django-registration Documentation

Release 2.3

James Bennett

Sep 25, 2017

Installation and configuration

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

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

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

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

CHAPTER 1

Installation guide

Before installing django-registration, you'll need to have a copy of Django already installed. For information on obtaining and installing Django, consult the Django download page, which offers convenient packaged downloads and installation instructions.

The 2.3 release of django-registration supports Django 1.8, 1.9, 1.10 and 1.11, on the following Python versions:

- Django 1.8 supports Python 2.7, 3.3, 3.4 and 3.5.
- Django 1.9 supports Python 2.7, 3.4 and 3.5.
- Django 1.10 supports Python 2.7, 3.4 and 3.5.
- Django 1.11 supports Python 2.7, 3.4, 3.5 and 3.6.

It is expected that django-registration 2.3 will work without modification on Python 3.6 once it is released.

Important: Python 3.2

Although Django 1.8 supported Python 3.2 at the time of its release, the Python 3.2 series has reached end-of-life, and as a result support for Python 3.2 has been dropped from django-registration.

Normal installation

The preferred method of installing django-registration is via `pip`, the standard Python package-installation tool. If you don't have `pip`, instructions are available for how to obtain and install it. If you're using Python 2.7.9 or later (for Python 2) or Python 3.4 or later (for Python 3), `pip` came bundled with your installation of Python.

Once you have `pip`, type:

```
pip install django-registration
```

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

Installing from a source checkout

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

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

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

From there, you can use normal git commands to check out the specific revision you want, and install it using `pip install -e .` (the `-e` flag specifies an "editable" install, allowing you to change code as you work on django-registration, and have your changes picked up automatically).

Next steps

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

CHAPTER 2

Quick start guide

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

The next steps will depend on which registration workflow you'd like to use. There are three workflows built in to django-registration; one is included largely for backwards compatibility with older releases, while the other two are recommended for new installations. Those two are:

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

The guide below covers use of these two workflows.

Important: Django's authentication system must be installed

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

Configuring the HMAC activation workflow

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

Required settings

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

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

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

```
ACCOUNT_ACTIVATION_DAYS = 7 # One-week activation window; you may, of course, use a␣
↪different value.
```

You'll also need to have `django.contrib.auth` in your `INSTALLED_APPS` setting, since all of the registration workflows in django-registration make use of it.

Warning: You should **not** add `registration` to your `INSTALLED_APPS` setting if you're following this document. This section is walking you through setup of the *the HMAC activation workflow*, and that does not make use of any custom models or other features which require `registration` to be in `INSTALLED_APPS`. Only add `registration` to your `INSTALLED_APPS` setting if you're using *the model-based activation workflow*, or something derived from it.

Setting up URLs

Each bundled registration workflow in django-registration includes a Django URLconf which sets up URL patterns for *the views in django-registration*, as well as several useful views in `django.contrib.auth` (e.g., login, logout, password change/reset). The URLconf for the HMAC activation workflow can be found at `registration.backends.hmac.urls`, and so can 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:

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

urlpatterns = [
    # Other URL patterns ...
    url(r'^accounts/', include('registration.backends.hmac.urls')),
    # More URL patterns ...
]
```

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

The following URL names are defined by this URLconf:

- `registration_register` is the account-registrationview..
- `registration_complete` is the post-registration success message.
- `registration_activate` is the account-activation view.
- `registration_activation_complete` is the post-activation success message.
- `registration_disallowed` is a message indicating registration is not currently permitted.

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.

Required templates

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

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

registration/registration_form.html

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

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

registration/registration_complete.html

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

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 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.

`user` The user registering for the new account.

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

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.

`user` The user registering for the new account.

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

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 plaintext email, and so these templates should output plain text rather than HTML.

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

Configuring the one-step workflow

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

The one-step workflow does not require any models other than those provided by Django's own authentication system, so only `django.contrib.auth` needs to be in your `INSTALLED_APPS` setting.

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

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

urlpatterns = [
    # Other URL patterns ...
    url(r'^accounts/', include('registration.backends.simple.urls')),
    # More URL patterns ...
]
```

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

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

Finally, you will need to create one template: `registration/registration_form.html`. See *the list of templates above* for details of this template's context.

