django-organizations Documentation

Release 0.9.2

Ben Lopatin

Jul 09, 2017

Contents

django-organizations is an application that provides group account functionality for Django, allowing user access and rights to be consolidated into group accounts.

Contents:

CHAPTER 1

Getting started

Django-organizations allows you to add multi-user accounts to your application and tie permissions, events, and other data to organization level accounts.

The core of the application consists of three models:

- An **organization** model; the group object. This is what you would associate your own app's functionality with, e.g. subscriptions, repositories, projects, etc.
- An **organization user**; a *through* model relating your **users** to your **organization**. It provides a convenient link for organization ownership (below) and also a way of storing organization/user specific information.
- An **organization owner**. This model links to an **organization user** who has rights over the life and death of the organization.

You can allow users to invite other users to their organizations and register with organizations as well. This functionality is provided through "backend" interfaces so that you can customize this code or use arbitrary user registration systems.

Installation

First add the application to your Python path. The easiest way is to use *pip*:

```
pip install django-organizations
```

Upgrading Django for existing installations

If you are upgrading the Django version of an existing deployment that deprecates South in favor of Django's native migrations, you will need to fake the migrations for Django Organizations (if you have installed the app directly, of course).

Configuration

Ensure that you have a user system in place to connect to your organizations.

To install the default models add *organizations* to *INSTALLED_APPS* in your settings file.:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'organizations',
)
```

This should work for the majority of cases, from either simple, out-of-the-box installations to custom organization models.

If however you want to use single-table customized organization models and/or custom organization user models, it may be best to treat Django organizations as a library and *not* install it in your Django project. See the *Advanced customization using abstract models* section.

URLs

If you plan on using the default URLs, hook the application URLs into your main application URL configuration in *urls.py*. If you plan on using the invitation/registration system, set your backend URLs, too:

```
from organizations.backends import invitation_backend

urlpatterns = [
    url(r'^accounts/', include('organizations.urls')),
    url(r'^invitations/', include(invitation_backend().get_urls())),
]
```

Registration & invitation

You can specify a different *invitation backend* in your project settings, and the *invitation_backend* function will provide the URLs defined by that backend. You can do the same with the *registration backend*:

```
INVITATION_BACKEND = 'myapp.backends.MyInvitationBackend'
REGISTRATION_BACKEND = 'myapp.backends.MyRegistrationBackend'
```

Auto slug field

Historically Django-Organizations relied on django-extensions for the base AutoSlugField. While django-extensions is great, this does require that every project install django-extensions for this one small feature.

If you decide to use the default django-organization models by adding *organizations* to your INSTALLED_APPS, you can choose a different AutoSlugField. Just specify the full dotted path like so:

