# Django Vox Documentation

**Alan Trick**

**May 23, 2019**

# Contents:

CHAPTER 1

# Getting Started

## 1.1 Installation

### 1.1.1 Getting the code

The recommended way to install django-vox is via pip (on Windows, replace `pip3` with `pip`)

```
$ pip3 install django-vox[markdown,twilio,html]
```

### 1.1.2 Configuring Django

Add `"django_vox"` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # ...
    "django_vox",
]
```

Additionally, you may want to configure certain *backends* depending on exactly what sort of other notifications you want.

## 1.2 The Demo

While you're welcome to use the setup instructions here, it may be easier to just try out the *demo* that comes with this package. The demo doesn't cover all the use cases (yet) but it does cover most of the standard stuff.

## 1.3 Registering the Models

The process of issuing and routing notifications in Django Vox happens in 4 steps.

1. **Notification definitions**: These definitions are registered, and attached to specific models.
2. **Recipients**: Any given notification can have 4 possible recipients.
    1. The model that the notification is attached to
    2. The actor of the notification (optional)
    3. The target of the notification (optional)
    4. The site contacts (these come built-in)
3. **Channels**: Additionally, each of these recipients can have zero, 1, or multiple "channels". For example, a User model might have one channel for the user themselves, one for the user's followers, and one for the user's emergency contact.
4. **Contact**: A contact specifies the actual address that a the notifications are sent to. Any model specified in a channel, should also define contacts in its registration, otherwise that recipient won't have any way of receiving the notifications.

Any model that will be either issuing notifications, or receiving them should have a registration. It might look something like this:

```
from django.db import models
from django_vox.registry import (VoxRegistration, Notification,
                                 Channel, objects, provides_contacts)

class User(models.Model):

    # required email, optional mobile_phone
    email = models.EmailField(blank=False)
    mobile_phone = models.CharField(blank=True, max_length=50)
    ...


class UserVox(VoxRegistration):


    @provides_contacts("email")
    def email_contact(self, instance, notification):
        yield instance.email

    @provides_contacts("sms")
    def email_contact(self, instance, notification):
        if instance.mobile_phone:
            yield instance.mobile_phone

    def get_channels(self):
        return {"": Channel.self(self)}

class PurchaseOrder(models.Model):

    customer = models.ForeignKey(User, on_delete=models.PROTECT)

    def save(self, *args, **kwargs):
        created = self.id is None
```

(continues on next page)

(continued from previous page)

```
        if not created:
            old = PurchaseOrder.objects.get(pk=self.pk)
        else:
            old = None
        super().save(*args, **kwargs)
        objects[type(self)].registration.post_save(
            created, old, self)


class PurchaseOrderVox(VoxRegistration):

    received = Notification(
        _("Notification that order was received."))
    on_hold = Notification(
        _("Notification that order is on hold."))

    def post_save(self, created, old, new):
        if created:
            self.received.issue(new)
        if old and not old.on_hold and new.on_hold:
            self.on_hold.issue(new)

    def get_channels(self):
        return {"cust": Channel.field(PurchaseOrder.customer)}

# the actual registration
objects.add(User, UserVox, regex=None)
objects.add(PurchaseOrder, PurchaseOrderVox, regex=None)
```

In the above example, you have a User model, which can receive emails, and optionally an SMS message. You also have purchase orders that have two notifications registered on them (`received` and `on_hold`). Whenever the purchase order is saved, it calls `post_save` on the registration object, and that fires the notifications themselves.

Once you've finished adding these, you'll need to regenerate the notifications table using the `make_notifications` management command:

```
python3 manage.py make_notifications
```

And there you have it. Now, in order for this to do anything useful, you'll need to add some appropriate *templates*. In this case, you'll want an email template for the "customer" recipient of the purchase order notifications, and possibly a template for a site contact too.

For more details on model registration and the various options, see the *Registrations in Detail* page.

## 1.4 One-time Messages from the Admin

The normal way to handle notifications is call `notification.issue(instance)` from within the code. It's also possible to manually issue notifications from the admin as long as a notification doesn't have an actor/target model. The other way of sending messages completely bypasses the `Notification` models and uses an Admin Action.

In order to send messages this way, you need to add the `django_vox.admin.notify` action to your `ModelAdmin` class. It might look something like this:

```
from django.contrib import admin
from django_vox.admin import notify

class UserAdmin(admin.ModelAdmin):
    actions = (notify, )

admin.site.register(YourUserModel, UserAdmin)
```

In order for this to work right, the model in question is treated as the channel, and so needs to have contacts registered for the appropriate backend & protocol that you want to use.

---

**Note:** Because we don't actually have a notification model here, a fake notification (`django_vox.models.OneTimeNotification`) is passed to the contact methods. This can be used if only want certain contacts to be accessible in this way.

---

CHAPTER 2

# Registrations in Detail

**Note:** This documentation assumes that you've read the *Getting Started* page.

## 2.1 Notifications

### 2.1.1 Specifying Actors and Targets

Notifications can have actors and targets. If you think of a notification as representing a specific action, an actor is whatever caused the action, and the target is whatever the action was done on. So for example, if have CMS application, and you might have a notification for page editing, the actor for the notification could be the user who edited the page, and the target might be the page that was edited. Finally, let's say you're using Django's built-in users. You might end up with code like this

```
# models for myapp
class Page(models.Model):
    content = models.TextField()

class PageVox(VoxRegistration):

    edited = Notification(
        _("Notification that the page was edited"),
        actor_type="auth.user",
        target_type="myapp.page",
    )
# Note: We're skipping the actual registration details here
# Let's say you have this view elsewhere to update a page

def update_page(request, page):
    # do some authentication
    form = PageForm(request.POST, instance=page)
```

(continues on next page)

(continued from previous page)

```
    if form.is_valid():
        form.save()
        PageVox.edited.issue(page, actor=request.user, target=page)
```

Note that now `actor` and `target` are specified when issuing the notification. These arguments are used if and only if there `actor_type` and `target_type` parameters set on the notification. Also note that these must be specified as keyword arguments (to reduce confusion).

In this case, we've triggered the notification in a view instead of in the model itself. That's a totally valid way of doing it, but it means you have to manually ensure the notification is triggered, and it won't get triggered in the admin unless you write your own ModelAdmin that does it.

### 2.1.2 Required Notifications

Normally, if a notification is issued, but either 1) no relevant notification templates are set up, 2) the channels are mis-configured, or 3) the contact methods aren't returning any results, nothing happens. In the case of 1 & 3, this is generally the desired behavior, and it doesn't mean there's anything wrong with what's going in. It's to be expected that sometimes the programmers will set up a notification in the code that either isn't actually used yet, or is only used some of the time.

Some messages though, really should get sent all the time, and you may not want such a failure to pass silently. If this is the case, you can pass `required=True` when you declare your notification. Required notifications will raise a `RuntimeError` exception if the notification was issued and no messages were actually sent.

## 2.2 Attachments

**Note:** Attachments are currently only supported by the email backends

Similar to how notifications are defined, you can also define attachments when making a registration class.

```
import pdfkit
from django_vox.registry import Attachment, VoxRegistration

class Page(models.Model):
    html_content = models.TextField()

class PageVox(VoxRegistration):

    pdf = Attachment(
        # attr is an instance method, or a callable on either
        # the model itself, or the registration class.
        # If it's a callable, it takes one argument(the model instance)
        attr="as_pdf",
        # you can also you mime_attr if you need a dynamic mime type
        mime_string="application/pdf",
        # defaults to "pdf" in this case, if left blank
        label=_("PDF Copy")
    )

    @staticmethod
    def as_pdf(instance):
```

(continues on next page)

(continued from previous page)

```
        """Generate PDF from page contents"""
        return pdfkit.from_string(instance.html_content, False)
```

The above example will let you attach a PDF copy of the page to any page notifications. Note that the actual assignment of attachments happens in the admin when you make notification templates, adding this just provides the option there.

## 2.3 Channels

As you saw before on the *Getting Started* page, you typically want to specify channels on an model that either has notifications, or is used as an actor or a target for notifications. Channels are specified in the `get_channels` method which returns a Mapping of string (the channel key) → a `django_vox.registry.Channel` object. There's current 3 ways to make `Channel` objects.

1. `Channel.self(self)`: this means you can send notifications directly to the model itself.
2. `Channel.field(Model.field)`: You can use this if "Model.field" is a ForeignKey, or a ManytoMany-Field.
3. `Channel(label, model class, function)`: You can also create your channel manually. In this case, label is a string, function is a callable that takes one argument (a model instance of the model that this registry is for) and returns an iterable of the "model class" objects.