CHAPTER 3

The HMAC activation workflow

The HMAC workflow, found in `registration.backends.hmac`, implements a two-step registration process (signup, followed by activation), but unlike the older *model-based activation workflow* uses no models and does not store its activation key; instead, the activation key sent to the user is a timestamped, HMAC-verified value.

Unless you need to maintain compatibility in an existing install of django-registration which used the model-based workflow, it's recommended you use the HMAC activation workflow for two-step signup processes.

Behavior and configuration

Since this workflow does not make use of any additional models beyond the user model (either Django's default `django.contrib.auth.models.User`, or *a custom user model*), *do not* add `registration` to your `INSTALLED_APPS` setting.

You will need to configure URLs, however. A default URLconf is provided, which you can `include()` in your URL configuration; that URLconf is `registration.backends.hmac.urls`. For example, to place user registration under the URL prefix `/accounts/`, you could place the following in your root URLconf:

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

urlpatterns = [
    # Other URL patterns ...
    url(r'^accounts/', include('registration.backends.hmac.urls')),
    # More URL patterns ...
]
```

That URLconf also sets up the views from `django.contrib.auth` (login, logout, password reset, etc.), though if you want those views at a different location, you can `include()` the URLconf `registration.auth_urls` to place only the `django.contrib.auth` views at a specific location in your URL hierarchy.

Note: URL patterns for activation

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

The default URL pattern for the activation view in `registration.backends.hmac.urls` handles this for you.

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

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

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

Views

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

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

class `registration.backends.hmac.views.`**`RegistrationView`**

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

Important customization points unique to this class are:

`create_inactive_user`(*form*)

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

`get_activation_key`(*user*)

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

`get_email_context`(*activation_key*)

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

`send_activation_email`(*user*)

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

`email_body_template`

A string specifying the template to use for the body of the activation email. Default is `"registration/activation_email.txt"`.

`email_subject_template`

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

class registration.backends.hmac.views.**ActivationView**
A subclass of *registration.views.ActivationView* implementing the activation portion of this workflow.

Important customization points unique to this class are:

get_user(*username*)
Given a username (determined by the activation key), look up and return the corresponding instance of the user model. Returns `None` if no such instance exists. In the base implementation, will include `is_active=False` in the query to avoid re-activation of already-active accounts.

validate_key(*activation_key*)
Given the activation key, verifies that it carries a valid signature and a timestamp no older than the number of days specified in the setting `ACCOUNT_ACTIVATION_DAYS`, and returns the username from the activation key. Returns `None` if the activation key has an invalid signature or if the timestamp is too old.

How it works

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

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

Comparison to the model-activation workflow

The primary advantage of the HMAC activation workflow is that it requires no persistent storage of the activation key. However, this means there is no longer an automated way to differentiate accounts which have been purposefully deactivated (for example, as a way to ban a user) from accounts which failed to activate within a specified window. Additionally, it is possible a user could, if manually deactivated, re-activate their account if still within the activation window; for this reason, when using the `is_active` field to "ban" a user, it is best to also set the user's password to an unusable value (i.e., by calling set_unusable_password() for that user). Calling `set_unusable_password()` will also make it easier to query for manually-deactivated users, as their passwords will (when using Django's default `User` implementation) begin with the exclamation mark (`!`) character.

Since the HMAC activation workflow does not use any models, it also does not make use of the admin interface and thus does not offer a convenient way to re-send an activation email. Users who have difficulty receiving the activation email can be manually activated by a site administrator.

However, the reduced overhead of not needing to store the activation key makes this generally preferable to *the model-based workflow*.

Security considerations

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

In particular, the activation key is of the form:

```
encoded_username:timestamp:signature
```

where `encoded_username` is the username of the new account, (URL-safe) base64-encoded, `timestamp` is a base62-encoded timestamp of the time the user registered, and `signature` is a (URL-safe) base64-encoded HMAC of the username and timestamp.

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

CHAPTER 4

The one-step workflow

As an alternative to the *HMAC* and *model-based* two-step (registration and activation) workflows, django-registration bundles a one-step registration workflow in `registration.backends.simple`. This 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.

Configuration

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

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

urlpatterns = [
    # Other URL patterns ...
    url(r'^accounts/', include('registration.backends.simple.urls')),
    # More URL patterns ...
]
```

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

Upon successful registration, the user will be redirected to the site's home page – the URL `/`. This can be changed by subclassing `registration.backends.simple.views.RegistrationView` and overriding the method `get_success_url()`.