```
ORGS_SLUGFIELD = 'django_extensions.db.fields.AutoSlugField'
```

While you can specify the source of this class, **its interfaces must be consistent**, including keyword arguments. Otherwise you will end up generating extraneous and possibly conflicting migrations in your own app. The SlugField must accept the *populate_from* keyword argument.

Users and multi-account membership

The key to these relationships is that while an *OrganizationUser* is associated with one and only one *Organization*, a *User* can be associated with multiple *OrganizationUsers* and hence multiple *Organizations*.

Note: This means that the OrganizationUser class cannot be used as a UserProfile as that requires a one-to-one relationship with the User class. User profile information is better provided by a profile specific model.

In your project you can associate accounts with things like subscriptions, documents, and other shared resources, all of which the account users can then access.

Views and Mixins

Hooking the django-organizations URLs into your project provides a default set of views for accessing and updating organizations and organization membership.

The included class based views are based on a set of mixins that allow the views to limit access by a user's relationship to an organization and that query the appropriate organization or user based on URL keywords.

Implementing in your own project

While django-organizations has some basic usability 'out-of-the-box', it's designed to be used as a foundation for project specific functionality. The *view mixins* should provide base functionality from which to work for most projects, and the *Cooking with Django Organizations* section provides detailed examples for various integration scenarios.

CHAPTER 2

Basic usage

After installing django-organizations you can make basic use of the accounts with minimal configuration.

Views

The application's default views and URL configuration provide functionality for account creation, user registration, and account management.

Creating accounts

Note: This is a to-do item, and an opportunity to contribute to the project!

User registration

You can register new users and organizations through your project's own system or use the extensible invitation and registration backends.

The default invitation backend accepts an email address and returns the user who either matches that email address or creates a new user with that email address. The view for adding a new user is then responsbile for adding this user to the organization.

The *OrganizationSignup* view is used for allowing a user new to the site to create an organization and account. This view relies on the registration backend to create and verify a new user.

The backends can be extended to fit the needs of a given site.

Creating accounts

When a new user signs up to create an account - meaning a new UserAccount for a new Account - the view creates a new *User*, a new *Account*, a new *AccountUser*, and a new *AccountOwner* object linking the newly created *Account* and *AccountUser*.

Adding users

The user registration system in django-organizations is based on the same token generating mechanism as Django's password reset functionality.

Changing ownership

Changing ownership of an organization is as simple as updating the *OrganizationOwner* such that it points to the new user. There is as of yet no out of the box view to do this, but adding your own will be trivial.

Invitation & registration backends

The invitation and registration backends provide a way for your account users to add new users to their accounts and if your application allows it, for users to create their own accounts at registration. Each base backend class is designed to provide a common interface which your backend classes can use to work with whatever user models, registration systems, additional account systems, or any other tools you need for your site.

CHAPTER 3

Customizing your organizations

The use cases from which django-organizations originated included more complex ways of determining access to the views as well as additional relationships to organizations. The application is extensible with these use cases in mind.

Custom organization models

Let's say you had an Account model in your app, which defined a group account to which multiple users could belong, and also had its own logo, a foreign key to a subscription plan, a website URL, and some descriptive information. Also, this client organization is related to a service provider organization.:

```
class ServiceProvider(Organization):
    """Now this model has a name field and a slug field"""
    url = models.URLField()


class Client(Organization):
    """Now this model has a name field and a slug field"""
    service_provider = models.ForeignKey(ServiceProvider,
            related_name="clients")
    subscription_plan = models.ForeignKey(SubscriptionPlan)
    subscription_start = models.DateField()
    url = models.URLField()
    description = models.TextField()

    objects = models.Manager()
```

Now the *ServiceProvider* and *Client* objects you create have the attributes of the Organization model class, including access to the OrganizationUser and OrganizationOwner models. This is an indirect relationship through a join in the database - this type of inheritance is multi-table inheritance so there will be a Client table and an Organization table; the latter is what the OrganizationUser and OrganizationOwner tables are still linked to.

Custom user model

By default django-organizations will map User objects from django's *contrib.auth* application to an Organization. However you can change this by specifying a different model in your settings using the *AUTH_USER_MODEL* setting. This should include the appname and model name in a string like so:

```
AUTH_USER_MODEL = 'auth.User'
```

or:

```
AUTH_USER_MODEL = 'myapp.MyUser'
```

Note: If you choose a different user class make sure to pay attention to the API. If it differs from the *auth.User* API you will likely need to use an extended backend, if you are not already.

This is worth noting here because Django Organizations is compatible with different user models in Django 1.4, which preceds the official swappable users feature in Django 1.4.

View mixins

Common use cases for extending the views include updating context variable names, adding project specific functionality, or updating access controls based on your project:

```
class ServiceProvidersOnly(LoginRequired, OrganizationMixin):
    def dispatch(self, request, *args, **kwargs):
        self.request = request
        self.args = args
        self.kwargs = kwargs
        self.organization = self.get_organization()
        self.service_provider = self.organization.provider
        if not self.organization.is_admin(request.user) and not \
                self.service_provider.is_member(request.user):
            raise PermissionDenied(_("Sorry, admins only"))
        return super(AdminRequiredMixin, self).dispatch(request, *args,
                **kwargs)
```

This mixin implements the same restriction as the *AdminRequiredMixin* mixin and adds an allowance for anyone who is a member of the service provider:

```
class AccountUpdateView(ServiceProviderOnly, BaseOrganizationUpdate):
    context_object_name = "account"
    template_name = "myapp/account_detail.html"

    def get_context_data(self, **kwargs):
        context = super(AccountUpdateView, self).get_context_data(**kwargs)
        context.update(provider=self.service_provider)
        return context
```

The *ServiceProvidersOnly* mixin inherits from the *LoginRequired* class which is a mixin for applying the *login_required* decorator. You can write your own (it's fairly simple) or use the convenient mixins provided by django-braces.

It would also have been possible to define the *ServiceProvidersOnly* without inheriting from a base class, and then defining all of the mixins in the view class definition. This has the benefit of explicitness at the expense of verbosity:

```
class ServiceProvidersOnly(object):
    def dispatch(self, request, *args, **kwargs):
        self.request = request
        self.args = args
        self.kwargs = kwargs
        self.organization = self.get_organization()
        self.service_provider = self.organization.provider
        if not self.organization.is_admin(request.user) and not \
                self.service_provider.is_member(request.user):
            raise PermissionDenied(_("Sorry, admins only"))
        return super(AdminRequiredMixin, self).dispatch(request, *args,
                **kwargs)


class AccountUpdateView(LoginRequired, OrganizationMixin,
                    ServiceProviderOnly, BaseOrganizationUpdate):
    context_object_name = "account"
    template_name = "myapp/account_detail.html"

    def get_context_data(self, **kwargs):
        context = super(AccountUpdateView, self).get_context_data(**kwargs)
        context.update(provider=self.service_provider)
        return context
```

While this isn't recommended in your own project, the mixins in django-organizations itself will err on the side of depending on composition rather than inheritence from other mixins. This may require defining a mixin in your own project that combines them for simplicity in your own views, but it reduces the inheritence chain potentially making functionality more difficult ot identify.

Note: The view mixins expressly allow superusers to access organization resources. If this is undesired behavior you will need to use your own mixins.

User registration and invitations

User registration and invitation plays an important role in how you will actually use Django Organizations, but it is a relatively minor aspect of the app. The default backends for both registration and invitation try to provide as little functionality to accomplish the task for most scenarios. These can be extended and customized in your own project provided that you expose a few consistent interfaces.

Creating the backend

Here we'll create a slightly modified invitation backend. The default backend presumes that your user model has a *username* attribute. If you're simply using the email address as your user identifier with a custom user model, this field might be missing.

The default *invite_by_email* method - which is part of the exposed interface - sends an invitation to the user based on the email address, and creates an inactive user account if there is no matching user. It satisfies the *auth.User* username's not null condition by filling the field with the contents of a freshly generated UUID.

In the example accounts app you would create a file named *backends.py*.:

```
from organizations.backends.defaults import InvitationBackend


class CustomInvitations(InvitationBackend):
    def invite_by_email(self, email, sender=None, request=None, **kwargs):
      try:
          user = self.user_model.objects.get(email=email)
      except self.user_model.DoesNotExist:
          user = self.user_model.objects.create(email=email,
                  password=self.user_model.objects.make_random_password())
          user.is_active = False
          user.save()
      self.send_invitation(user, sender, **kwargs)
      return user
```

This removes the username from the *create* method.

Configuring the backend

In your settings file you will need to specify that your backend should be used:

```
INVITATION_BACKEND = 'accounts.backends.CustomInvitations'
```

Your URLs can be configured as normal:

```
from organizations.backends import invitation_backend

urlpatterns = [
    ...
    url(r'^invite/', include(invitation_backend().get_urls())),
]
```

The *invitation_backend* function simply returns the URL patterns from the *get_urls* method of the specified backend.

CHAPTER 4

Cooking with Django Organizations

This section aims to provide some clear examples of how to build your application with Django organizations. The out-of-the-box setup works but is unlikely to meet your needs without some further customization.

Proxy models

The simplest way to customize an installation of Django organizations (with respect to the database) is to modify only the Django admin interface to your organizations.

In this example, an organization model is provided in a site but the organizations require a new name. Site admins should be able to add, edit, and delete account organizations, as well as users for those accounts.

Proxy models for convenience

This can be accomplished with proxy models in your models module.:

```
from organizations.models import Organization, OrganizationUser

class Account(Organization):
    class Meta:
        proxy = True

class AccountUser(OrganizationUser):
    class Meta:
        proxy = True
```

In your own models you could just add *verbose_name* and *verbose_name_plural* attributes in your class's *Meta* class, but you'd have to extend those classes here to do that anyhow. The *proxy* attribute ensures that this is just a wrapper around the class and the database is not changed from the default tables. Note that the *OrganizationOwner* is absent. In this particular example there's no need to expose that separately.

User focused admin interface

The admin interface should allow a site admin to modify the organizations and the site users. We want to avoid requiring site admins to separately manage users through the *auth.User* or other user interface *and* an organization interface. It seems convenient as a developer to just let site admins pick users from the organization's many-to-many *users* field, but this is cumbersome and doesn't factor in user invitations.

So the strategy here is to take advantage of the explicit through model's organization foreign key and provide a limited user interface through the account user form.

Model admin definitions

```
from django.contrib import admin
from organizations.models import (Organization, OrganizationUser,
    OrganizationOwner)
from myapp.forms import UserAdminForm
from myapp.models import Account, AccountUser

class AccountUserAdmin(admin.ModelAdmin):
    form = UserAdminForm()

admin.site.unregister(Organization)
admin.site.unregister(OrganizationUser)
admin.site.unregister(OrganizationOwner)
admin.site.register(Account)
admin.site.register(AccountUser, AccountUserAdmin)
```

It's very simple. All it does is ensure that the default Organization model interfaces are hidden and then substitute the form class on the AccountUser admin.

That form is where the business all happens

The admin form class

We'll go through this piece by piece, but here's the full class:

```
from django import forms
from django.conf import settings
from django.contrib.sites.models import Site
from myapp.models import AccountUser

class AccountUserForm(forms.ModelForm):
    """
    Form class for editing OrganizationUsers *and* the linked user model.
    """
    first_name = forms.CharField(max_length=100)
    last_name = forms.CharField(max_length=100)
    email = forms.EmailField()

    class Meta:
        exclude = ('user', 'is_admin')
        model = AccountUser

    def __init__(self, *args, **kwargs):
        super(AccountUserForm, self).__init__(*args, **kwargs)
        if self.instance.pk is not None:
```

```
            self.fields['first_name'].initial = self.instance.user.first_name
            self.fields['last_name'].initial = self.instance.user.last_name
            self.fields['email'].initial = self.instance.user.email

    def save(self, *args, **kwargs):
        """
        This method saves changes to the linked user model.
        """
        if self.instance.pk is None:
            site = Site.objects.get(pk=settings.SITE_ID)
            self.instance.user = invitation_backend().invite_by_email(
                    self.cleaned_data['email'],
                    **{'first_name': self.cleaned_data['first_name'],
                        'last_name': self.cleaned_data['last_name'],
                        'organization': self.cleaned_data['organization'],
                        'domain': site})
        self.instance.user.first_name = self.cleaned_data['first_name']
        self.instance.user.last_name = self.cleaned_data['last_name']
        self.instance.user.email = self.cleaned_data['email']
        self.instance.user.save()
        return super(AccountUserForm, self).save(*args, **kwargs)
```

This is a model form class but primarily manages a linked model.

The *AccountUser* model only has three fields: a foreign key to the organization, a foreign key to the user, and since this is our default class, a Boolean field for admins. The form will only show a choice for the organization. Meanwhile, the site admin will have a chance to view and edit the name and email address of the user, pulled from the underlying user model, e.g. *auth.User*.

The *__init__* method is responsible for populating the form with the data for existing account users. It calls the super method first which is necessary to create the fields. It checks if the primary key is none, rather than testing the attribute, since the attribute will be there whether or not the model has been saved yet - it will just be a *NoneType*.

The *save* method only does a little bit more. There's some logic there for populating the invitation email (more on that below) but mostly this just updates the linked user, instead of just the link to the user.

Handling user invitations

Each invited user is sent an email with a unique registration link. On the registration page they have the opportunity to update their name and create their own password. One of the things this form does is prevent users from changing their email address. There's no reason your own project needs to do this, of course.:

```
from django import forms
from django.conf import settings
from django.contrib.sites.models import Site
from organizations.backends import invitation_backend
from organizations.backends.forms import UserRegistrationForm
from partners.models import PartnerUser

class RegistrationForm(UserRegistrationForm):
    """
    Form class that allows a user to register after clicking through an
    invitation.
    """
    first_name = forms.CharField(max_length=30)
    last_name = forms.CharField(max_length=30)
    email = forms.EmailField(widget=forms.TextInput(
```

```
        attrs={'class': 'disabled', 'readonly': 'readonly'}))
    password = forms.CharField(max_length=128, widget=forms.PasswordInput)
    password_confirm = forms.CharField(max_length=128, widget=forms.PasswordInput)

    def clean(self):
        password = self.cleaned_data.get("password")
        password_confirm = self.cleaned_data.get("password_confirm")
        if password != password_confirm or not password:
            raise forms.ValidationError("Your password entries must match")
        return super(RegistrationForm, self).clean()

class AccountUserForm(forms.ModelForm):
    # See above
```

Custom org with simple inheritance

Simply extending the organization model with your own requires the least amount of fuss and for most applications will probably suffice. This entails using the stock *OrganizationUser* model and multi-table inheritance to support additional fields on your organization model.

Models

Here's an example from a sport team management application.:

```
from django.db import models
from organizations.models import Organization
from sports.models import Sport

class Team(Organization):
    sport = models.ForeignKey(Sport, related_name="teams")
    city = models.CharField(max_length=100)
```

That's all that's required to update your models. The *Team* model will use the default *OrganizationUser* and *OrganizationOwner* models.

Views

The class based views can be configured to refer to different model classes and context variable names by adding a view attributes in your own class or in the *as_view* class method call.:

```
class TeamDetail(BaseOrganizationDetail):
    org_model = Team
    org_context_name = 'team'
```

Multiple organizations with simple inheritance

You can take the inheritance strategy one step further and add additional organization classes if need be.:

```
from django.db import models
from organizations.models import Organization
from sports.models import Sport

class Association(Organization):
    sport = models.ForeignKey(Sport, related_name="associations")

class Team(Organization):
    association = models.ForeignKey(Association, related_name="teams")
    city = models.CharField(max_length=100)
```

As in this example you can add them in the same app although it probably makes more sense to add them in their own apps.

Advanced customization using abstract models

As of version 0.2.0 you can add your own fully customized models using unique table sets, i.e. single table inheritance. In order to do this, your app should define an organization model, an organization user model, and an organization owner model, each inheriting from one of the abstract models provided in `organizations.abstract`. Here's an example from an *accounts* app:

```
from django.db import models
from organizations.abstract import (AbstractOrganization,
                                    AbstractOrganizationUser,
                                    AbstractOrganizationOwner)

class Account(AbstractOrganization):
    monthly_subscription = models.IntegerField(default=1000)

class AccountUser(AbstractOrganizationUser):
    user_type = models.CharField(max_length=1, default='')

class AccountOwner(AbstractOrganizationOwner):
    pass
```

This will create the following tables:

- *accounts_account*
- *accounts_accountuser*
- *accounts_accountowner*

The *accounts_account* table will include all of the necessary fields for this and only this organization model.

Note: Unlike in the example of multi-table inheritance, you cannot add more than one custom organization model to an individual app. Each additional organization class you want must be defined in its own app. Only one organization set per app.

A more minimalistic approach using base models

The base models provided in `organizations.base` marked with the *Base* suffix provide the almost-bare minimum fields required to manage organizations. These models are very basic and can be used if your implementation must differ considerably from the default one.

Here's an example of a custom *accounts* inheriting the minimal *Base* models:

```
from django.db import models
from organizations.base import (OrganizationBase,
                                OrganizationUserBase,
                                OrganizationOwnerBase)

class Account(OrganizationBase):
    monthly_subscription = models.IntegerField(default=1000)

class AccountUser(OrganizationUserBase):
    user_type = models.CharField(max_length=1, default='')

class AccountOwner(OrganizationOwnerBase):
    pass
```

Difference between abstract and base models

The **abstract models** (provided in `organizations.abstract`) include timestamps, a slug field on the organization, and an `is_admin` field on the organization user. The first two are implemented with additional dependencies. Use these models if you are happy with the way this additional logic is implemented.

The **base models** (provided in `organizations.base`) instead provide only the bare minimum fields required to implement and manage organizations: if you want a slug field or timestamps on your models, you'll need to add those in. However you can do so however you want. And if you don't want any of those fields, you don't have to take them.

Extending the base admin classes

If you chose the "single table inheritance" approach, you may want to reuse the base admin classes too, in order to avoid having too much boilerplate code in your application, eg:

```
from django.contrib import admin

from .base_admin import (BaseOwnerInline,
                         BaseOrganizationAdmin,
                         BaseOrganizationUserAdmin,
                         BaseOrganizationOwnerAdmin)
from .models import Organization, OrganizationUser, OrganizationOwner


class OwnerInline(BaseOwnerInline):
    model = OrganizationOwner


class OrganizationAdmin(BaseOrganizationAdmin):
    inlines = [OwnerInline]


class OrganizationUserAdmin(BaseOrganizationUserAdmin):
    pass


class OrganizationOwnerAdmin(BaseOrganizationOwnerAdmin):
    pass
```

```
admin.site.register(Organization, OrganizationAdmin)
admin.site.register(OrganizationUser, OrganizationUserAdmin)
admin.site.register(OrganizationOwner, OrganizationOwnerAdmin)
```

Restricting and isolating resources

A fairly common use case for group accounts is to associate resources of one kind or another with that group account. Two questions arise next, how to associate this content with the accounts, and secondly how to restrict access to group members?

Associating resources

The simplest way to associate resources with an account is with a foreign key.:

```
class Account(Organization):
    """We'll skip any other fields or methods for the example"""

class MeetingMinutes(models.Model):
    """A representative resource model"""
    account = models.ForeignKey('Account', related_name='meeting_minutes')
```

We now have a definite way of linking our meeting minutes resource with an account. Accessing only those meeting minutes related to the account is straightforward using the related name.:

```
account = get_object_or_404(Account, pk=pk)
relevant_meeting_minutes = account.meeting_minutes.all()
```

This works if the resource is defined in your project. If you're pulling this in from another app, e.g. a third party Django app, then you can't directly add a foreign key to the model. You can create a linking model which can be used in a similar fashion.:

```
from third_party_app.models import Document

class DocumentLink(models.Model):
    account = models.ForeignKey('Account', related_name="document_links")
    document = models.ForeignKey('Document', unique=True)
```

The linking model should in *most scenarios* enforce uniquness against the linked resource model to prevent multiple organizations from having access to the resource.

Providing access may be a little less straightforward. You can use the related name as before, however that will result in a queryset of *DocumentLink* objects, rather than *Document* as expected. This works, but for more foolproof results you might add a custom queryset method or define a *ManyToManyField* on your group account model if this makes sense with regard to your application hierarchy.:

```
class Account(Organization):
    documents = models.ManyToManyField('third_party_app.Document', through=
↪'DocumentLink')
```

Restricting access

Access restriction is based on two levels: managers (queryset) to limit the available data in a way that is easy to understand and work with, and view mixins or decorators to actually allow users (you can also use middleware in some implementations).

Managers and querysets work as demonstrated above to provide a foolproof way of getting only the relevant resources for a given organization. Where you can, avoid explicit access filters in your views, forms, management commands, etc. Relying on the filters from related managers whenever possible reduces the room for mistakes with result in data leaks.

The mixins in the django-organizations codebase provide a good starting point for additional restrictions. Note that you can also use decorators for functional views in the same way.

As an example from directly within a simple view class, you might want to restrict access to a particular document based on the organization. The "one liner" solution is to check for a match using a filter against the user.:

```
@login_required
def document_view(request, document_pk):
    doc = get_object_or_404(Document, pk=document_pk, account__users=request.user)
    return render(request, "document.html", {"document": doc})
```

An improvement for clarity is to define a manager method that encapsulates some of the logic.:

```
@login_required
def document_view(request, document_pk):
    try:
        doc = Document.objects.for_user(request.user).get(pk=document_pk)
    except Document.DoesNotExist:
        raise Http404
    return render(request, "document.html", {"document": doc})
```

The *for_user* method can then reduce the queryset to only documents belonging to *any* organization which the current user is a member of.

CHAPTER 5

Reference

Contents:

Models

Organization

OrganizationUser

OrganizationOwner

Managers

OrgManager

Base manager class for the *Organization* model.

`OrgManager.`**`get_for_user`**`(`*user*`)`
Returns a QuerySet of Organizations that the given user is a member of.

ActiveOrgManager

This manager extends the *OrgManager* class by defining a base queryset including only active Organizations. This manager is accessible from the *active* attribute on the *Organization* class.

Mixins

OrganizationMixin

OrganizationUserMixin

MembershipRequiredMixin

AdminRequiredMixin

OwnerRequiredMixin

Views

Base views

BaseOrganizationList

BaseOrganizationDetail

BaseOrganizationCreate

BaseOrganizationUpdate

BaseOrganizationDelete

BaseOrganizationUserList

BaseOrganizationUserDetail

BaseOrganizationUserCreate

BaseOrganizationUserRemind

BaseOrganizationUserUpdate

BaseOrganizationUserDelete

Controlled views

OrganizationCreate

OrganizationDetail

OrganizationUpdate

OrganizationDelete

OrganizationUserList

OrganizationUserDetail

OrganizationUserUpdate

OrganizationUserCreate

OrganizationUserRemind

OrganizationUserDelete

Misc. views

OrganizationSignup

signup_success

Forms

OrganizationForm

OrganizationUserForm

OrganizationUserAddForm

OrganizationAddForm

SignUpForm

Settings

settings.**INVITATION_BACKEND**
: The full dotted path to the invitation backend. Defaults to:

```
INVITATION_BACKEND = 'organizations.backends.defaults.InvitationBackend'
```

settings.**REGISTRATION_BACKEND**
: The full dotted path to the regisration backend. Defaults to:

```
REGISTRATION_BACKEND = 'organizations.backends.defaults.RegistrationBackend'
```

settings.**AUTH_USER_MODEL**
: This setting is introduced in Django 1.5 to support swappable user models. The defined here will be used by django-organizations as the related user class to Organizations.

  Though the swappable user model functionality is absent, this setting can be used in Django 1.4 with django-organizations to relate a custom user model. If undefined it will default to:

```
AUTH_USER_MODEL = 'auth.User'
```

Invitation and Registration Backends

The purpose of the backends is to provide scaffolding for adding and managing users and organizations. **The scope is limited to the basics of adding new users and creating new organizations**.

While the default backends should suffice for basic implementations, the backends are designed to be easily extended for your specific project needs. If you make use of a profile model or a user model other than *auth.User* you should extend the releveant backends for your own project. If you've used custom URL names then you'll also want to extend the backends to use your own success URLs.

You do not have to implement these backends to use django-organizations, but they will make user management within accounts easier.

The two default backends share a common structure and interface. This includes methods for sending emails, generating URLs, and template references.

The backend URLs will need to be configured to allow for registration and/or user activation. You can add these by referring to the backend's *get_urls* method::