```
from django.utils import timezone
from django.db import models
from django_vox.registry import Channel, VoxRegistration, provides_contacts

def youth_members(org):
    # 18 years ago
     adult_birthday = timezone.now().replace(
         year=timezone.now().year - 18)
    return org.members.filter(birthday__gte=adult_birthday)

class Organisation(models.Model):
    org_email = models.EmailField()
    # note: I'm omitting the User class and its registration
    members = models.ManyToManyField(to=User)

    @provides_contacts("email")
    def email_contact(self, instance, notification):
        yield instance.org_email

class OrganisationVox(VoxRegistration):

    def get_channels(self):
        return {
            "": Channel.self(self),
            "all": Channel.field(Organization.members),
            "youth": Channel.self(_("Youth"), User, youth_members),
        }
```

**Note:** The channel keys should be unique strings. They don't need to be long and fancy, and won't be visible to end users.

## 2.4 Builtin Registration Classes

You've already seen the `VoxRegistration` class already. There's another built-in registration class called `SignalVoxRegistration`. It uses Django's built-in model signals and provides three notifications (created, updated, and deleted) that automatically work. You can use this registration class directly, but if you want to contact anything besides site contacts, you'll want to subclass it and add `get_channels` and maybe contact methods.

## 2.5 Registration Inheritance

You can subclass registration classes, just like normal Python classes. There's a few things to be aware of.

1. You can only have one contact method for a given protocol. If a parent class has a method decorated with `@provides_contacts("email")` and the child class does too, the child class's method will be used.
2. The same applies to notification code names. Unless you manually specify A notification code name, however, its always the same as attribute name, so this is a moot point in practice.

## 2.6 Site Contacts

Site contacts are a special kind of contact that come build-in. You can set them up in the Django admin site. Site contacts are global to the site, but you can enable or disable them on a per-notification basis.

...

CHAPTER 3

# Backends

A backend is what is responsible for sending the actual messages. A backend implements a "protocol" like email or SMS. Multiple backends might implement the same protocol, so you might have a backend that sends email via SMTP, and another one that uses the mailgun API. The important thing is that all the backends implementing a specific protocol must accept the same form of addresses.

Most of these backends require extra dependencies that are not required for the base package. The specific extras are listed in the documentation, and you can mix/match them. For example:

```
# adds dependencies for markdown email, xmpp, twitter and the twilio
# backends
pip3 install django-vox[xmpp,markdown,twitter,twilio]
```

In order to add or remove backends, you need to set the `DJANGO_VOX_BACKENDS` setting in your projects `settings.py` file. The setting is a list of class names for backends that are in-use/enabled. If not set, the default is (assuming you have the required dependencies):