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()*, and specifying the custom subclass in your URL patterns.

Templates

The one-step workflow uses only one custom template:

registration/registration_form.html

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

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

CHAPTER 5

The model-based activation workflow

This workflow implements a two-step – registration, followed by activation – process for user signup.

Note: Use of the model-based workflow is discouraged

The model-based activation workflow was originally the *only* workflow built in to django-registration, and later was the default one. However, it no longer represents the best practice for registration with modern versions of Django, and so it continues to be included only for backwards compatibility with existing installations of django-registration.

If you're setting up a new installation and want a two-step process with activation, it's recommended you use *the HMAC activation workflow* instead.

Also, note that this workflow was previously found in `registration.backends.default`, and imports from that location still function in django-registration 2.3 but now raise deprecation warnings. The correct location going forward is `registration.backends.model_activation`.

Default behavior and configuration

To make use of this workflow, add `registration` to your `INSTALLED_APPS`, run `manage.py migrate` to install its model, and include the URLconf `registration.backends.model_activation.urls` at whatever location you choose in your URL hierarchy. For example:

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

urlpatterns = [
    # Other URL patterns ...
    url(r'^accounts/', include('registration.backends.model_activation.urls')),
    # More URL patterns ...
]
```

This workflow makes use of the following settings:

- *ACCOUNT_ACTIVATION_DAYS*

- *REGISTRATION_OPEN*

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

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

Upon successful registration – not activation – the user will be redirected to the URL pattern named `registration_complete`.

Upon successful activation, the user will be redirected to the URL pattern named `registration_activation_complete`.

This workflow uses the same templates and contexts as *the HMAC activation workflow*, which is covered in *the quick-start guide*. Refer to the quick-start guide for documentation on those templates and their contexts.

How account data is stored for activation

During registration, a new instance of the user model (by default, Django's `django.contrib.auth.models.User` – see *the custom user documentation* for notes on using a different model) 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 representation of the information needed to activate a new user account. This is **not** a user profile; it just provides a place to temporarily store the activation key and determine whether a given account has been activated.

Has the following fields:

`user`
A `OneToOneField` to the user model, 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 hex digest 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`.

user The user registering for the new account.

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.

Note that, to avoid header-injection vulnerabilities, 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

expired()
Deprecated since version 2.3: This method is *deprecated* and scheduled to be removed in django-registration 3.0.

Return instances of *RegistrationProfile* corresponding to expired users. A user is considered to be "expired" if:

- The activation key of the user's *RegistrationProfile* is not set to *RegistrationProfile.ACTIVATED*, and
- The user's `is_active` field of is `False`, and

- The user's `date_joined` field is more than *ACCOUNT_ACTIVATION_DAYS* in the past.

Return type `QuerySet` of *RegistrationProfile*

delete_expired_users()

Deprecated since version 2.3: This method is *deprecated* and scheduled to be removed in django-registration 3.0, as is the referenced `cleanupregistration` management command.

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.

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(*form*, *site*, *send_email=True*)

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

- **form** – A bound instance of a subclass of *RegistrationForm* representing the (already-validated) data the user is trying to register with.
- **site** – An object representing the site on which the account is being registered. :type site: `django.contrib.sites.models.Site` or `django.contrib.sites.models.RequestSite` :param send_email: 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). :type send_email: bool :rtype: 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

Base view classes

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

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

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

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

Since it's a subclass of `FormView`, `RegistrationView` has all the usual attributes and methods you can override.

When writing your own subclass, one method is required:

`register`(*form*)

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

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

Useful optional 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 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. A string containing a (relative) URL, or a string 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 return a string containing a (relative) URL, or a string 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. Default value is the value of the setting *`REGISTRATION_OPEN`*.

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

One method is required:

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

This method should return the newly-activated user instance (if activation was successful), or boolean `False` if activation was not successful.

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

`success_url`
The URL to redirect to after successful activation. A string containing a (relative) URL, or a string 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 activation. Should be a string. Default value is `registration/activate.html`.

`get_success_url``(`*user*`)`
Return a URL to redirect to after successful registration, on a per-request or per-user basis. If not overridden, will use *`success_url`*. Should return a string containing a (relative) URL, or a string name of a URL pattern, or a 3-tuple of arguments suitable for passing to Django's `redirect` shortcut.

CHAPTER 7

Base form classes

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

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

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

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

`password2` The password to use for the new account. This is represented as a password input (`input type="password"` in the rendered HTML).

Because this is a subclass of Django's own `UserCreationForm`, 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.

Note: Unicode usernames

There is one important difference in form behavior depending on the version of Python you're using. Django's username validation regex allows a username to contain any word character along with the following set of additional characters: `.@+-`. However, on Python 2 this regex uses the `ASCII` flag (since Python 2's string type is ASCII by default), while on Python 3 it uses the `UNICODE` flag (since Python 3's string type is Unicode). This means that usernames containing non-ASCII word characters are only permitted when using Python 3.

The validation error for mismatched passwords is attached to the `password2` field. This is a backwards-incompatible change from django-registration 1.0.

Note: Validation of usernames

Because it's a subclass of Django's `UserCreationForm`, `RegistrationForm` will inherit the base validation defined by Django. It also adds a custom `clean()` method which applies one custom validator: *`ReservedNameValidator`*. See the documentation for `ReservedNameValidator` for notes on why it exists and how to customize its behavior.

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

Custom user models

When django-registration was first developed, Django's authentication system supported only its own built-in user model, `django.contrib.auth.models.User`. More recent versions of Django have introduced support for custom user models.

Older versions of django-registration did not generally support custom user models due to the additional complexity required. However, django-registration now can support custom user models. Depending on how significantly your custom user model differs from Django's default, you may need to change only a few lines of code; custom user models significantly different from the default model may require more work to support.

Overview

The primary issue when using django-registration with a custom user model will be *`RegistrationForm`*. `RegistrationForm` is a subclass of Django's built-in `UserCreationForm`, which in turn is a `ModelForm` with its model set to `django.contrib.auth.models.User`. The only changes made by django-registration are to apply the reserved name validator (*`registration.validators.ReservedNameValidator`*) and make the `email` field required (by default, Django's user model makes this field optional; it is required in `RegistrationForm` because two of the three built-in workflows of django-registration require an email address in order to send account-activation instructions to the user). As a result, you will always be required to supply a custom form class when using django-registration with a custom user model.

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

