django-inspectional-registration Documentation

Release 0.6.2

Alisue

November 14, 2016

Contents

Author Alisue <lambdalisue@hashnote.net>

Supported python versions 2.6, 2.7, 3.2, 3.3, 3.4

Supported django versions 1.3 - 1.7

django-inspectional-registration is a enhanced application of django-registration. The following features are available

- Inspection steps for registration. You can accept or reject the account registration before sending activation key to the user.
- Password will be filled in after the activation step to prevent that the user forget them previously filled password in registration step (No password filling in registration step)
- Password can be generated programatically and force to activate the user. The generated password will be sent to the user by e-mail.
- Any Django models are available to use as supplemental information of registration if the models are subclasses of `registration.supplements.RegistrationSupplementBase`. It is commonly used for inspection.
- You can send any additional messages to the user in each steps (acceptance, rejection and activation)
- The behaviors of the application are customizable with Backend feature.
- The E-mails or HTMLs are customizable with Django template system.
- Can be migrate from django-registration simply by south
- django-mailer compatible. Emails sent from the application will use django-mailer if 'mailer' is in your `INSTALLED_APPS`

CHAPTER 1

Documentations

1.1 Quick Tutorial

1.1.1 1. Install django-inspectional-registration

django-inspectional-registration is found on PyPI so execute the following command:

```
$ pip install django-inspectional-registration

or

$ easy_install django-inspectional-registration
```

And also the application is developped on github so you can install it from the repository as:

```
$ pip install git+https://github.com/lambdalisue/django-inspectional-registration.git#egg=django-insp
```

1.1.2 2. Configure the application

To configure django-inspectional-registration, follow the instructions below

1. Add `'registration','django.contrib.admin'` to your `INSTALLED_APPS` of `settings.py`

   Note: If you already use django-registration, see Quick Migrations for migration.

2. Add `'registration.supplements.default'` to your `INSTALLED_APPS` of `settings.py` or set `REGISTRATION_SUPPLEMENT_CLASS` to `None`

   Note: django-inspectional-registration can handle registration supplemental information. If you want to use your own custom registration supplemental information, check About Registration Supplement for documents.

   Settings `REGISTRATION_SUPPLEMENT_CLASS` to `None` mean no registration supplemental information will be used.

3. Add `url('^registration/', include('registration.urls')),` to your very top of (same directory as `settings.py` in default) `urls.py` like below:

```
    from django.conf.urls.defaults import patterns, include, urls

    from django.contrib import admin
    admin.autodiscover()

    urlpatterns = pattern('',
        # some urls...

        # django-inspectional-registration require Django Admin page
        # to inspect registrations
        url('^admin/', include(admin.site.urls)),

        # Add django-inspectional-registration urls. The urls also define
        # Login, Logout and password_change or lot more for handle
        # registration.
        url('^registration/', include('registration.urls')),
    )
```

4. Call `syncdb` command to create the database tables like below:

```
    $ ./manage.py syncdb
```

5. Confirm that Django E-mail settings were properly configured. See https://docs.djangoproject.com/en/dev/topics/email/ for more detail.

Note: If you don't want or too lazy to configure the settings. See django-mailer which store the email on database before sending.

To use django-mailer insted of django's default email system in this application. Simply add 'mailer' to your `INSTALLED_APPS` then the application will use django-mailer insted.

1.1.3 How to use it

1. Access http://localhost:8000/registration/register then you will see the registration page. So fill up (use your own real email address) the fields and click `Register` button.

Note: Did you start your development server? If not:

```
    $ ./manage.py runserver 8000
```

---

2. Now go on the http://localhost:8000/admin/registration/registrationprofile/1/ and accept your registration by clicking `Save` button.

Note: To reject or force to activate the registration, change `Action` and click `Save`

`Message` will be passed to each email template thus you can use the value of `Message` as `{{ message }}` in your email template. In default, the `Message` is only available in rejection email template to explain why the registration was rejected.

---

3. You may get an Email from your website. The email contains an activation key so click the url.

Note: If you get `http://example.com/register/activate/XXXXXXXX` for your activation key,

that mean you haven't configure the site domain name in Django Admin. To prevent this, just set domain name of your site in Admin page.

4. Two password form will be displayed on the activation page, fill up the password and click `Activate` to activate your account.

1.2 Quick Migrations

1.2.1 Instructions

django-inspectional-registration can be migrate from django-registration by south. To migrate, follow the instructions

1. Confirm your application has `'south'`, `'django.contrib.admin'` and in your `INSTALLED_APPS`, if you haven't, add these and run `syncdb` command to create the database table required.
2. Execute following commands:

```
$ ./manage.py migrate registration 0001 --fake
$ ./manage.py migrate registration
```

3. Rewrite your most top of `urls.py` as:

```
from django.conf.urls.defaults import patterns, include, urls

from django.contrib import admin
admin.autodiscover()

urlpatterns = pattern('',
    # some urls...

    # django-inspectional-registration require Django Admin page
    # to inspect registrations
    url('^admin/', include(admin.site.urls)),

    # Add django-inspectional-registration urls. The urls also define
    # Login, Logout and password_change or lot more for handle
    # registration.
    url('^registration/', include('registration.urls')),
)
```

4. Set `REGISTRATION_SUPPLEMENT_CLASS` to `None` in your `settings.py`

Note: django-inspectional-registration can handle registration supplemental information. If you want to use your own custom registration supplemental information, check About Registration Supplement for documents.

Settings `REGISTRATION_SUPPLEMENT_CLASS` to `None` mean no registration supplemental information will be used.

5. Done. Enjoy!

1.2.2 The database difference between django-registration and django-inspectional-registration

django-inspectional-registration add new `CharField` named `registration.models.RegistrationProfile._status` to the `registration.models.RegistrationProfile` and change the storategy to delete `RegistrationProfile` which has been activated from the database insted of setting `'ALREADY_ACTIVATED'` to `registration.models.RegistrationProfile.activation_key`.

`activation_key` will be generated when the `_status` of `RegistrationProfile` be `'accepted'` otherwise it is set `None`

1.3 About Registration Supplement

Registration Supplement is a django model class which inherit `registration.supplements.RegistrationSupplementBa`
It is used to add supplemental information to each registration. Filling the supplemental information is required in registration step and the filled supplemental information will be displayed in Django Admin page.

To disable this supplement feature, set `REGISTRATION_SUPPLEMENT_CLASS` to `None` in your `settings.py`.

1.3.1 Quick tutorial to create your own Registration Supplement

1. Create new app named `supplementtut` with the command below:

```
$ ./manage.py startapp supplementtut
```

2. Create new registration supplement model in your `models.py` as:

```
from __future__ import unicode_literals
from django.db import models
from django.utils.encoding import python_2_unicode_compatible
from registration.supplements.base import RegistrationSupplementBase

class MyRegistrationSupplement(RegistrationSupplementBase):

    realname = models.CharField("Real name", max_length=100, help_text="Please fill your real na
    age = models.IntegerField("Age")
    remarks = models.TextField("Remarks", blank=True)

    def __str__(self):
        # a summary of this supplement
        return "%s (%s)" % (self.realname, self.age)
```

3. Add `supplementtut` to `INSTALLED_APPS` and set `REGISTRATION_SUPPLEMENT_CLASS` to `"supplementtut.models.MyRegistrationSupplement` in your `settings.py`
4. Done, execute `syncdb` and `runserver` to confirm your registration supplement is used correctly. See more documentation in `registration.supplements.RegistrationSupplementBase`

1.4 About Registration Backend

Registration is handled by Registration Backend. See *registration.backends.RegistrationBackendBase* and *registration.backends.default.DefaultRegistrationBackend* for more detail.

