django-registration Documentation

Release 0.8

James Bennett

November 21, 2015

Contents

This documentation covers the 0.8 release of django-registration, a simple but extensible application providing user registration functionality for Django-powered websites.

Although nearly all aspects of the registration process are customizable, the default setup of django-registration attempts to cover the most common use case: two-phase registration, consisting of initial signup followed by a confirmation email which contains instructions for activating the new account.

To get up and running quickly, consult the *quick-start guide*, which describes all the necessary steps to install django-registration 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:

CHAPTER 1

Quick start guide

Before installing django-registration, you'll need to have a copy of Django already installed. For the 0.8 release, Django 1.1 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

There are several ways to install django-registration:

- Automatically, via a package manager.
- Manually, by downloading a copy of the release package and installing it yourself.
- Manually, by performing a Mercurial 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 most popular are easy_install and pip. Either can be used to install django-registration.

Using `easy_install`, type:

```
easy_install -Z django-registration
```

Note that the `-Z` flag is required, to tell `easy_install` not to create a zipped package; zipped packages prevent certain features of Django from working properly.

Using `pip`, type:

```
pip install django-registration
```

It is also possible that your operating system distributor provides a packaged version of django-registration (for example, Debian GNU/Linux provides a package, installable via `apt-get-install python-django-registration`). Consult your operating system's package list for details, but be aware that third-party distributions may be providing older versions of django-registration, 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 and install it manually. The latest release package can be downloaded from django-registration'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-0.8.tar.gz` at a command line on Linux, Mac OS X or other Unix-like systems). This will create the directory `django-registration-0.8`, 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 Mercurial checkout

If you'd like to try out the latest in-development code, you can obtain it from the django-registration repository, which is hosted at Bitbucket and uses Mercurial for version control. To obtain the latest code and documentation, you'll need to have Mercurial installed, at which point you can type:

```
hg clone http://bitbucket.org/ubernostrum/django-registration/
```

You can also obtain a copy of a particular release of django-registration by specifying the `-r` argument to `hg 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 0.8 release, type:

```
hg clone -r v0.8 http://bitbucket.org/ubernostrum/django-registration/
```

In either case, this will create a copy of the django-registration Mercurial repository on your computer; you can then add the `django-registration` 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 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 Required 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.