```
DJANGO_VOX_BACKENDS = (
    # disabled by default
    # 'django_vox.backends.activity.Backend',
    'django_vox.backends.html_email.Backend',
    'django_vox.backends.markdown_email.Backend',
    'django_vox.backends.postmark_email.Backend',
    'django_vox.backends.template_email.Backend',
    'django_vox.backends.twilio.Backend',
    'django_vox.backends.twitter.Backend',
    'django_vox.backends.slack.Backend',
    'django_vox.backends.json_webhook.Backend',
    'django_vox.backends.xmpp.Backend',
)
```

Django-vox provides a few built-in backends. Here's how to set them up and use them.

## 3.1 Activity Backend

**Protocol** `activity`

**Class** `django_vox.backends.activity.Backend`

This is the backend for Activity Streams support. Setup is covered on the *Activities* page.

## 3.2 Email Backends

**Protocol** `email`

These backends are a wrapper around Django's internal mailing system. As such, it uses all of the built-in email settings including `DEFAULT_FROM_EMAIL`, and everything that starts with `EMAIL` in the standard django settings.

There are 3 backends included:

- One that uses HTML (and converts it to text for you)
- One that uses Markdown (and coverts it to HTML for you)
- One that uses Django template blocks to specify both HTML & text (not recommended)

### 3.2.1 HTML Email Backend

**Class** `django_vox.backends.html_email.Backend`

When using this, the content of your template will have to be HTML. If you don't, it will be HTML anyways, but it will look real bad, and the god will frown on you. The backend automatically uses a library to convert the HTML into plain-text, so that there is a text version of the email, and so that the spam filters think better of you.

Incidentally, the subject field is not HTML formatted.

### 3.2.2 Markdown Email Backend

**Class** `django_vox.backends.markdown_email.Backend`

**Extra** `[markdown]`

This backend has you specify the content of your templates with markdown. While markdown doesn't give you quite the flexibility as HTML, it's a bit more intuitive.

### 3.2.3 Template-based Email Backend

**Class** `django_vox.backends.template_email.Backend`

This backend isn't recommended because it's probably too confusing to be wroth it. However, if you really need to tailor-make your emails, it's a solution that you can make work.

Writing notification templates for emails are a little more complicated than they are for the other backends, because emails can have multiple parts to them (subject, text, and html). The basic form looks like this:

```
{% block subject %}Email Subject{% endblock %}
{% block text_body %}Text body of email{% endblock %}
{% block html_body %}HTML body of email{% endblock %}
```

## 3.3 Postmark Templates

**Class** `django_vox.backends.postmark_email.Backend`

This backend requires one config setting: `DJANGO_VOX_POSTMARK_TOKEN`. It should be, unsurprisingly, your token for interacting with the postmark API. When using this backend, the 'Subject' field refers to Postmark's "template alias" and the template content should look something like this:

```
parameter_one: {{ content.attribute }}
parameter_two: {{ recipient.name }}
```

## 3.4 Twilio

**Protocol** `sms`

**Class** `django_vox.backends.twilio.Backend`

**Extra** `[twilio]`

The twilio backend uses Twilio's python library. It depends on 3 settings, all of which needs to be set for proper functioning.

| | |
|---|---|
| `DJANGO_VOX_TWILIO_ACCOUNT_SID` | Twilio account ID |
| `DJANGO_VOX_TWILIO_AUTH_TOKEN` | Twilio authentication token |
| `DJANGO_VOX_TWILIO_FROM_NUMBER` | Phone # to send Twilio SMS from |

## 3.5 Twitter

**Protocol** `twitter`

**Class** `django_vox.backends.twitter.Backend`

**Extra** `[twitter]`

The twitter backend allows you to post updates to twitter and (with the right permissions), send direct messages to your followers. In order to set it up, you first need to create a twitter application. The python-twitter docs explain the process well. Note that you can ignore callback URL, and you'll want to set the name, description, and website fields to the name, description, and website of your application.

Once you're done that, you may want to turn on "Read, Write and Access direct messages" in the "Permissions" tab. Then generate/regenerate your access token and secret.

Once you're done that, you'll want to set the following values in your settings.py file:

| | |
|---|---|
| `DJANGO_VOX_TWITTER_CONSUMER_KEY` | Consumer Key (API Key) |
| `DJANGO_VOX_TWITTER_CONSUMER_SECRET` | Consumer Secret (API Secret) |
| `DJANGO_VOX_TWITTER_TOKEN_KEY` | Access Token |
| `DJANGO_VOX_TWITTER_TOKEN_SECRET` | Access Token Secret |

**Note:** In order to post a message to your wall, make a site contact with the the twitter protocol and a *blank* address. In order to send a direct message, you'll need a address that equals your user's twitter handle (not including the "@"

prefix).

## 3.6 Webhook (JSON)

**Protocol** `json-webhook`

**Class** `django_vox.backends.json_webhook.Backend`

This backend post JSON-formatted data to webhook. It's useful for implementing generic webhooks or integrating with systems like Huginn or Zapier. The way you specify parameters is the same as with the Postmark backend:

```
parameter_one: {{ content.attribute }}
parameter_two: Hello World
```

This will translate into:

```
{'parameter_one': '<content.attribute>',
 'parameter_two': 'Hello World'}
```

## 3.7 Webhook (Slack)

**Protocol** `slack-webhook`

**Class** `django_vox.backends.slack.Backend`

This backend requires no configuration in django, all of the configuration is essentially part of the addresses used in the protocol. For setting up slack-webhook addresses, see the documentation on *protocols*.

## 3.8 XMPP

**Protocol** `xmpp`

**Class** `django_vox.backends.xmpp.Backend`

**Extra** `[xmpp]`

This backends lets you send messages over xmpp to other xmpp users. It's pretty straightforward; however, it's also pretty slow right now, so don't use it unless your also doing notifications in the background.

To set this up, you need to have the XMPP address and password in your settings. Here's the relevant settings.

| | |
|---|---|
| `DJANGO_VOX_XMPP_JID` | XMPP address |
| `DJANGO_VOX_XMPP_PASSWORD` | Password |

CHAPTER 4

# Protocols and Addresses

A protocol is what lets us sort out and make sense of contact information in django-vox. Classic examples of a "protocol" are things like email, SMS, and XMPP. You can also have proprietary protocols like Slack webhooks, or things that use Google or Apple's push notifications.

Each protocol has it's own kind of addresses. When a contact is sent a message, django-vox automatically selects a backend that matches the available contacts (and addresses by extension) for that protocol.

## 4.1 Activity

ID: `activity`

This protocol is for Activity Streams (and only slightly supported ActivityPub). Message are stored locally in the database and are retrievable from an inbox. Setting this up is a bit of work, see the section on *Activities*.

## 4.2 Email

ID: `email`

The email protocol is really straightforward, the contact's address is just the email address.

## 4.3 SMS

ID: `sms`

The contact's address for SMS is the contact's phone number in E.164 format. It's recommended to use `django-phonenumber-field` if you want to store these numbers in a database.

## 4.4 Twitter

Addresses for the twitter protocol can take two forms:

1. An empty string means the message will get posted as a status update for the account specified in the setting.
2. Anything else will be sent as a direct message to the user with that handle. You shouldn't prefix an '@' to the address and you need to have the correct permissions set in order for this to work.

## 4.5 Webhook Protocols

While webhooks aren't typically a convenient way to contact end-users, they can be pretty useful for setting up site contacts. Because of the templating system, you can be quite flexible in the way you set them up.

| Name | ID | Purpose |
|---|---|---|
| JSON Webhook | `json-webhook` | Generic JSON data (specified in template) |
| Slack Webhook | `slack-webhook` | Posting messages to Slack |

## 4.6 XMPP

XMPP (or Jabber) is a standardized, decentralized chat system. The contact's address should just be their XMPP address (or JID, as it's sometimes called).

CHAPTER 5

# Notification Templates

Notification templates use django's build-in templating system. This gives us more than enough power to craft the kind of message we want, without making things too onerous. Here, we go over the various fields on a template object, and what they do.

## 5.1 Fields

**Backend** This sets the *backend* that the template will use.

**Recipients** Here, you can select one or multiple kinds of recipients. If none are selected, the template won't be used.

**Subject** You can use template variables here but be careful not to make it too long.[1]

**Content** Here's where the body of your message goes. You can use template variables here and there is a toolbar at the top of the text books that has a few tools for convenience. Of particular note is the check mark button (✓) that shows a preview. Use this to check your template.

**Attachments** You can add zero-multiple attachments here. What's available here depends on what's set up in the code[2]

**From address** Normally optional since backends have a way to specify a site-wide default from address if they need one at all. You can use template variables here.[3]

**Bulk** When this is on, only one message will be sent per template to all recipients and the `recipients` parameter will not be available. When it is turned off, a separate message will be made for each recipient, and you can use the `recipients` parameter.

**Enabled** Turning this off will case the template to be ignored.

[1] The subject field only shows if the backend supports it, i.e. it has `USE_SUBJECT` set to `True`.
[2] The attachments field only shows if the backend supports it, i.e. it has `USE_ATTACHMENTS` set to `True`.
[3] The from address field only shows if the backend supports it, i.e. it has `USE_FROM_SUBJECT` set to `True`.

## 5.2 Example Content

The way a template looks shouldn't be too foreign to you if you're already used to django; but just in case you're wondering, here's an example.