1.5 About Registration Templates

django-inspectional-registration use the following templates

1.5.1 Email templates

Used to create the email

acceptance Email

Sent when inspector accpet the account registration

registration/acceptance_email.txt Used to create acceptance email. The following context will be passed

site An instance of `django.contrib.site.Site` to determine the site name and domain name

user A user instance

profile An instance of `registration.models.RegistrationProfile`

activation_key An activation key used to generate activation url. To generate activation url, use the following template command:

```
http://{{ site.domain }}{% url 'registration_activate' activation_key=activation_key %}
```

expiration_days A number of days remaining during which the account may be activated.

message A message from inspector. Not used in default template.

registration/acceptance_email_subject.txt Used to create acceptance email subject. The following context will be passed

site An instance of `django.contrib.site.Site` to determine the site name and domain name

user A user instance

profile An instance of `registration.models.RegistrationProfile`

activation_key An activation key used to generate activation url. To generate activation url, use the following template command:

```
http://{{ site.domain }}{% url 'registration_activate' activation_key=activation_key %}
```

expiration_days A number of days remaining during which the account may be activated.

message A message from inspector. Not used in default template.

Note: All newline will be removed in this template because it is a subject.

Activation Email

Sent when the activation has complete.

registration/activation_email.txt Used to create activation email. The following context will be passed

site An instance of `django.contrib.site.Site` to determine the site name and domain name

user A user instance

password A password of the account. Use this for telling the password when the password is generated automatically.

is_generated If `True`, the password was generated programatically thus you have to tell the password to the user.

message A message from inspector. Not used in default template.

registration/activation_email_subject.txt Used to create activation email subject. The following context will be passed

site An instance of `django.contrib.site.Site` to determine the site name and domain name

user A user instance

password A password of the account. Use this for telling the password when the password is generated automatically.

is_generated If `True`, the password was generated programatically thus you have to tell the password to the user.

message A message from inspector. Not used in default template.

Note: All newline will be removed in this template because it is a subject.

Registration Email

Sent when the registration has complete.

registration/registration_email.txt Used to create registration email. The following context will be passed

site An instance of `django.contrib.site.Site` to determine the site name and domain name

user A user instance

profile An instance of `registration.models.RegistrationProfile`

registration/registration_email_subject.txt Used to create registration email subject. The following context will be passed

site An instance of `django.contrib.site.Site` to determine the site name and domain name

user A user instance

profile An instance of `registration.models.RegistrationProfile`

Note: All newline will be removed in this template because it is a subject.

Rejection Email

Sent when inspector reject the account registration

registration/rejection_email.txt Used to create rejection email. The following context will be passed

site An instance of `django.contrib.site.Site` to determine the site name and domain name

user A user instance

profile An instance of `registration.models.RegistrationProfile`

message A message from inspector. Used for explain why the account registration was rejected in default template

registration/rejection_email_subject.txt Used to create rejection email subject. The following context will be passed

site An instance of `django.contrib.site.Site` to determine the site name and domain name

user A user instance

profile An instance of `registration.models.RegistrationProfile`

message A message from inspector. Used for explain why the account registration was rejected in default template

---

Note: All newline will be removed in this template because it is a subject.

1.5.2 HTML Templates

The following template will be used

registration/activation_complete.html Used for activation complete page.

registration/activation_form Used for activation page. `form` context will be passed to generate the activation form.

registration/login.html Used for login page. `form` context will be passed to generate the login form.

registration/logout.html Used for logged out page.

registration/registration_closed.html Used for registration closed page.

registration/registration_complete.html Used for registration complete page. `registration_profile` context will be passed.

registration/registration_form.html Used for registration page. `form` context will be passed to generate registration form and `supplement_form` context will be passed to generate registration supplement form when the registration supplement exists. Use the following code in your template:

```
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    {{ supplement_form.as_p }}
    <p><input type="submit" value="Register"></p>
</form>
```

1.6 About Registration Signals

django-inspectional-registration provide the following signals.

user_registered(user, profile, request) It is called when a user has registered. The arguments are:

user An instance of User model

profile An instance of RegistrationProfile model of the `user`

request An instance of django's HttpRequest. It is useful to automatically get extra user informations

user_accepted(user, profile, request) It is called when a user has accepted by inspectors. The arguments are:

user An instance of User model

profile An instance of RegistrationProfile model of the `user`

request An instance of django's HttpRequest. It is useful to automatically get extra user informations

user_rejected(user, profile, request) It is called when a user has rejected by inspectors. The arguments are:

user An instance of User model

profile An instance of RegistrationProfile model of the `user`

request An instance of django's HttpRequest. It is useful to automatically get extra user informations

user_activated(user, profile, is_generated, request) It is called when a user has activated by 1) the user access the activation url, 2) inspectors forcely activate the user. The arguments are:

user An instance of User model

password If the user have forcely activated by inspectors, this indicate the raw password, otherwise it is `None` (So that non automatically generated user password is protected from the suffering).

is_generated When inspectors forcely activate the user, it become `True`. It mean that the user do not know own account password thus you need to tell the password to the user somehow (default activation e-mail automatically include the user password if this `is_generated` is `True`)

request An instance of django's HttpRequest. It is useful to automatically get extra user informations

1.7 About Registration Settings

ACCOUNT_ACTIVATION_DAYS The number of days to determine the remaining during which the account may be activated.

Default: `7`

REGISTRATION_DEFAULT_PASSWORD_LENGTH The integer length of the default password programatically generate.

Default: `10`

REGISTRATION_BACKEND_CLASS A string dotted python path for registration backend class.

Default: `'registration.backends.default.DefaultRegistrationBackend'`

REGISTRATION_SUPPLEMENT_CLASS A string dotted python path for registration supplement class.

Default: `'registration.supplements.default.DefaultRegistrationSupplement'`

REGISTRATION_ADMIN_INLINE_BASE_CLASS A string dotted python path for registration supplement admin inline base class.

Default: `'registration.admin.RegistrationSupplementAdminInlineBase'`

REGISTRATION_OPEN A boolean value whether the registration is currently allowed.

Default: `True`

REGISTRATION_REGISTRATION_EMAIL Set `False` to disable sending registration email to the user.

Default: `True`

REGISTRATION_ACCEPTANCE_EMAIL Set `False` to disable sending acceptance email to the user.

Default: `True`

REGISTRATION_REJECTION_EMAIL Set `False` to disable sending rejection email to the user.

Default: `True`

REGISTRATION_ACTIVATION_EMAIL Set `False` to disable sending activation email to the user.

Default: `True`

REGISTRATION_DJANGO_AUTH_URLS_ENABLE (from Version 0.4.0) If it is `False`, django-inspectional-registration do not define the views of django.contrib.auth. It is required to define these view manually.

Default: `True`

REGISTRATION_DJANGO_AUTH_URL_NAMES_PREFIX (from Version 0.4.0) It is used as a prefix string of view names of django.contrib.auth. For backward compatibility, set this value to `'auth_'`.

Default: `''`

REGISTRATION_DJANGO_AUTH_URL_NAMES_SUFFIX (from Version 0.4.0) It is used as a suffix string of view names of django.contrib.auth. For backward compatibility, set this value to `''`.