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.
```

Once you've done this, run `manage.py syncdb` 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*, 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.

1.2.3 Required templates

In the default setup, you will need to create several templates required by django-registration, and possibly additional templates required by views in `django.contrib.auth`. The templates requires 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.

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

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.

`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.

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 defaults to plain-text email, and so these templates should simply output plain text rather than HTML.

To make use of the views from `django.contrib.auth` (which are set up for you by the default URLconf mentioned 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 2

Release notes

The 0.8 release of django-registration represents a complete rewrite of the previous codebase, and as such introduces a number of new features and greatly enhances the flexibility and customizability of django-registration. This document summarizes those features; for a list of changes which impact existing installations, consult *the upgrade guide*.

2.1 The backend system

The largest overall change consists of factoring out the logic of user registration into pluggable/swappable backend classes. The *registration views* now accept a (required) argument, `backend`, which indicates the backend class to use, and that class has full control over the registration (and, if needed, activation) process, including:

- Determining whether registration will be allowed at all, on a per-request basis.
- Specifying a form class to use for account registration.
- Implementing the actual process of account creation.
- Determining whether a separate activation step is needed, and if so what it will entail.
- Specifying actions to take (e.g., redirects, automatic login, etc.) following successful registration or activation.

For full details, see the documentation for *the backend API*.

The workflow used by previous releases of django-registration (two-step registration/activation) has been implemented using this system, and is shipped as *the default backend* in django-registration 0.8.

2.2 Other new features

An alternate *one-step registration system* is provided, for use by sites which do not require a two-step registration/activation system.

During the registration and (optional) activation process, *custom signals* are now sent, allowing easy injection of custom processing into the registration workflow without needing to write a full backend.

The default backend now supplies several custom admin actions to make the process of administering a site with django-registration simpler.

The `activate()` view now supplies any captured keyword arguments from the URL (in the case of the default backend, this is the activation key) to its template in case of unsuccessful activation; this greatly simplifies the process of determining why activation failed and displaying appropriate error messages.

CHAPTER 3

Upgrade guide

The 0.8 release of django-registration represents a complete rewrite of the previous codebase, and introduces several new features which greatly enhance the customizability and extensibility of django-registration. Whenever possible, changes were made in ways which preserve backwards compatibility with previous releases, but some changes to existing installations will still be required in order to upgrade to 0.8. This document provides a summary of those changes, and of the new features available in the 0.8 release.

3.1 Django version requirement

As of 0.8, django-registration requires Django 1.3 or newer; older Django releases may work, but are officially unsupported.

3.2 Backwards-incompatible changes

If you're upgrading from an older release of django-registration, and if you were using the default setup (i.e., the included default URLconf and no custom URL patterns or custom arguments to views), most things will continue to work as normal (although you will need to create one new template; see the section on views below). However, the old default URLconf has been deprecated and will be removed in version 1.0 of django-registration, so it is recommended that you begin migrating now. To do so, change any use of `registration.urls` to `registration.backends.default.urls`. For example, if you had the following in your root URLconf:

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

you should change it to:

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

The older include will continue to work until django-registration 1.0; in 0.8 it raises a `PendingDeprecationWarning` (which is ignored by default in Python), in 0.9 it will raise `DeprecationWarning` (which will begin printing warning messages on import) and in 1.0 it will be removed entirely.

3.2.1 Changes to registration views

The views used to handle user registration have changed significantly as of django-registration 0.8. Both views now require the keyword argument `backend`, which specifies the *registration backend* to use, and so any URL pattern for these views must supply that argument. The URLconf provided with *the default backend* properly passes this argument.

The `profile_callback` argument of the *`register()`* view has been removed; the functionality it provided can now be implemented easily via a custom backend, or by connecting listeners to *the signals sent during the registration process*.

The *`activate()`* view now issues a redirect upon successful activation; in the default backend this is to the URL pattern named `registration_activation_complete`; in the default setup, this will redirect to a view which renders the template `registration/activation_complete.html`, and so this template should be present when using the default backend and default configuration. Other backends can specify the location to redirect to through their `post_activation_redirect()` method, and this can be overridden on a case-by-case basis by passing the (new) keyword argument `success_url` to the `activate()` view. On unsuccessful activation, the `activate()` view still displays the same template, but its context has changed: the context will simply consist of any keyword arguments captured in the URL and passed to the view.

3.2.2 Changes to registration forms

