django-organizations Documentation

Release 0.1.6

Ben Lopatin

March 26, 2013

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

ONE

GETTING STARTED

django-organizations allows you to add multi-user accounts to your application and tie permissions and actions to organization level accounts.

The core of the application consists of these three models:

- **Organization** The group object. This is what you would associate your own app's functionality with, e.g. subscriptions, repositories, projects, etc.
- **OrganizationUser** A custom *through* model for the ManyToMany relationship between the *Organization* model and your user model. It stores additional information about the user specific to the organization and provides a convenient link for organization ownership.
- **OrganizationOwner** The user with rights over the life and death of the organization. This is a one to one relationship with the *OrganizationUser* model. This allows users to own multiple organizations and makes it easy to enforce ownership from within the organization's membership.

1.1 Installation

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

```
pip install django-organizations
```

You should install by downloading the source and running:

```
$ python setup.py install
```

1.2 Configuration

First ensure that you have a user system in place to connect to your organizations. django-organizations will work just fine with the Django's *contrib.auth* package. To use it, make sure you have *django.contrib.auth* installed.

Add the *organizations* application to your *INSTALLED_APPS* list:

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

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

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

```
ORGS_INVITATION_BACKEND = 'myapp.backends.MyInvitationBackend'
```

1.3 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. That is better provided by your own project or application.

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.

For many projects a simple one-user-per-account model will suffice, and this can be handled quite ably in your own application's logic.

1.4 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 <https://docs.djangoproject.com/en/1.4/topics/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.

1.5 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 unique situations.

CHAPTER

TWO

BASIC USAGE

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

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

2.1 Authentication

django-organizations relies on *django.contrib.auth* for the *User* model and authentication mechanisms.

2.2 Creating accounts

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

2.3 User registration

Registering new users with organizations is accomplished by 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.

2.3.1 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*.

2.3.2 Adding users

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

2.4 Changing ownership

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

CHAPTER

THREE

CUSTOMIZING USAGE

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.

3.1 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.

3.2 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.

3.3 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):
            return HttpResponseForbidden(_("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 explitness 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):
            return HttpResponseForbidden(_("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 proejct, 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 behaviour you will need to use your own mixins.

CHAPTER

FOUR

INVITATION AND REGISTRATION BACKENDS

The backends are used for adding new users to organizations, either by way of self-registration or invitation by existing organization users. The default backends should suffice for simple implementations. 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.

4.1 Registration

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

4.2 Invitations

An invitation backend is used for adding new users to an existing organization. The backend should add existing users and create new users.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*