Default: `''`

1.8 About Registration Contributions

1.8.1 How to use contribution

Registration contributions are simple django app thus you just need to add the path of the contribution to `INSTALLED_APPS`. See the documentation of each contribution for more detail.

autologin
notification

1.9 FAQ

1.9.1 Help! Email have not been sent to the user!

To enable sending email in django, you must have the following settings in your `settings.py`:

```
# if your smtp host use TLS
#EMAIL_USE_TLS = True
# url of your smtp host
EMAIL_HOST = ''
# if your smtp host requre username
#EMAIL_HOST_USER = ''
# if your smtp host require password
#EMAIL_HOST_PASSWORD = ''
# port number which your smtp host used (default 25)
# EMAIL_PORT = 587
DEFAULT_FROM_EMAIL = 'webmaster@your.domain'
```

If you don't have SMTP host but you have Gmail, use the following settings to use your Gmail for SMTP host:

```
EMAIL_USE_TLS = True
EMAIL_PORT = 587
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_email_address@gmail.com'
EMAIL_HOST_PASSWORD = 'your gmail password'
DEFAULT_FROM_EMAIL = 'your_email_address@gmail.com'
```

1.9.2 How can I get notification email when new user has registered in the site?

Use *registration.contrib.notification*.

Add 'registration.contrib.notification' to your INSTALLED_APPS and create following template files in your template directory.

- registration/notification_email.txt
- registration/notification_email_subject.txt

1.9.3 I want to use django-inspectional-registration but I don't need inspection step

If you don't need inspection step, use original django-registration in that case.

However, sometime you do may want to use django-inspectional-registration but inspection. Then follow the instructions below

1. Disable sending registration email with setting REGISTRATION_REGISTRATION_EMAIL to False
2. Add special signal reciever which automatically accept the user registration:

```
    from registration.backends import get_backend
    from registration.signals import user_registered

    def automatically_accept_registration_reciver(sender, user, profile, request, **kwargs):
        backend = get_backend()
        backend.accept(profile, request=request)
    user_registered.connect(automatically_accept_registration_reciver)
```

Then the application behaviors like django-registration

1.9.4 How can I contribute to django-inspectional-registration

Any contributions include adding translations are welcome! Use github's pull request for contribution.

1.10 src

1.10.1 registration package

Subpackages

registration.admin package

Submodules

registration.admin.forms module

class registration.admin.forms.**RegistrationAdminForm**(**args*, ***kwargs*)

Bases: django.forms.models.ModelForm

A special form for handling RegistrationProfile

This form handle RegistrationProfile correctly in save() method. Because RegistrationProfile is not assumed to handle by hands, instance modification by hands is not allowed. Thus subclasses should feel free to add any additions they need, but should avoid overriding a save() method.

ACCEPTED_ACTIONS = ((u'accept', <django.utils.functional.__proxy__ object at 0x7f2100d686d0>), (u'activate', <django

class Meta

exclude = (u'user', u'_status')

model
alias of RegistrationProfile

RegistrationAdminForm.**REJECTED_ACTIONS = ((u'accept', <django.utils.functional.__proxy__ object at 0x7f2100**

RegistrationAdminForm.**UNTREATED_ACTIONS = ((u'accept', <django.utils.functional.__proxy__ object at 0x7f210**

RegistrationAdminForm.**base_fields = OrderedDict([('action_name', <django.forms.fields.ChoiceField object at**

RegistrationAdminForm.**clean_action**()
clean action value

Insted of raising AttributeError, validate the current registration profile status and the requested action and then raise ValidationError

RegistrationAdminForm.**declared_fields = OrderedDict([('action_name', <django.forms.fields.ChoiceField ob**

RegistrationAdminForm.**media**

RegistrationAdminForm.**registration_backend = <registration.backends.default.DefaultRegistrationBackend**

RegistrationAdminForm.**save**(*commit=True*)
Call appropriate action via current registration backend

Insted of modifing the registration profile, this method call current registration backend's accept/reject/activate method as requested.

RegistrationAdminForm.**save_m2m**(*x*)

Module contents

class registration.admin.**RegistrationSupplementAdminInlineBase**(*parent_model*, *admin_site*)

Bases: django.contrib.admin.options.StackedInline

Registration supplement admin inline base class

This inline class is used to generate admin inline class of current registration supplement. Used inline class is defined as settings.REGISTRATION_SUPPLEMENT_ADMIN_INLINE_BASE_CLASS thus if you want to modify the inline class of supplement, create a subclass of this class and set to REGISTRATION_SUPPLEMENT_ADMIN_INLINE_BASE_CLASS

fields = ()

get_readonly_fields(*request*, *obj=None*)
get readonly fields of supplement

Readonly fields will be generated by supplement's `get_admin_fields` and `get_admin_excludes` method thus if you want to change the fields displayed in django admin site. You want to change the method or attributes `admin_fields` or `admin_excludes` which is loaded by those method in default.

See more detail in `registration.supplements.DefaultRegistrationSupplement` documentation.

has_change_permission(*request*, *obj=None*)

media

class `registration.admin.`**RegistrationAdmin**(*model*, *admin_site*)

Bases: `django.contrib.admin.options.ModelAdmin`

Admin class of RegistrationProfile

Admin users can accept/reject registration and activate user in Django Admin page.

If `REGISTRATION_SUPPLEMENT_CLASS` is specified, admin users can see the summary of the supplemental information in list view and detail of it in change view.

`RegistrationProfile` is not assumed to handle by hand thus adding/changing/deleting is not accepted even in Admin page. `RegistrationProfile` only can be accepted/rejected or activated. To prevent these disallowed functions, the special AdminForm called `RegistrationAdminForm` is used. Its `save` method is overridden and it actually does not save the instance. It just call `accept`, `reject` or `activate` method of current registration backend. So you don't want to override the `save` method of the form.

accept_users(*request*, *queryset*)

Accept the selected users, if they are not already accepted

actions = (u'accept_users', u'reject_users', u'force_activate_users', u'resend_acceptance_email')

backend = <registration.backends.default.DefaultRegistrationBackend object>

change_view(**args*, ***kwargs*)

called for change view

Check permissions of the admin user for `POST` request depends on what action is requested and raise PermissionDenied if the action is not accepted for the admin user.

display_activation_key(*obj*)

Display activation key with link

Note that displaying activation key is not recommended in security reason. If you really want to use this method, create your own subclass and re-register to admin.site

Even this is a little bit risky, it is really useful for developping (without checking email, you can activate any user you want) thus I created but turned off in default :-p

display_supplement_summary(*obj*)

Display supplement summary

Display `__unicode__` method result of `REGISTRATION_SUPPLEMENT_CLASS` `Not available` when `REGISTRATION_SUPPLEMENT_CLASS` is not specified

force_activate_users(*request*, *queryset*)

Activates the selected users, if they are not already activated

form

alias of `RegistrationAdminForm`

get_actions(*request*)

get actions displaied in admin site

RegistrationProfile should not be deleted in admin site thus 'delete_selected' is disabled in default.