Previously, the form used to collect data during registration was expected to implement a `save()` method which would create the new user account. This is no longer the case; creating the account is handled by the backend, and so any custom logic should be moved into a custom backend, or by connecting listeners to *the signals sent during the registration process*.

3.2.3 Changes to the `RegistrationProfile` model

The *`create_inactive_user()`* method of *`RegistrationManager`* now has an additional required argument: `site`. This allows django-registration to easily be used regardless of whether `django.contrib.sites` is installed, since a `RequestSite` object can be passed in place of a regular `Site` object.

The *`user_registered`* signal is no longer sent by `create_inactive_user()`, and the *`user_activated`* signal is no longer sent by *`activate_user()`*; these signals are now sent by the backend after these methods have been called. Note that *these signals* were added after the django-registration 0.7 release but before the refactoring which introduced *the backend API*, so only installations which were tracking the in-development codebase will have made use of them.

The sending of activation emails has been factored out of `create_inactive_user()`, and now exists as the method *`send_activation_email()`* on instances of `RegistrationProfile`.

CHAPTER 4

User registration backends

At its core, django-registration is built around the idea of pluggable backends which can implement different workflows for user registration. Although *the default backend* uses a common two-phase system (registration followed by activation), backends are generally free to implement any workflow desired by their authors.

This is deliberately meant to be complementary to Django's own pluggable authentication backends; a site which uses an OpenID authentication backend, for example, can and should make use of a registration backend which handles signups via OpenID. And, like a Django authentication backend, a registration backend is simply a class which implements a particular standard API (described below).

This allows for a great deal of flexibility in the actual workflow of registration; backends can, for example, implement any of the following (not an exhaustive list):

- One-step (register, and done) or multi-step (register and activate) signup.
- Invitation-based registration.
- Selectively allowing or disallowing registration (e.g., by requiring particular credentials to register).
- Enabling/disabling registration entirely.
- Registering via sources other than a standard username/password, such as OpenID.
- Selective customization of the registration process (e.g., using different forms or imposing different requirements for different types of users).

4.1 Specifying the backend to use

To determine which backend to use, the *views in django-registration* accept a keyword argument `backend`; in all cases, this should be a string containing the full dotted Python import path to the backend class to be used. So, for example, to use the default backend, you'd pass the string `'registration.backends.default.DefaultBackend'` as the value of the `backend` argument (and the default URLconf included with that backend does so). The specified backend class will then be imported and instantiated (by calling its constructor with no arguments), and the resulting instance will be used for all backend-specific functionality.

If the specified backend class cannot be imported, django-registration will raise `django.core.exceptions.ImproperlyConfigured`.

4.2 Backend API

To be used as a registration backend, a class must implement the following methods. For many cases, subclassing the default backend and selectively overriding behavior will be suitable, but for other situations (e.g., workflows significantly different from the default) a full implementation is needed.

4.2.1 register(request, **kwargs)

This method implements the logic of actually creating the new user account. Often, but not necessarily always, this will involve creating an instance of `django.contrib.auth.models.User` from the supplied data.

This method will only be called after a signup form has been displayed, and the data collected by the form has been properly validated.

Arguments to this method are:

`request` The Django HttpRequest object in which a new user is attempting to register.

`**kwargs` A dictionary of the `cleaned_data` from the signup form.

After creating the new user account, this method should create or obtain an instance of `django.contrib.auth.models.User` representing that account. It should then send the signal *`registration.signals.user_registered`*, with three arguments:

`sender` The backend class (e.g., `self.__class__`).

`user` The `User` instance representing the new account.

`request` The `HttpRequest` in which the user registered.

Finally, this method should return the `User` instance.

4.2.2 activate(request, **kwargs)

For workflows which require a separate activation step, this method should implement the necessary logic for account activation.

Arguments to this method are:

`request` The Django `HttpRequest` object in which the account is being activated.

`**kwargs` A dictionary of any additional arguments (e.g., information captured from the URL, such as an activation key) received by the *`activate()`* view. The combination of the `HttpRequest` and this additional information must be sufficient to identify the account which will be activated.

If the account cannot be successfully activated (for example, in the default backend if the activation period has expired), this method should return `False`.

If the account is successfully activated, this method should create or obtain an instance of `django.contrib.auth.models.User` representing the activated account. It should then send the signal *`registration.signals.user_activated`*, with three arguments:

`sender` The backend class.

`user` The `User` instance representing the activated account.

`request` The `HttpRequest` in which the user activated.

This method should then return the `User` instance.

For workflows which do not require a separate activation step, this method can and should raise `NotImplementedError`.

4.2.3 registration_allowed(request)

This method returns a boolean value indicating whether the given `HttpRequest` is permitted to register a new account (`True` if registration is permitted, `False` otherwise). It may determine this based on some aspect of the `HttpRequest` (e.g., the presence or absence of an invitation code in the URL), based on a setting (in the default backend, a setting can be used to disable registration), information in the database or any other information it can access.

Arguments to this method are:

`request` The Django `HttpRequest` object in which a new user is attempting to register.

If this method returns `False`, the *`register()`* view will not display a form for account creation; instead, it will issue a redirect to a URL explaining that registration is not permitted.

4.2.4 get_form_class(request)

This method should return a form class – a subclass of `django.forms.Form` – suitable for use in registering users with this backend. As such, it should collect and validate any information required by the backend's `register` method.

Arguments to this method are:

`request` The Django `HttpRequest` object in which a new user is attempting to register.

4.2.5 post_registration_redirect(request, user)

This method should return a location to which the user will be redirected after successful registration. This should be a tuple of `(to, args, kwargs)`, suitable for use as the arguments to Django's "redirect" shortcut.

Arguments to this method are:

`request` The Django `HttpRequest` object in which the user registered.

`user` The `User` instance representing the new user account.

4.2.6 post_activation_redirect(request, user)

For workflows which require a separate activation step, this method should return a location to which the user will be redirected after successful activation. This should be a tuple of `(to, args, kwargs)`, suitable for use as the arguments to Django's "redirect" shortcut.

Arguments to this method are:

`request` The Django `HttpRequest` object in which the user activated.

`user` The `User` instance representing the activated user account.

For workflows which do not require a separate activation step, this method can and should raise `NotImplementedError`.

The default backend

A default *registration backend* is bundled with django-registration, as the class `registration.backends.default.DefaultBackend`, 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.

5.1 Default behavior and configuration

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.

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.

Upon successful registration – not activation – the default redirect is to the URL pattern named `registration_complete`; this can be overridden by passing the keyword argument `success_url` to the `register()` view.

Upon successful activation, the default redirect is to the URL pattern named `registration_activation_complete`; this can be overridden by passing the keyword argument `success_url` to the `activate()` view.

5.2 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. Initially, the activation key is the hexdigest of a SHA1 hash; after activation, this is reset to *`ACTIVATED`*.

Additionally, one class attribute exists:

`ACTIVATED`
A constant string used as the value of *`activation_key`* for accounts which have been activated.

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 *`activation_key`* is *`ACTIVATED`*, 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*`)`
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.

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.

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 *`activation_key`* of the *`RegistrationProfile`* for the account will be set to *`RegistrationProfile.ACTIVATED`* 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`(*username*, *email*, *password*, *site*[, *send_email*])

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

- **`username`** (*string*) – The username to use for the new account.
- **`email`** (*string*) – The email address to use for the new account.
- **`password`** (*string*) – The password 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)

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`

CHAPTER 6

The “simple” (one-step) backend

As an alternative to *the default backend*, and an example of writing *registration backends*, django-registration 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.

6.1 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 URL specified by the `get_absolute_url()` method of the newly-created `User` object; by default, this will be `/users/<username>/`, although it can be overridden in either of two ways:

1. Specify a custom URL pattern for the *`register()`* view, passing the keyword argument `success_url`.
2. Override the default `get_absolute_url()` of the `User` model in your Django configuration, as covered in Django’s settings documentation.

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 `register()` view and passing the keyword argument `form_class`.

Note that because this backend does not use an activation step, attempting to use the *`activate()`* view with this backend or calling the backend’s `activate()` or `post_activation_redirect()` methods will raise `NotImplementedError`.

CHAPTER 7

Forms for user registration

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 *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).

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.

Because it does not apply to any single field of the form, the validation error for mismatched passwords is attached to the form itself, and so must be accessed via the form's `non_field_errors()` method.

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.

CHAPTER 8

Registration views

In order to allow users to register using whatever workflow is implemented by the *registration backend* in use, django-registration provides two views. Both are designed to allow easy configurability without writing or rewriting view code.

`registration.views.`**`activate`**(*request*, *backend*[, *template_name*[, *success_url*[, *extra_context*[, ***kwargs*]]]])

Activate a user's account, for workflows which require a separate activation step.

The actual activation of the account will be delegated to the backend specified by the `backend` keyword argument; the backend's `activate()` method will be called, passing the `HttpRequest` and any keyword arguments captured from the URL, and will be assumed to return a `User` if activation was successful, or a value which evaluates to `False` in boolean context if not.

Upon successful activation, the backend's `post_activation_redirect()` method will be called, passing the `HttpRequest` and the activated `User` to determine the URL to redirect the user to. To override this, pass the argument `success_url` (see below).

On unsuccessful activation, will render the template `registration/activate.html` to display an error message; to override thise, pass the argument `template_name` (see below).

Context

The context will be populated from the keyword arguments captured in the URL. This view uses `RequestContext`, so variables populated by context processors will also be present in the context.