```
from organizations.backends import invitation_backend

urlpatterns = [
    url(r'^invitations/', include(invitation_backend().get_urls())),
 ]
```

Registration Backend

The registration backend is used for creating new users with new organizations, e.g. new user sign up.

Attributes

`RegistrationBackend.`**`activation_subject`**
Template path for the activation email subject. Default:

```
invitation_subject = 'organizations/email/activation_subject.txt'
```

`RegistrationBackend.`**`activation_body`**
Template path for the activation email body. Default:

```
invitation_body = 'organizations/email/activation_body.html'
```

`RegistrationBackend.`**`reminder_subject`**
Template path for the reminder email subject. Default:

```
reminder_subject = 'organizations/email/reminder_subject.txt'
```

`RegistrationBackend.`**`reminder_body`**
Template path for the reminder email body. Default:

```
reminder_body = 'organizations/email/reminder_body.html'
```

`RegistrationBackend.`**`form_class`**
Form class which should be used for activating a user account when registering. Default:

```
form_class = UserRegistrationForm
```

Invitation backend

The invitation backend is used for adding new users to an *existing organization*.

Attributes

`InvitationBackend.`**`invitation_subject`**
Template path for the invitation email subject. Default:

```
invitation_subject = 'organizations/email/invitation_subject.txt'
```

`InvitationBackend.`**`invitation_body`**
Template path for the invitation email body. Default:

```
invitation_body = 'organizations/email/invitation_body.html'
```

`InvitationBackend.`**`reminder_subject`**
Template path for the reminder email subject. Default:

```
reminder_subject = 'organizations/email/reminder_subject.txt'
```

`InvitationBackend.`**`reminder_body`**
Template path for the reminder email body. Default:

```
reminder_body = 'organizations/email/reminder_body.html'
```

`InvitationBackend.`**`form_class`**
Form class which should be used for activating a user account in response to an invitation. Default:

```
form_class = UserRegistrationForm
```

Methods

The primary methods of interest are the *invite_by_email* method and the *get_success_url* method.

`InvitationBackend.`**`get_success_url`**`()`
This method behaves as expected and returns a URL to which the user should be redirected after successfully activating an account. By default it returns the user to the organization list URL, but can be configured to any URL:

```
def get_success_url(self):
    return reverse('my_fave_app')
```

`InvitationBackend.`**`invite_by_email`**(*email*, *sender=None*, *request=None*, ***kwargs*)
This is the primary interface method for the invitation backend. This method should be referenced from your invitation form or view and if you need to customize what happens when a user is invited, this is where to do it.

Usage example in a form class:

```
class AccountUserAddForm(OrganizationUserAddForm):

    class Meta:
        model = OrganizationUser

    def save(self, *args, **kwargs):
        try:
            user = get_user_model().objects.get(email__iexact=self.cleaned_data[
↪'email'])
        except get_user_model().MultipleObjectsReturned:
            raise forms.ValidationError("This email address has been used␣
↪multiple times.")
        except get_user_model().DoesNotExist:
            user = invitation_backend().invite_by_email(
                    self.cleaned_data['email'],
                    **{'domain': get_current_site(self.request),
                        'organization': self.organization})

        return OrganizationUser.objects.create(user=user,
                organization=self.organization)
```

Note: As the example shows, the invitation backend does not associate the individual user with the organization account, it only creates the user so it can be associated in addition to sending the invitation.

Use additional keyword arguments passed via ***kwargs* to include contextual information in the invitation, such as what account the user is being invited to join.

`InvitationBackend.`**`activate_view`**`(`*request*, *user_id*, *token*`)`

This method is a view for activating a user account via a unique link sent via email. The view ensures the token matches a user and is valid, that the user is unregistered, and that the user's entered data is valid (e.g. password, names). User entered data is validated against the *form_class*.

The view then ensures the user's *OrganizationUser* connections are activated, logs the user in with the entered credentials and redirects to the success URL.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index