```
from registration.forms import RegistrationForm

from mycustomuserapp.models import MyCustomUser


class MyCustomUserForm(RegistrationForm):
```

```
    class Meta:
        model = MyCustomUser
```

You will also need to specify the fields to include in the form, via the `fields` declaration.

And then in your URL configuration (example here uses the HMAC activation workflow):

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

from registration.backends.hmac.views import RegistrationView

from mycustomuserapp.forms import MyCustomUserForm


urlpatterns = [
    # ... other URL patterns here
    url(r'^accounts/register/$',
        RegistrationView.as_view(
            form_class=MyCustomUserForm
        ),
        name='registration_register',
    ),
    url(r'^accounts/', include('registration.backends.hmac.urls')),
]
```

If your custom user model is not compatible with the built-in workflows of django-registration (see next section), you will probably need to subclass the provided views (either the base registration views, or the views of the workflow you want to use) and make the appropriate changes for your user model.

Determining compatibility of a custom user model

The built-in workflows and other code of django-registration do as much as is possible to ensure compatibility with custom user models; `django.contrib.auth.models.User` is never directly imported or referred to, and all code in django-registration instead uses `settings.AUTH_USER_MODEL` or `django.contrib.auth.get_user_model()` to refer to the user model, and `USERNAME_FIELD` when access to the username is required.

However, there are still some specific requirements you'll want to be aware of.

The two-step activation workflows – both *HMAC*- and *model*-based – require that your user model have the following fields:

- `email` – a textual field (`EmailField`, `CharField` or `TextField`) holding the user's email address. Note that this field is required by `RegistrationForm`, which is a difference from Django's default `UserCreationForm`.
- `is_active` – a `BooleanField` indicating whether the user's account is active.

You also *must* specify the attribute `USERNAME_FIELD` on your user model to denote the field used as the username. Additionally, your user model must implement the `email_user` method for sending email to the user.

The model-based activation workflow requires one additional field:

- `date_joined` – a `DateField` or `DateTimeField` indicating when the user's account was registered.