Each actions has permissions thus delete the action if the accessed user doesn't have appropriate permission.

get_inline_instances(*request*, *obj=None*)
return inline instances with registration supplement inline instance

get_object(*request*, *object_id*, *from_field=None*)
add `request` instance to model instance and return

To get `request` instance in form, `request` instance is stored in the model instance.

has_accept_permission(*request*, *obj*)
whether the user has accept permission

has_activate_permission(*request*, *obj*)
whether the user has activate permission

has_add_permission(*request*)
registration profile should not be created by hand

has_delete_permission(*request*, *obj=None*)
registration profile should not be created by hand

has_reject_permission(*request*, *obj*)
whether the user has reject permission

list_display = (u'user', u'get_status_display', u'activation_key_expired', u'display_supplement_summary')

list_filter = (u'_status',)

media

raw_id_fields = [u'user']

readonly_fields = (u'user', u'_status')

reject_users(*request*, *queryset*)
Reject the selected users, if they are not already accepted

resend_acceptance_email(*request*, *queryset*)
Re-sends acceptance emails for the selected users

Note that this will *only* send acceptance emails for users who are eligible to activate; emails will not be sent to users whose activation keys have expired or who have already activated or rejected.

search_fields = (u'user__username', u'user__first_name', u'user__last_name')

registration.backends package

Subpackages

registration.backends.default package

Module contents

class `registration.backends.default.`**DefaultRegistrationBackend**
Bases: *registration.backends.base.RegistrationBackendBase*

Default registration backend class

A registration backend which floows a simple workflow:

1. User sigs up, inactive account with unusable password is created.

2.Inspector accept or reject the account registration.

3.Email is sent to user with/without activation link (without when rejected)

4.User clicks activation link, enter password, account is now active

Using this backend requires that

- `registration` be listed in the `INSTALLED_APPS` settings (since this backend makes use of models defined in this application).
- `django.contrib.admin` be listed in the `INSTALLED_APPS` settings
- The setting `ACCOUNT_ACTIVATION_DAYS` be supplied, specifying (as an integer) the number of days from acceptance during which a user may activate their account (after that period expires, activation will be disallowed). Default is `7`
- The creation of the templates
  - `registration/registration_email.txt`
  - `registration/registration_email_subject.txt`
  - `registration/acceptance_email.txt`
  - `registration/acceptance_email_subject.txt`
  - `registration/rejection_email.txt`
  - `registration/rejection_email_subject.txt`
  - `registration/activation_email.txt`
  - `registration/activation_email_subject.txt`

Additinally, registration can be temporarily closed by adding the setting `REGISTRATION_OPEN` and setting it to `False`. Omitting this setting, or setting it to `True`, will be imterpreted as meaning that registration is currently open and permitted.

Internally, this is accomplished via storing an activation key in an instance of `registration.models.RegistrationProfile`. See that model and its custom manager for full documentation of its fields and supported operations.

accept (*profile*, *request*, *send_email=None*, *message=None*, *force=False*)
accept the account registration of `profile`

Given a profile, accept account registration, which will set inspection status of `profile` to `accepted` and generate new activation key of `profile`.

An email will be sent to the supplied email address; The email will be rendered using two templates. See the documentation for `RegistrationProfile.send_acceptance_email()` for information about these templates and the contexts provided to them.

If `REGISTRATION_acceptance_EMAIL` of settings is `None`, no acceptance email will be sent.

After successful acceptance, the signal `registration.signals.user_accepted` will be sent, with the newly accepted `User` as the keyword argument `uesr`, the `RegistrationProfile` of the `User` as the keyword argument `profile` and the class of this backend as the sender

activate (*activation_key*, *request*, *password=None*, *send_email=None*, *message=None*, *no_profile_delete=False*)
activate user with `activation_key` and `password`

Given an activation key, password, look up and activate the user account corresponding to that key (if possible) and set its password.

If `password` is not given, password will be generated

An email will be sent to the supplied email address; The email will be rendered using two templates. See the documentation for `RegistrationProfile.send_activation_email()` for information about these templates and the contexts provided to them.

If `REGISTRATION_ACTIVATION_EMAIL` of settings is `None`, no activation email will be sent.

After successful activation, the signal `registration.signals.user_activated` will be sent, with the newly activated `User` as the keyword argument `uesr`, the password of the `User` as the keyword argument `password`, whether the password has generated or not as the keyword argument `is_generated` and the class of this backend as the sender

get_activation_complete_url(*user*)
Return a url to redirect to after successful user activation

get_activation_form_class()
Return the default form class used for user activation

get_registration_closed_url()
Return a url to redirect to if registration is closed

get_registration_complete_url(*user*)
Return a url to redirect to after successful user registration

get_registration_form_class()
Return the default form class used for user registration

get_supplement_class()
Return the current registration supplement class

get_supplement_form_class()
Return the default form class used for user registration supplement

register(*username*, *email*, *request*, *supplement=None*, *send_email=None*)
register new user with `username` and `email`

Given a username, email address, register a new user account, which will initially be inactive and has unusable password.

Along with the new `User` object, a new `registration.models.RegistrationProfile` will be created, tied to that `User`, containing the inspection status and activation key which will be used for this account (activation key is not generated untill its inspection status is set to `accepted`)

An email will be sent to the supplied email address; The email will be rendered using two templates. See the documentation for `RegistrationProfile.send_registration_email()` for information about these templates and the contexts provided to them.

If `REGISTRATION_REGISTRATION_EMAIL` of settings is `None`, no registration email will be sent.

After the `User` and `RegistrationProfile` are created and the registration email is sent, the signal `registration.signals.user_registered` will be sent, with the new `User` as the keyword argument `user`, the `RegistrationProfile` of the new `User` as the keyword argument `profile` and the class of this backend as the sender.

registration_allowed()
Indicate whether account registration is currently permitted, based on the value of the setting `REGISTRATION_OEPN`. This is determined as follows:

- If `REGISTRATION_OPEN` is not specified in settings, or is set to `True`, registration is permitted.
- If `REGISTRATION_OPEN` is both specified and set to `False`, registration is not permitted.

reject(*profile*, *request*, *send_email=None*, *message=None*)
reject the account registration of `profile`

Given a profile, reject account registration, which will set inspection status of `profile` to `rejected` and delete activation key of `profile` if exists.

An email will be sent to the supplied email address; The email will be rendered using two templates. See the documentation for `RegistrationProfile.send_rejection_email()` for information about these templates and the contexts provided to them.

If `REGISTRATION_REJECTION_EMAIL` of settings is `None`, no rejection email will be sent.

After successful rejection, the signal `registration.signals.user_rejected` will be sent, with the newly rejected `User` as the keyword argument `uesr`, the `RegistrationProfile` of the `User` as the keyword argument `profile` and the class of this backend as the sender

Submodules

registration.backends.base module

class `registration.backends.base.`**RegistrationBackendBase**

Bases: `object`

Base class of registration backend

```
get_site                    -- return current site
register                    -- register a new user
accept                      -- accept a registration
reject                      -- reject a registration
activate                    -- activate a user
get_supplement_class        -- get registration supplement class
get_activation_form_class   -- get activation form class
get_registration_form_class -- get registration form class
get_supplement_form_class   -- get registration supplement form class
get_activation_complete_url -- get activation complete redirect url
get_registration_complete_url -- get registration complete redirect url
get_registration_closed_url -- get registration closed redirect url
registration_allowed        -- whether registration is allowed now
```

accept(*profile*, *request*, *send_email=True*, *message=None*, *force=False*)

accept account registration with given `profile` (an instance of `RegistrationProfile`)

Returning should be a instance of accepted `User` for success, `None` for fail.

This method **SHOULD** work even after the account registration has rejected.

activate(*activation_key*, *request*, *password=None*, *send_email=True*, *message=None*, *no_profile_delete=False*)

activate account with `activation_key` and `password`

This method should be called after the account registration has accepted, otherwise it should not be success.

Returning is `user`, `password` and `is_generated` for success, `None` for fail.