Parameters

- **`backend`** (*string*) – The dotted Python path to the backend class to use.
- **`extra_context`** (*dict*) – Optionally, variables to add to the template context. Any callable object in this dictionary will be called to produce the final result which appears in the context.
- **`template_name`** (*string*) – Optional. A custom template name to use. If not specified, this will default to `registration/activate.html`.
- **`**kwargs`** – Any keyword arguments captured from the URL, such as an activation key, which will be passed to the backend's `activate()` method.

`registration.views.`**`register`**(*request*, *backend*[, *success_url*[, *form_class*[, *disallowed_url*[, *template_name*[, *extra_context*]]]]])

Allow a new user to register an account.

The actual registration of the account will be delegated to the backend specified by the `backend` keyword argument. The backend is used as follows:

1. The backend's `registration_allowed()` method will be called, passing the `HttpRequest`, to determine whether registration of an account is to be allowed; if not, a redirect is issued to a page indicating that registration is not permitted.
2. The form to use for account registration will be obtained by calling the backend's `get_form_class()` method, passing the `HttpRequest`. To override this, pass the keyword argument `form_class`.
3. If valid, the form's `cleaned_data` will be passed (as keyword arguments, and along with the `HttpRequest`) to the backend's `register()` method, which should return a `User` object representing the new account.
4. Upon successful registration, the backend's `post_registration_redirect()` method will be called, passing the `HttpRequest` and the new `User`, to determine the URL to redirect to. To override this, pass the keyword argument `success_url`.

Context

`form` The form instance being used to collect registration data.

This view uses `RequestContext`, so variables populated by context processors will also be present in the context.

Parameters

- **`backend`** (*string*) – The dotted Python path to the backend class to use.
- **`disallowed_url`** (*string*) – The URL to redirect to if registration is not permitted (e.g., if registration is closed). This should be a string suitable for passing as the `to` argument to Django's "redirect" shortcut. If not specified, this will default to `registration_disallowed`.
- **`extra_context`** (*dict*) – Optionally, variables to add to the template context. Any callable object in this dictionary will be called to produce the final result which appears in the context.
- **`form_class`** (subclass of `django.forms.Form`) – The form class to use for registration; this should be some subclass of `django.forms.Form`. If not specified, the backend's `get_form_class()` method will be called to obtain the form class.
- **`success_url`** (*string*) – The URL to redirect to after successful registration. This should be a string suitable for passing as the `to` argument to Django's "redirect" shortcut. If not specified, the backend's `post_registration_redirect()` method will be called to obtain the URL.
- **`template_name`** (*string*) – Optional. A custom template name to use. If not specified, this will default to `registration/registration_form.html`.

Custom signals used by django-registration

Much of django-registration'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.

CHAPTER 10

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.

10.1 General

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 modifiying 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 <http://bitbucket.org/ubernostrum/django-registration/src/tip/LICENSE>.

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 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).

10.2 Installation and setup

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

Do I need to put a copy of django-registration 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 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.

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 generally 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 backend* you use, including the variables which end up in template contexts. Since django-registration has no way of knowing in advance what backend 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.

10.3 Configuration

Do I need to rewrite the views to change the way they behave? No. There are several ways you can customize behavior without making any changes whatsoever:

- Pass custom arguments – e.g., to specify forms, template names, etc. – to *the registration views*.
- Use the *signals* sent by the views to add custom behavior.
- Write a custom *registration backend* which implements the behavior you need, and have the views use your backend.

If none of these are sufficient, your best option is likely to simply write your own views; however, it is hoped that the level of customization exposed by these options will be sufficient for nearly all user-registration workflows.

How do I pass custom arguments to the views? Part 3 of the official Django tutorial, when it introduces generic views, covers the necessary mechanism: simply provide a dictionary of keyword arguments in your URLconf.

Does that mean I should rewrite django-registration's default URLconf? No; if you'd like to pass custom arguments to the registration views, simply write and include your own URLconf instead of including the default one provided with django-registration.

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 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.

10.4 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 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 in that case can cause various issues, including incorrectly connecting signal handlers. For example, if you were to place django-registration 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 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 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).

10.5 Tips and tricks

How do I log a user in immediately after registration or activation? You can most likely do this simply by writing a function which listens for the appropriate *signal*; your function should set the `backend` attribute of the user to the correct authentication backend, and then call `django.contrib.auth.login()` to log the user in.

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's default configuration.
- django-profiles, an application which provides simple user-profile management.

Index