The one-step workflow requires that your user model set `USERNAME_FIELD`, and requires that it define a field named `password` for storing the user's password (it will expect to find this value in the `password1` field of the registration form); the combination of `USERNAME_FIELD` and `password` must be sufficient to log a user in. Also

note that `RegistrationForm` requires the `email` field, so either provide that field on your model or subclass `RegistrationForm`.

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

CHAPTER 9

Validation utilities

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

The available error messages are:

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

`registration.validators.`**FREE_EMAIL**
: Error message raised by *RegistrationFormNoFreeEmail* when the supplied email address is rejected by its list of free-email domains.

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

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

All of these error messages are marked for translation; most have translations into multiple languages already in django-registration.

Additionally, two custom validators are provided:

class `registration.validators.`**ReservedNameValidator**
: A custom validator (see Django's validators documentation) which prohibits the use of a reserved name as the value.

  By default, this validator is applied to the username field of *registration.forms.RegistrationForm* and all of its subclasses. The validator is applied in a form-level `clean()` method on `RegistrationForm`, so to remove it (not recommended), subclass `RegistrationForm` and override `clean()`. For no custom form-level validation, you could implement it as:

```
def clean(self):
    pass
```

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

Note: Why reserved names are reserved

Many Web applications enable per-user URLs (to display account information), and some may also create email addresses or even subdomains, based on a user's username. While this is often useful, it also represents a risk: a user might register a name which conflicts with an important URL, email address or subdomain, and this might give that user control over it.

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

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

Several constants are provided which are used by this validator:

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

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

`registration.validators.`**`CA_ADDRESSES`**
A list of email usernames commonly used by certificate authorities when verifying identity.

`registration.validators.`**`RFC_2142`**
A list of common email usernames specified by RFC 2142.

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

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

`registration.validators.`**`OTHER_SENSITIVE_NAMES`**
Other names, not covered by the above lists, which have the potential to conflict with common URLs or subdomains, such as "blog" and "docs".

`registration.validators.`**`DEFAULT_RESERVED_NAMES`**
A list made of the concatentation of all of the above lists, used as the default set of reserved names for *ReservedNameValidator*.

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

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

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

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

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

registration.validators.**validate_confusables_email**(*value*)

A custom validator which prohibits the use of dangerously-confusable email address.

Django permits broad swaths of Unicode to be used in email addresses; while this is useful for serving a world-wide audience, it also creates the possibility of homograph attacks through the use of characters which are easily visually confused for each other (for example, "pypl" contains a Cyrillic "", visually indistinguishable in many fonts from a Latin "").

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

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

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

CHAPTER 10

Custom settings

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

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

  This setting is required if using one of the built-in two-step workflows:

  - *The two-step HMAC activation workflow*
  - *The model-based activation workflow*

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

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

  Used by:

  - *The two-step HMAC activation workflow*
  - *The one-step workflow*
  - *The model-based activation workflow*

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

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

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

  Used by:

  - *The two-step HMAC activation workflow*

CHAPTER 11

Signals used by django-registration

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

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

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

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

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

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

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

`registration.signals.`**`user_registered`**
: Sent when a new user account is registered. Provides the following arguments:

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

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

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

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

CHAPTER 12

Feature and API deprecation cycle

The following features or APIs of django-registration are deprecated and scheduled to be removed in future releases. Please make a note of this and update your use of django-registration accordingly. When possible, deprecated features will emit a `DeprecationWarning` as an additional warning of pending removal.

`registration.urls`

Will be removed in: django-registration 3.0

This URLconf was provided in the earliest days of django-registration, when *the model-based workflow* was the only one provided. Sites using the model-based workflow should instead `include()` the URLconf `registration.backends.model_activation.urls`.

`registration.backends.default`

Will be removed in: django-registration 3.0

Once django-registration began supporting multiple workflows, the model-based workflow was moved to `registration.backends.default`. Later, it was renamed to `registration.backends.model_activation`, but a module was left in place at `registration.backends.default` for compatibility.

Sites using the model-based workflow should ensure all imports are from `registration.backends.model_activation`.

`registration.auth_urls`

Will be removed in: django-registration 3.0

For convenience, each URLconf provided in django-registration also sets up URLs for the views in `django.contrib.auth` (login, logout, password change, and password reset). These URLs are identical – except for the names assigned to them – to those defined in `django.contrib.auth.urls`.