If `password` is not given, this method will generate password and `is_generated` should be `True` in this case.

get_activation_complete_url(*user*)
get activation complete url

get_activation_form_class()
get activation form class

get_registration_closed_url()
get registration closed url

get_registration_complete_url(*user*)
get registration complete url

get_registration_form_class()
get registration form class

get_site(*request*)
get current `django.contrib.Site` instance

return `django.contrib.RequestSite` instance when the `Site` is not installed.

get_supplement_class()
Return the current registration supplement class

get_supplement_form_class()
get registration supplement form class

register(*username*, *email*, *request*, *supplement=None*, *send_email=True*)
register a new user account with given `username` and `email`

Returning should be a instance of new `User`

registration_allowed()
return `False` if the registration has closed

reject(*profile*, *request*, *send_email=True*, *message=None*)
reject account registration with given `profile` (an instance of `RegistrationProfile`)

Returning should be a instance of accepted `User` for success, `None` for fail.

This method **SHOULD NOT** work after the account registration has accepted.

Module contents

`registration.backends.`**get_backend**(*path=None*)
Return an instance of a registration backend, given the dotted Python import path (as a string) to the backend class.

If the backend cannot be located (e.g., because no such module exists, or because the module does not contain a class of the appropriate name), `django.core.exceptions.ImproperlyConfigured` is raised.

class `registration.backends.`**RegistrationBackendBase**
Bases: `object`

Base class of registration backend

```
get_site                    -- return current site
register                    -- register a new user
accept                      -- accept a registration
reject                      -- reject a registration
activate                    -- activate a user
get_supplement_class        -- get registration supplement class
```

```
get_activation_form_class     -- get activation form class
get_registration_form_class   -- get registration form class
get_supplement_form_class     -- get registration supplement form class
get_activation_complete_url   -- get activation complete redirect url
get_registration_complete_url -- get registration complete redirect url
get_registration_closed_url   -- get registration closed redirect url
registration_allowed          -- whether registration is allowed now
```

accept(*profile*, *request*, *send_email=True*, *message=None*, *force=False*)
accept account registration with given `profile` (an instance of `RegistrationProfile`)

Returning should be a instance of accepted `User` for success, `None` for fail.

This method **SHOULD** work even after the account registration has rejected.

activate(*activation_key*, *request*, *password=None*, *send_email=True*, *message=None*, *no_profile_delete=False*)
activate account with `activation_key` and `password`

This method should be called after the account registration has accepted, otherwise it should not be success.

Returning is `user`, `password` and `is_generated` for success, `None` for fail.

If `password` is not given, this method will generate password and `is_generated` should be `True` in this case.

get_activation_complete_url(*user*)
get activation complete url

get_activation_form_class()
get activation form class

get_registration_closed_url()
get registration closed url

get_registration_complete_url(*user*)
get registration complete url

get_registration_form_class()
get registration form class

get_site(*request*)
get current `django.contrib.Site` instance

return `django.contrib.RequestSite` instance when the `Site` is not installed.

get_supplement_class()
Return the current registration supplement class

get_supplement_form_class()
get registration supplement form class

register(*username*, *email*, *request*, *supplement=None*, *send_email=True*)
register a new user account with given `username` and `email`

Returning should be a instance of new `User`

registration_allowed()
return `False` if the registration has closed

reject(*profile*, *request*, *send_email=True*, *message=None*)
reject account registration with given `profile` (an instance of `RegistrationProfile`)

Returning should be a instance of accepted `User` for success, `None` for fail.

This method **SHOULD NOT** work after the account registration has accepted.

registration.contrib package

Subpackages

registration.contrib.autologin package

Submodules

registration.contrib.autologin.conf module

class `registration.contrib.autologin.conf.`**InspectionalRegistrationAutoLoginAppConf**(***kwargs*)
Bases: `appconf.base.AppConf`

AUTO_LOGIN = True

class Meta

prefix = u'registration'

registration.contrib.autologin.models module

registration.contrib.autologin.tests module

class `registration.contrib.autologin.tests.`**RegistrationAutoLoginTestCase**(*methodName='runTest'*)
Bases: `django.test.testcases.TestCase`

backend = <registration.backends.default.DefaultRegistrationBackend object>

mock_request = <WSGIRequest: GET '/'>

test_auto_login()
Wheather auto login feature works correctly

test_no_auto_login_with_no_password()
Auto login feature should not be occur with no password (programatically activated by Django Admin action)

test_no_auto_login_with_setting()
Auto login feature should be able to off with `REGISTRATION_AUTO_LOGIN = False`

Module contents

`registration.contrib.autologin.`**auto_login_reciver**(*sender*, *user*, *password*, *is_generated*, *request*, ***kwargs*)
automatically log activated user in when they have activated

`registration.contrib.autologin.`**is_auto_login_enable**()
get whether the registration autologin is enable

registration.contrib.notification package

Subpackages

registration.contrib.notification.tests package

Submodules

registration.contrib.notification.tests.urls module

Module contents

class registration.contrib.notification.tests.**RegistrationNotificationTestCase**(*methodName='runT*

Bases: django.test.testcases.TestCase

backend = <registration.backends.default.DefaultRegistrationBackend object>

mock_request = <WSGIRequest: GET '/'>

test_notify_admins()

test_notify_all()

test_notify_duplicated()

test_notify_managers()

test_notify_recipients_function()

test_notify_recipients_iterable()

Submodules

registration.contrib.notification.conf module

class registration.contrib.notification.conf.**InspectionalRegistrationNotificationAppConf**(***kw*

Bases: appconf.base.AppConf

class Meta

prefix = u'registration'

InspectionalRegistrationNotificationAppConf.**NOTIFICATION = True**

InspectionalRegistrationNotificationAppConf.**NOTIFICATION_ADMINS = True**

InspectionalRegistrationNotificationAppConf.**NOTIFICATION_EMAIL_SUBJECT_TEMPLATE_NAME = u**

InspectionalRegistrationNotificationAppConf.**NOTIFICATION_EMAIL_TEMPLATE_NAME = u'registrati**

InspectionalRegistrationNotificationAppConf.**NOTIFICATION_MANAGERS = True**

InspectionalRegistrationNotificationAppConf.**NOTIFICATION_RECIPIENTS = None**

registration.contrib.notification.models module

Module contents

registration.contrib.notification.**is_notification_enable**()
get whether the registration notification is enable

registration.contrib.notification.**send_notification_email_reciver**(*sender*, *user*, *profile*, *request*, ***kwargs*)
send a notification email to admins/managers

Module contents

registration.management package

Subpackages

registration.management.commands package

Submodules

registration.management.commands.cleanup_expired_registrations module

class registration.management.commands.cleanup_expired_registrations.**Command**(*stdout=None*, *stderr=None*, *no_color=False*)
Bases: django.core.management.base.BaseCommand

handle(***options*)

help = u'Delete expired user registrations from the database'

registration.management.commands.cleanup_registrations module

class registration.management.commands.cleanup_registrations.**Command**(*stdout=None*, *stderr=None*, *no_color=False*)
Bases: django.core.management.base.BaseCommand

handle(***options*)

help = u'Delete expired/rejected user registrations from the database'

registration.management.commands.cleanup_rejected_registrations module

class registration.management.commands.cleanup_rejected_registrations.**Command**(*stdout=None*, *stderr=None*, *no_color=False*)
Bases: django.core.management.base.BaseCommand

handle(***options*)

help = u'Delete rejected user registrations from the database'

registration.management.commands.cleanupregistration module

Module contents

Module contents

registration.migrations package

Submodules

registration.migrations.0001_initial module

class registration.migrations.0001_initial.**Migration**(*name*, *app_label*)

