequently-asked questions

---

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

### 9.1 General

**What license is django-registration-redux under?** django-registration-redux 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-redux; if you did not receive a copy of this file, you can [view it online](#).

**Why are the forms and models for the default backend not in the default backend?** The model and manager used by *the default backend* are in `registration.models`, and the default form class (and various subclasses) are in `registration.forms`; logically, they might be expected to exist in `registration.backends.default`, but there are several reasons why that's not such a good idea:

1. Older versions of django-registration-redux made use of the model and form classes, and moving them would create an unnecessary backwards incompatibility: `import` statements would need to be changed, and some database updates would be needed to reflect the new location of the `RegistrationProfile` model.
2. Due to the design of Django's ORM, the `RegistrationProfile` model would end up with an `app_label` of `default`, which isn't particularly descriptive and may conflict with other applications. By keeping it in `registration.models`, it retains an `app_label` of `registration`, which more accurately reflects what it does and is less likely to cause problems.
3. Although the `RegistrationProfile` model and the various *form classes* are used by the default backend, they can and are meant to be reused as needed by other backends. Any backend which uses an activation step should feel free to reuse the `RegistrationProfile` model, for example, and the registration form classes are in no way tied to a specific backend (and cover a number of common use cases which will crop up regardless of the specific backend logic in use).

### 9.2 Installation and setup

**How do I install django-registration-redux?** Full instructions are available in *the quick start guide*.

**Do I need to put a copy of django-registration-redux in every project I use it in?** No; putting applications in your project directory is a very bad habit, and you should stop doing it. If you followed the instructions mentioned above, django-registration-redux was installed into a location that's on your Python import path, so you'll

only ever need to add `registration` to your `INSTALLED_APPS` setting (in any project, or in any number of projects), and it will work.

## 9.3 Configuration

### 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.

**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-registration-redux` provides a URLconf which *only* contains the patterns for the auth views, and which you can include in your own URLconf anywhere you'd like; it lives at `registration.auth_urls`.

**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.

## 9.4 Troubleshooting

### I've got functions listening for the registration/activation signals, but they're not getting called!

The most common cause of this is placing `django-registration-redux` in a sub-directory that's on your Python import path, rather than installing it directly onto the import path as normal. Importing from `django-registration-redux` in that case can cause various issues, including incorrectly connecting signal handlers. For example, if you were to place `django-registration-redux` inside a directory named `django_apps`, and refer to it in that manner, you would end up with a situation where your code does this:

```
from django_apps.registration.signals import user_registered
```

But `django-registration-redux` will be doing:

```
from registration.signals import user_registered
```

From Python's point of view, these import statements refer to two different objects in two different modules, and so signal handlers connected to the signal from the first import will not be called when the signal is sent using the second import.

To avoid this problem, follow the standard practice of installing `django-registration-redux` directly on your import path and always referring to it by its own module name: `registration` (and in general, it is always a good idea to follow normal Python practices for installing and using Django applications).

### I want to use custom templates, but django keeps using the admin templates instead of mine!

To fix this, make sure that in the `INSTALLED_APPS` of your `settings.py` the entry for the `registration` app is placed above `django.contrib.admin`.

## 9.5 Tips and tricks

**How do I log a user in immediately after registration or activation?** Take a look at the implementation of *the simple backend*, which logs a user in immediately after registration.

**How do I re-send an activation email?** Assuming you're using *the default backend*, a custom admin action is provided for this; in the admin for the *RegistrationProfile* model, simply click the checkbox for the user(s) you'd like to re-send the email for, then select the "Re-send activation emails" action.

**How do I manually activate a user?** In the default backend, a custom admin action is provided for this. In the admin for the *RegistrationProfile* model, click the checkbox for the user(s) you'd like to activate, then select the "Activate users" action.

**See also:**

- [Django's authentication documentation](#); Django's authentication system is used by django-registration-redux's default configuration.



**r**

`registration.backends.default`, 10  
`registration.backends.simple`, 14  
`registration.forms`, 15  
`registration.signals`, 20  
`registration.views`, 18





**A**

activate() (registration.views.ActivationView method), 20  
 activate\_user() (registration.models.RegistrationManager method), 13  
 activated (registration.models.RegistrationProfile attribute), 12  
 activation\_key (registration.models.RegistrationProfile attribute), 12  
 activation\_key\_expired() (registration.models.RegistrationProfile method), 12  
 ActivationView (class in registration.views), 19

**C**

create\_inactive\_user() (registration.models.RegistrationManager method), 13  
 create\_profile() (registration.models.RegistrationManager method), 14

**D**

delete\_expired\_users() (registration.models.RegistrationManager method), 13  
 disallowed\_url (registration.views.RegistrationView attribute), 19

**F**

form\_class (registration.views.RegistrationView attribute), 19

**G**

get\_form\_class() (registration.views.RegistrationView method), 19  
 get\_success\_url() (registration.views.ActivationView method), 20  
 get\_success\_url() (registration.views.RegistrationView method), 19

**R**

register() (registration.views.RegistrationView method), 19  
 registration.backends.default (module), 10  
 registration.backends.simple (module), 14  
 registration.forms (module), 15  
 registration.signals (module), 20  
 registration.views (module), 18  
 registration\_allowed() (registration.views.RegistrationView method), 19  
 RegistrationForm (class in registration.forms), 17  
 RegistrationFormNoFreeEmail (class in registration.forms), 17  
 RegistrationFormTermsOfService (class in registration.forms), 17  
 RegistrationFormUniqueEmail (class in registration.forms), 17  
 RegistrationManager (class in registration.models), 13  
 RegistrationProfile (class in registration.models), 12  
 RegistrationView (class in registration.views), 19

**S**

send\_activation\_email() (registration.models.RegistrationProfile method), 12  
 success\_url (registration.views.RegistrationView attribute), 19

**T**

template\_name (registration.views.ActivationView attribute), 20  
 template\_name (registration.views.RegistrationView attribute), 19

**U**

user (registration.models.RegistrationProfile attribute), 12  
 user\_activated (in module registration.signals), 21  
 user\_registered (in module registration.signals), 21