As of 3.0, `registration.auth_urls` will be removed, and django-registration will encourage users to instead `include()` the URLconf `django.contrib.auth.urls` at an appropriate location in their root URLconf.

Expired-account cleanup

Will be removed in: django-registration 3.0

The model-based workflow includes several pieces of code for deleting "expired" – registered but never activated – accounts. This was originally intended as a convenience, but several contentious bug reports have shown that it is less convenient and more prone to ambiguity than desired. As a result, this code will be removed in 3.0. This entails removing the following:

- The `expired()` method
- The `delete_expired_users()` method
- The `cleanupregistration` management command, which invokes the `delete_expired_users` method.

Sites wishing to clean up expired accounts will need to implement a method for doing this which conforms to their needs and their interpretation of "expired".

CHAPTER 13

Security guide

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

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

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

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

on your codebase.

Recommendation: use the HMAC workflow

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

The HMAC workflow provides a verification step – the user must click a link sent to the email address they used to register – which can serve as an impediment to automated account creation for malicious purposes. And unlike *the model-based workflow*, the HMAC workflow does not need to store any additional server-side data (other than the user account itself – the model workflow uses an additional model to store the activation key).

The HMAC workflow generates an activation key which consists of the new account's username and the timestamp of the signup, verified using Django's cryptographic signing tools which in turn use the HMAC implementation from the Python standard library. Thus, django-registration is not inventing or buliding any new cryptography, but only using existing/vetted implementations in an approved and standard manner.

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

Restrictions on user names: reserved names

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

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

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

Restrictions on user names and email addresses: Unicode

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

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

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

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

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

It is strongly recommended that you leave this validation enabled.

CHAPTER 14

Upgrading from previous versions

Prior to 2.0, the last widely-deployed release of django-registration was 0.8; a 1.0 release was published, and 2.3 is mostly backwards-compatible with it, but 1.0 appears not to have seen wide adoption. As such, this guide covers the process of upgrading from django-registration 0.8, as well as from 1.0.

Backends are now class-based views

In django-registration 0.8, a registration workflow was implemented as a class with specific methods for the various steps of the registration process. In django-registration 2.0 and later, a registration workflow is implemented as one or more class-based views.

In general, the required changes to implement a 0.8 registration workflow in django-registration 2.3 is:

0.8 backend class implementation	2.0+ view subclass implementation
Backend class implementing `register()`	*registration.views.RegistrationView.register()*
Backend class implementing `activate()`	*registration.views.ActivationView.activate()*
Backend class implementing `registration_allowed()`	*registration.views.RegistrationView.registration_allowed()*
Backend class implementing `get_form_class()`	*registration.views.RegistrationView.get_form_class()*
Backend class implementing `post_registration_redirect()`	*registration.views.RegistrationView.get_success_url()*
Backend class implementing `post_activation_redirect()`	*registration.views.ActivationView.get_success_url()*

URLconf changes

If you were using one of the provided workflows in django-registration 0.8 without modification, you will not need to make any changes; both `registration.backends.default.urls` and `registration.backends.`

simple.urls have been updated in django-registration 2.0+ to correctly point to the new class-based views:

0.8 URLconf view reference	2.3 URLconf view reference
registration.views.register	registration.views.RegistrationView.as_view()
registration.views.activate	registration.views.ActivationView.as_view()

However, if you were using the two-step model-activation workflow, you should begin referring to registration.backends.model_activation.urls instead of registration.backends.default.urls or registration.urls, as the latter two are deprecated and support for them will be removed in a future release.

If you were passing custom arguments to the built-in registration views, those arguments should continue to work, so long as your URLconf is updated to refer to the new class-based views. For details of how to pass custom arguments to class-based views, see the Django class-based view documentation.

Template changes

When using *RegistrationForm*, the error from mismatched passwords now is attached to the password2 field rather than being a form-level error. To check for and display this error, you will need to change to accessing it via the password2 field rather than via non_field_errors() or the __all__ key in the errors dictionary.

Changes since 1.0

If you used django-registration 1.0, or a pre-2.0 checkout of the code, you will need to make some minor adjustments.

If you previously used registration.backends.default, you will now see deprecation warnings, as the former "default" workflow is now found in registration.backends.model_activation. Use of registration.backends.default continues to work in django-registration 2.3, but will be removed in the future.