Bases: django.db.migrations.migration.Migration

dependencies = [(u'auth', u'__first__')]

operations = [<CreateModel fields=[(u'id', <django.db.models.fields.AutoField>), (u'_status', <django.db.models.fields

Module contents

registration.south_migrations package

Submodules

registration.south_migrations.0001_initial module

registration.south_migrations.0002_auto__add_field_registrationprofile__status__chg_field_registrationpro module

registration.south_migrations.0003_status module

registration.south_migrations.0004_activation_keys module

Module contents

registration.supplements package

Subpackages

registration.supplements.default package

Submodules

registration.supplements.default.models module

class registration.supplements.default.models.**DefaultRegistrationSupplement**(**args*, ***kwargs*)

Bases: *registration.supplements.base.RegistrationSupplementBase*

A simple registration supplement model which requires remarks

exception DoesNotExist

Bases: django.core.exceptions.ObjectDoesNotExist

exception DefaultRegistrationSupplement.**MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

DefaultRegistrationSupplement.**id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

DefaultRegistrationSupplement.**objects = <django.db.models.manager.Manager object>**

DefaultRegistrationSupplement.**registration_profile**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

DefaultRegistrationSupplement.**remarks**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Module contents

Submodules

registration.supplements.base module

class registration.supplements.base.**RegistrationSupplementBase**(**args*, ***kwargs*)

Bases: django.db.models.base.Model

A registration supplement abstract model

Registration supplement model is used to add supplemental information to the account registration. The supplemental information is written by the user who tried to register the site and displaied in django admin page to help determine the acceptance/rejection of the registration

The __str__() method is used to display the summary of the supplemental information in django admin's change list view. Thus subclasses must define them own __str__() method.

The get_form_class() is a class method return a value of form_class attribute to determine the form class used for filling up the supplemental informatin in registration view if form_class is specified. Otherwise the method create django's ModelForm and return.

The get_admin_fields() is a class method return a list of field names displayed in django admin site. It simply return a value of admin_fields attribute in default. If the method return None, then all fields except id (and fields in admin_excludes) will be displayed.

The `get_admin_excludes()` is a class method return a list of field names NOT displayed in django admin site. It simply return a value of `admin_excludes` attribute in default. If the method return `None`, then all fields selected with `admin_fields` except `id` will be displayed.

The `registration_profile` field is used to determine the registration profile associated with. `related_name` of the field is used to get the supplemental information in `_get_supplement()` method of `RegistrationProfile` thus DO NOT CHANGE the name.

class Meta

abstract = False

`RegistrationSupplementBase.`**admin_excludes = None**

`RegistrationSupplementBase.`**admin_fields = None**

`RegistrationSupplementBase.`**form_class = None**

classmethod `RegistrationSupplementBase.`**get_admin_excludes**`()`
Return a list of field names NOT displayed in django admin site

It is simply return a value of `admin_excludes` in default. If it returns `None` then all fields (selected in `admin_fields`) except `id` will be displayed.

classmethod `RegistrationSupplementBase.`**get_admin_fields**`()`
Return a list of field names displayed in django admin site

It is simply return a value of `admin_fields` in default. If it returns `None` then all fields except `id` (and fields in `admin_excludes`) will be displayed.

classmethod `RegistrationSupplementBase.`**get_form_class**`()`
Return the form class used for this registration supplement model

When `form_class` is specified, this method return the value of the attribute. Otherwise it generate django's `ModelForm`, set it to `form_class` and return it

This method MUST BE class method.

`RegistrationSupplementBase.`**registration_profile**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`RegistrationSupplementBase.`**registration_profile_id**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Module contents

`registration.supplements.`**get_supplement_class**(*path=None*)
Return an instance of a registration supplement, given the dotted Python import path (as a string) to the supplement class.

If the addition cannot be located (e.g., because no such module exists, or because the module does not contain a class of the appropriate name), `django.core.exceptions.ImproperlyConfigured` is raised.

registration.tests package

Submodules

registration.tests.compat module

registration.tests.mock module

registration.tests.mock.**mock_request**()

Construct and return a mock `HttpRequest` object; this is used in testing backend methods which expect an `HttpRequest` but which are not being called from views.

registration.tests.mock.**mock_site**()

Construct and return a mock *Site'* object; this is used in testing methods which expect an `Site`

registration.tests.test_admin module

class registration.tests.test_admin.**RegistrationAdminTestCase**(*methodName='runTest'*)

Bases: `django.test.testcases.TestCase`

setUp()

test_accept_users_action()

test_change_list_view_get()

test_change_view_get()

test_change_view_get_404()

test_change_view_post_invalid_activate_from_rejected()

test_change_view_post_invalid_activate_from_untreated()

test_change_view_post_invalid_force_activate_from_accepted()

test_change_view_post_invalid_reject_from_accepted()

test_change_view_post_invalid_reject_from_rejected()

test_change_view_post_valid_accept_from_accepted()

test_change_view_post_valid_accept_from_rejected()

test_change_view_post_valid_accept_from_untreated()

test_change_view_post_valid_activate_from_accepted()

test_change_view_post_valid_force_activate_from_rejected()

test_change_view_post_valid_force_activate_from_untreated()

test_change_view_post_valid_reject_from_untreated()

test_force_activate_users_action()

test_get_inline_instances_with_default_supplements(**args*, ***kwargs*)

test_get_inline_instances_without_supplements(**args*, ***kwargs*)

test_reject_users_action()

test_resend_acceptance_email_action()

registration.tests.test_backends module

class registration.tests.test_backends.**DefaultRegistrationBackendTestCase**(*methodName='runTest'*)

Bases: django.test.testcases.TestCase

setUp()

test_acceptance()

test_acceptance_signal()

test_acceptance_signal_fail()

test_activation_signal()

test_activation_with_password()

test_activation_without_password()

test_allow()

test_expired_activation()

test_get_activation_complete_url()

test_get_activation_form_class()

test_get_registration_closed_url()

test_get_registration_complete_url()

test_get_registration_form_class()

test_registration()

test_registration_signal()

test_registration_signal_with_supplement(**args*, ***kwargs*)

test_rejected_activation()

test_rejection()

test_rejection_signal()

test_rejection_signal_fail()

test_untreated_activation()

class registration.tests.test_backends.**RegistrationBackendRetrievalTests**(*methodName='runTest'*)

Bases: django.test.testcases.TestCase

test_backend_attribute_error()

test_backend_error_invalid()

test_get_backend()

registration.tests.test_forms module

class registration.tests.test_forms.**ActivationFormTests**(*methodName='runTest'*)

Bases: django.test.testcases.TestCase

Test the default registration forms.

test_activation_form()

Test that ActivationForm enforces username constraints and matching passwords.

class registration.tests.test_forms.**RegistrationFormTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Test the default registration forms.

test_registration_form()
Test that RegistrationForm enforces username constraints and matching passwords.

test_registration_form_no_free_email()
Test that RegistrationFormNoFreeEmail disallows registration with free email addresses.

test_registration_form_tos()
Test that RegistrationFormTermsOfService requires agreement to the terms of service.

test_registration_form_unique_email()
Test that RegistrationFormUniqueEmail validates uniqueness of email addresses.

registration.tests.test_models module

class registration.tests.test_models.**RegistrationProfileManagerTestCase**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_acceptance()

test_acceptance_after_acceptance_fail()

test_acceptance_after_rejection_success()

test_acceptance_email()

test_acceptance_force()

test_acceptance_no_email()

test_activation_email()

test_activation_no_email()

test_activation_with_expired_fail()

test_activation_with_invalid_key_fail()

test_activation_with_password()

test_activation_with_rejected_fail()

test_activation_with_untreated_fail()

test_activation_without_password()

test_expired_user_deletion()

test_management_command_cleanup_expired_registrations()

test_management_command_cleanup_registrations()

test_management_command_cleanup_rejected_registrations()

test_management_command_cleanupregistration()

test_register()

test_register_email()

test_register_no_email()

test_rejected_user_deletion()

test_rejection()

test_rejection_after_acceptance_fail()

test_rejection_after_rejection_fail()

test_rejection_email()

test_rejection_no_email()

user_info = {u'username': u'alice', u'email': u'alice@example.com'}

class registration.tests.test_models.**RegistrationProfileTestCase**(*methodName='runTest'*)

Bases: django.test.testcases.TestCase

create_inactive_user()

setUp()

test_profile_creation()

test_profile_status_modification()

test_send_acceptance_email()

test_send_activation_email()

test_send_registration_email()

test_send_rejection_email()

user_info = {u'username': u'alice', u'password': u'password', u'email': u'alice@example.com'}

registration.tests.test_supplements module

class registration.tests.test_supplements.**RegistrationSupplementRetrievalTests**(*methodName='runT*

Bases: django.test.testcases.TestCase

test_get_supplement_class()

test_supplement_attribute_error()

test_supplement_error_invalid()

class registration.tests.test_supplements.**RegistrationViewWithDefaultRegistrationSupplementTe**

Bases: django.test.testcases.TestCase

test_registration_view_get()

A GET to the register view uses the appropriate template and populates the registration form into the context.

test_registration_view_post_failure()

A POST to the register view with invalid data does not create a user, and displays appropriate error messages.

test_registration_view_post_no_remarks_failure()

A POST to the register view with invalid data does not create a user, and displays appropriate error messages.

test_registration_view_post_success()

A POST to the register view with valid data properly creates a new user and issues a redirect.

registration.tests.test_views module

class registration.tests.test_views.**RegistrationViewTestCase**(*methodName='runTest'*)

Bases: django.test.testcases.TestCase

setUp()

test_activation_view_get_fail()
A GET to the ActivationView view wht invalid activation_key raise Http404

test_activation_view_get_success()
A GET to the ActivationView view with valid activation_key

test_activation_view_post_failure()
A POST to the ActivationView view with invalid data does not activate a user, and raise Http404

test_activation_view_post_success()
A POST to the ActivationView view with valid data properly handles a valid activation

test_registration_complete_view_get()
A GET to the complete view uses the appropriate template and populates the registration form into the context.

test_registration_view_closed()
Any attempt to access the register view when registration is closed fails and redirects.

test_registration_view_get()
A GET to the register view uses the appropriate template and populates the registration form into the context.

test_registration_view_post_failure()
A POST to the register view with invalid data does not create a user, and displays appropriate error messages.

test_registration_view_post_success()
A POST to the register view with valid data properly creates a new user and issues a redirect.

registration.tests.utils module

registration.tests.utils.**with_apps**(**apps*)
Class decorator that makes sure the passed apps are present in INSTALLED_APPS.

Module contents

Submodules

registration.compat module

registration.compat.**patterns**(*x*, **args*)

registration.conf module

class registration.conf.**InspectionalRegistrationAppConf**(***kwargs*)
Bases: appconf.base.AppConf

ACCEPTANCE_EMAIL = True

ACTIVATION_EMAIL = True

BACKEND_CLASS = u'registration.backends.default.DefaultRegistrationBackend'

DEFAULT_PASSWORD_LENGTH = 10

DJANGO_AUTH_URLS_ENABLE = True

DJANGO_AUTH_URL_NAMES_PREFIX = u''

DJANGO_AUTH_URL_NAMES_SUFFIX = u''

class Meta

prefix = u'registration'

InspectionalRegistrationAppConf.**OPEN = True**

InspectionalRegistrationAppConf.**REJECTION_EMAIL = True**

InspectionalRegistrationAppConf.**SUPPLEMENT_ADMIN_INLINE_BASE_CLASS = u'registration.admin.Regi**

InspectionalRegistrationAppConf.**SUPPLEMENT_CLASS = u'registration.supplements.default.models.DefaultR**

InspectionalRegistrationAppConf.**USE_OBJECT_PERMISSION = False**

registration.conf.**configure_other_settings**()

registration.forms module

class registration.forms.**ActivationForm**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*)

Bases: django.forms.forms.Form

Form for activating a user account.

Requires the password to be entered twice to catch typos.

Subclasses should feel free to add any additional validation they need, but should avoid defining a save() method – the actual saving of collected user data is delegated to the active registration backend.

base_fields = OrderedDict([('password1', <django.forms.fields.CharField object at 0x7f2100d5db10>), ('password2',

clean()

Check the passed two password are equal

Verifiy that the values entered into the two password fields match. Note that an error here will end up in non_field_errors() because it doesn't apply to a single field.

declared_fields = OrderedDict([('password1', <django.forms.fields.CharField object at 0x7f2100d5db10>), ('passwo

media

class registration.forms.**RegistrationForm**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*)

Bases: django.forms.forms.Form

Form for registration a user account.

Validates that the requested username is not already in use, and requires the email to be entered twice to catch typos.

Subclasses should feel free to add any additional validation they need, but should avoid defining a save() method – the actual saving of collected user data is delegated to the active registration backend.

base_fields = OrderedDict([('username', <django.forms.fields.RegexField object at 0x7f2100d5df10>), ('email1', <dja

clean()
Check the passed two email are equal

Verifiy that the values entered into the two email fields match. Note that an error here will end up in non_field_errors() because it doesn't apply to a single field.

clean_username()
Validate that the username is alphanumeric and is not already in use.

declared_fields = OrderedDict([('username', <django.forms.fields.RegexField object at 0x7f2100d5df10>), ('email1'

media

class registration.forms.**RegistrationFormNoFreeEmail**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*)

Bases: *registration.forms.RegistrationForm*

Subclass of RegistrationForm which disallows registration with email addresses from popular free webmail services; moderately useful for preventing automated spam registration.

To change the list of banned domains, subclass this form and override the attribute bad_domains.

bad_domains = [u'aim.com', u'aol.com', u'email.com', u'gmail.com', u'googlemail.com', u'hotmail.com', u'hushmail.co

base_fields = OrderedDict([('username', <django.forms.fields.RegexField object at 0x7f2100d5df10>), ('email1', <dja

clean_email1()
Check the supplied email address against a list of known free webmail domains.

declared_fields = OrderedDict([('username', <django.forms.fields.RegexField object at 0x7f2100d5df10>), ('email1'

media

class registration.forms.**RegistrationFormTermsOfService**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*)

Bases: *registration.forms.RegistrationForm*

Subclass of RegistrationForm which adds a required checkbox for agreeing to a site's Terms of Service.

base_fields = OrderedDict([('username', <django.forms.fields.RegexField object at 0x7f2100d5df10>), ('email1', <dja

declared_fields = OrderedDict([('username', <django.forms.fields.RegexField object at 0x7f2100d5df10>), ('email1'

media

class registration.forms.**RegistrationFormUniqueEmail**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*)