Similarly, references to registration.urls should become references to registration.backends.model_activation.urls, and registration.urls is deprecated and will be removed in a future release.

If you had written custom subclasses of *RegistrationView* or of RegistrationView subclasses in the built-in workflows, the following changes need to be noted:

- The register method now receives the *RegistrationForm* instance used during signup, rather than keyword arguments corresponding to the form's cleaned_data.
- RegistrationForm itself is now a subclass of Django's built-in UserCreationForm, and as such is now a ModelForm subclass. This can cause metaclass conflict errors if you write a class which is a subclass of both RegistrationForm and a non-ModelForm form class; to avoid this, ensure that subclasses of RegistrationForm and/or ModelForm come first in your subclass' method resolution order.
- As noted above, the password-mismatch error message is now attached to the password2 field rather than being a form-level error.

Changes since 2.0

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

This is technically backwards-incompatible, since a set of usernames which previously could be registered now cannot be registered, but was included because the security benefits outweigh the edge cases of the now-disallowed usernames.

If you need to allow users to register with usernames forbidden by this validator, see its documentation for notes on how to customize or disable it.

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

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

CHAPTER 15

Frequently-asked questions

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

General

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

By using django-allauth. No single application can or should provide a universal API for every authentication system ever developed; django-registration sticks to making it easy to implement typical signup workflows using Django's own user model and auth system (with some ability to use custom user models), while apps like `django-allauth` handle integration with third-party authentication services far more effectively.

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 <https://github.com/ubernostrum/django-registration/blob/master/LICENSE>.

What versions of Django and Python are supported?

As of django-registration 2.3, Django 1.8, 1.9, 1.10 and 1.11 are supported, on Python 2.7, 3.3 (Django 1.8 only), 3.4, 3.5 and 3.6 (Django 1.11 only). Although Django 1.8 supported Python 3.2 at initial release, Python 3.2 is now at its end-of-life and django-registration no longer supports it.

I found a bug or want to make an improvement!

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

If you'd like to contribute to django-registration, that's great! Just please remember that pull requests should include tests and documentation for any changes made, and that following PEP 8 is mandatory.

Pull requests without documentation won't be merged, and PEP 8 style violations or test coverage below 100% are both configured to break the build.

How secure is django-registration?

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

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

- django-registration 2.3 only supports versions of Django which were receiving upstream security support at the time of release.
- django-registration does not attempt to generate or store passwords, and does not transmit credentials which could be used to log in (the only "credential" ever sent out by django-registration is an activation key used in the two-step activation workflows, and that key can only be used to make an account active; it cannot be used to log in).
- django-registration works with Django's own security features (including cryptographic features) where possible, rather than reinventing its own.

For more details, see *The security guide*.

How do I run the tests?

django-registration makes use of Django's own built-in unit-testing tools, and supports several ways to execute its test suite:

- Within a Django project, invoke `manage.py test registration`.
- If you've installed django-registration (so that it's on your Python import path) and Django, but don't yet have a project created or want to test independently of a project, you can run `registration/runtests.py`, or you can invoke `python setup.py test` (which will run `registration/runtests.py`).

Additionally, the `setup.cfg` file included in django-registration provides configuration for coverage.py, enabling easy recording and reporting of test coverage.

Installation and setup

How do I install django-registration?

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

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

No, for two reasons:

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

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

Configuration

Should I used the model-based or HMAC activation workflow?

You're free to choose whichever one you think best fits your needs. However, *the model-based workflow* is mostly provided for backwards compatibility with older versions of django-registration; it dates to 2007, and though it is still as functional as ever, *the HMAC workflow* has less overhead (i.e., no need to install or work with any models) due to being able to take advantage of more modern features in Django.

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 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.

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

See *the custom user documentation*.

Tips and tricks

How do I close user signups?

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

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

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

How do I re-send an activation email?

Assuming you're using *the model-based workflow*, 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 re-send the email for, then select the "Re-send activation emails" action.

How do I manually activate a user?

In *the model-based workflow*, 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.

In the HMAC-based workflow, toggle the `is_active` field of the user in the admin.

How do I allow Unicode in usernames?

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

See also:

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

Python Module Index

d

r

Index

P

R

S

T

U

V