Bases: *registration.forms.RegistrationForm*

Subclass of RegistrationForm which enforces uniqueness of email address

base_fields = OrderedDict([('username', <django.forms.fields.RegexField object at 0x7f2100d5df10>), ('email1', <dja

clean_email1()

Validate that the supplied email address is unique for the site.

declared_fields = OrderedDict([('username', <django.forms.fields.RegexField object at 0x7f2100d5df10>), ('email1'

media

registration.models module

registration.signals module

registration.urls module

registration.utils module

registration.utils.**generate_activation_key**(*username*)

generate activation key with username

originally written by ubernostrum in django-registration

registration.utils.**generate_random_password**(*length=10*)

generate random password with passed length

registration.utils.**get_site**(*request*)

get current django.contrib.Site instance

return django.contrib.RequestSite instance when the Site is not installed.

registration.utils.**send_mail**(*subject*, *message*, *from_email*, *recipients*)

send mail to recipients

this method use django-mailer send_mail method when the app is in INSTALLED_APPS

> **Note:** django-mailer send_mail is not used duaring unittest because it is a little bit difficult to check the number of mail sent in unittest for both django-mailer and original django send_mail

registration.views module

class registration.views.**ActivationCompleteView**(***kwargs*)

Bases: django.views.generic.base.TemplateView

A simple template view for activation complete

template_name = u'registration/activation_complete.html'

class registration.views.**ActivationView**(**args*, ***kwargs*)
Bases: django.views.generic.base.TemplateResponseMixin, django.views.generic.edit.FormMixin, django.views.generic.detail.SingleObjectMixin, django.views.generic.edit.ProcessFormView

A complex view for activation

GET: Display an ActivationForm which has password1 and password2 for activation user who has activation_key password1 and password2 should be equal to prepend typo

POST: Activate the user who has activation_key with passed password1

form_valid(*form*)
activate user who has activation_key with password1

this method is called when form validation has successed.

get(*request*, **args*, ***kwargs*)

get_form_class()
get activation form class via backend

get_object(*queryset=None*)
get RegistrationProfile instance by activation_key

activation_key should be passed by URL

get_queryset()
get RegistrationProfile queryset which status is 'accepted'

get_success_url()
get activation complete url via backend

model
alias of RegistrationProfile

post(*request*, **args*, ***kwargs*)

template_name = u'registration/activation_form.html'

class registration.views.**RegistrationClosedView**(***kwargs*)
Bases: django.views.generic.base.TemplateView

A simple template view for registraion closed

This view is called when user accessed to RegistrationView with REGISTRATION_OPEN = False

template_name = u'registration/registration_closed.html'

class registration.views.**RegistrationCompleteView**(***kwargs*)
Bases: django.views.generic.base.TemplateView

A simple template view for registration complete

get_context_data(***kwargs*)

template_name = u'registration/registration_complete.html'

class registration.views.**RegistrationView**(**args*, ***kwargs*)
Bases: django.views.generic.edit.FormMixin, django.views.generic.base.TemplateResponseMixi
django.views.generic.edit.ProcessFormView

A complex view for registration

GET: Display an RegistrationForm which has username, email1 and email2 for registration. email1 and email2 should be equal to prepend typo.

`form` and `supplement_form` is in context to display these form.

POST: Register the user with passed `username` and `email1`

dispatch(*request*, **args*, ***kwargs*)

form_invalid(*form*, *supplement_form=None*)

form_valid(*form*, *supplement_form=None*)
register user with `username` and `email1`

this method is called when form validation has successed.

get(*request*, **args*, ***kwargs*)

get_disallowed_url()
get registration closed url via backend

get_form_class()
get registration form class via backend

get_success_url()
get registration complete url via backend

get_supplement_form(*supplement_form_class*)
get registration supplement form instance

get_supplement_form_class()
get registration supplement form class via backend

post(*request*, **args*, ***kwargs*)

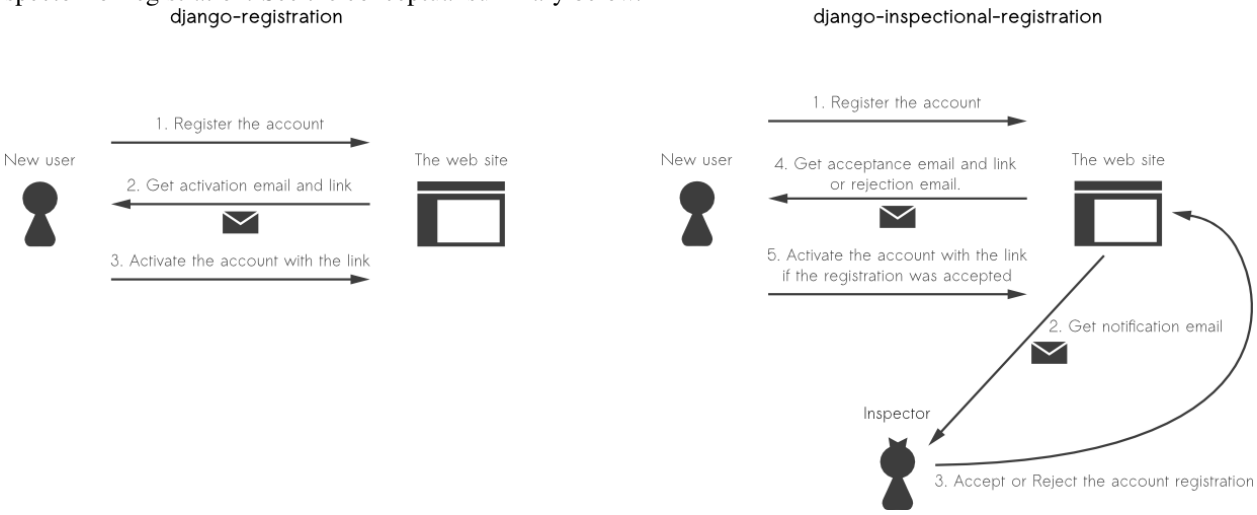
template_name = u'registration/registration_form.html'

Module contents

CHAPTER 2

The difference between django-registration

While django-registration requires 3 steps for registration, django-inspectional-registration requires 5 steps and inspector for registration. See the conceptual summary below.



CHAPTER 3

For translators

You can compile the latest message files with the following command

```
$ python setup.py compile_messages
```

The command above is automatically called before `sdist` command if you call `python manage.py sdist`.

CHAPTER 4

Backward incompatibility

Because of an issue#24, django-inspectional-registration add the following three new options.

- `REGISTRATION_DJANGO_AUTH_URLS_ENABLE` If it is `False`, django-inspectional-registration do not define the views of django.contrib.auth. It is required to define these view manually. (Default: `True`)
- `REGISTRATION_DJANGO_AUTH_URL_NAMES_PREFIX` It is used as a prefix string of view names of django.contrib.auth. For backward compatibility, set this value to `'auth_'`. (Default: `''`)
- `REGISTRATION_DJANGO_AUTH_URL_NAMES_SUFFIX` It is used as a suffix string of view names of django.contrib.auth. For backward compatibility, set this value to `''`. (Default: `''`)

This changes were introduced from version 0.4.0, to keep the backward compatibility, write the following in your settings module.

```
REGISTRATION_DJANGO_AUTH_URLS_ENABLE = True
REGISTRATION_DJANGO_AUTH_URL_NAMES_PREFIX = 'auth_'
REGISTRATION_DJANGO_AUTH_URL_NAMES_SUFFIX = ''
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

r

A

B

C

D

E

F

G

H

I

L

M

N

O

S

T

U

W