```
<p>Hi {{ recipient.name }},</p>

<p>{{ object.poster.name }} has posted a comment on your blog titled
{{ object.article.title }}.</p>
```

Note that the exact variables available will depend on which model the notification is attached to. This example assumes `bulk` is turned off.

## 5.3 Variables

Several variables are provided to you in the template context.

**`object`** This refers to whatever model the notification is attached to. It is visible as the content-type field of the notification when you're editing it in the admin. Most of the time, you're probably going to be wanting to use this.

**`actor`** This is only available if `actor_type` is specified for the notification. It refers to whoever or whatever is causing action associated with the notification.

**`target`** This is only available if `target_model` is specified for the notification. It refers to whoever or whatever the action associated with the notification is affecting.

**`recipient`** The type of this depends on which channel is selected as the recipient of a notification, and what kind of objects that channel returns. In practice, it will probably be some sort of user/user-profile object. When site contacts are the recipient, the value is a `SiteContact` object.

Most of the time, it's recommended to just try and use a field on the `object` variable instead of `target` or `actor`. Sometimes, though, this is just not possible, and you want to be able to differentiate between the two at runtime, so that's why they exist.

## 5.4 Miscellaneous Notes

### 5.4.1 Escaping

Django's template engine has been primarily designed for outputting HTML. The only place in which this really matters is when it comes to escaping content. Plain text and HTML content work fine, however, with other formats like Markdown we need to wrap all the template variables with a custom variable escaping object that escapes everything on the fly. This has a few consequences.

1. Most variables will be wrapped in this class. While the class mostly mimics the behavior of the underlying object, any template filter using `isinstance` will fail.
2. In order to maintain compatibility with template filters, we don't try to escape any of the basic numeric or date/time objects. For the most part this is okay, but it is theoretically possible to end up with a weird result.
3. The result of template filters is typically not escaped correctly.

CHAPTER 6

# Activities

The activities backend provides support for the Activity Streams (and very slightly ActivityPub) standards. Message are stored locally in the database and are retrievable from an inbox. References to notions like actors and inboxes refer to the ideas in that standard.

Setting up the activities is fairly involved, and also entirely optional unless you actually want to use the activity backend. As a result, it's got its own documentation. Note that this backend is somewhat experimental. Don't use it in production unless you've tested it really well and know what you're doing.

## 6.1 Settings

Because we need to generate full uri/iris (including the domain) and we need to be able to do it without an HTTP request, we need to have a way of finding out your domain & scheme. If you're already using `django.contrib.sites` and you have it set up with a `SITE_ID` in your settings, that'll work. Otherwise a simpler solution is to just set these two settings:

| | |
|---|---|
| `SITE_DOMAIN` | Your site's domain name |
| `SITE_SSL` | True if you use HTTPS, False otherwise |

**Caution:** `SITE_SSL` should nearly always be True (the default) unless you're in development testing on localhost.

Also, because this backend isn't enabled by default, you'll need to alter `DJANGO_VOX_BACKENDS` and add `'django_vox.backends.activity.Backend',`. You can see an example on the *Backends* page.

## 6.2 Registering actors

Like we had to register channels before, now we have to register actors too. It's mostly accurate to say that actors should be any users that you want to be able send/receive activities.

Actors all have endpoints, and inboxes (and some unimplemented things). When you add an actor you specify the route for his/her endpoint using a regex, much like you would make a normal Django 1 url route. The parameters in the regex should correspond to the identifying field in the user. Here's an example:

```
actors[User].set_regex(r'^users/(?P<username>[0-9]+)/$')
```

Additionally, you'll also need to return the activity contact from the `get_contacts_for_notification` method for the actors. If you want to send them all the possible notification, then add the following code:

```
def get_contacts_for_notification(self, _notification):
    yield Contact(self.name, 'activity', self.get_object_address())
```

## 6.3 Setting up models

Just like we had to add a bunch of properties to the models for the basic features in django_vox, there's a few more to add to get good results for the activity stream. These aren't strictly necessary, but your results will be pretty plain without them. Code samples of all of these are available in the test site the comes with django_vox.

First, the notification parameters take a new parameter `activity_type`. It can be set to an `aspy.Activity` subclass. If it's not set, django_vox with either match the notification name to an activity type, or use the default 'Create' type.

---

**Note:** This code makes use of the `aspy` library. It's a dependency, so you should get it automatically, just `import aspy`.

---

Second, the object of your activity entries defaults to the plain activity streams "Object" with an id (based on `get_absolute_url()` and a name (based on `__str__()`). This is pretty bare-bones to say the least. You can specify something more colorful by implementing the `__activity__()` method and returning another subclass of `aspy.Object` with perhaps a few properties.

Finally, there's a few rare cases where the same model might need to give you different objects for different kinds of notifications. If you need to do this, you can override `VoxModel.get_activity_object()`.

---

**Note:** If your model is registered with `django_vox.registry.objects`, it's recommended to use `VoxModel.get_object_address()` to get the object's ID, otherwise you can use `django_vox.base.full_iri(self.get_absolute_url())`.

---

## 6.4 Accessing the Inboxes

At this point, you should be able to make up activity notifications, issue them, and then retrieve them using `django_vox.models.InboxItem`. However, if you want to use our hackish ActivityPub half-implementation, there's one/two more steps. First we have to enable the inbox middleware. Add this to your settings.py:

```
MIDDLEWARE = [
    ...
    'django_vox.middleware.activity_inbox_middleware',
]
```

There's still a few things that remain to be documented, like reading inbox items, and adding the ability to perform actions on data in your models by posting to the inbox.

CHAPTER 7

# Running the Demo

The project comes with a demo that can be run from the project directory using:

```
make demo
```

The demo runs with everything in memory, so as soon as you stop it, everything you've done to it will be gone! It uses django's development server, so if you change the source files, the server will reload and any data you had will get reset.

## 7.1 The Jist

The demo is a really basic blog app with the ability for people to "subscribe" by adding their email address and name. Subscribers get emailed, whenever a new article is posted, and the email contains a link that lets them comment. The article's author gets notified whenever comments are posted. Finally, there's also a site contact that gets notified whenever a new subscriber is added.

The demo is set up with django's console email backend, so all the email notifications are printed to the console, where you can see what is going on.

## 7.2 Walkthrough

You can use the demo however you want, but in case you're lost, here's a walkthough you can follow that will show you its features.

1. First, go to the admin site. The url should be `http://127.0.0.1:8000/admin/`. The username is `author@example.org` and the password is `password`. Once in, you can go to "Articles" (under Vox Demo) and add a new one.
2. The demo comes with a subscriber already added, so once you add the article, an email should show up the console. The email contains a link to the new article. Click (or copy-paste) the link to open it in a browser. The loaded page should display the added article and an "add a comment" section. Go ahead and add a comment. After adding a comment you should see another email in the console addressed to the article's author.

3. Additionally, if you go to the blog's index (`http://127.0.0.1:8000/`), you'll see a form to subscribe to the blog. If you add a subscriber, you'll get another email in the console notifying the site contact that somebody subscribed.
4. Finally, you can look though the admin to see how things are set up. You can alter the site contacts or play with the notifications and templates. Bare in mind that while all the backends are enabled, and selectable, the users only have email contacts, so the other backends won't do anything for anything besides site contacts.

CHAPTER 8

# Extras

## 8.1 Background Tasks

Django vox can integrate with django-backgroundtasks if available. Doing so is pretty simple, and (particularly if you have to do database lookups inside `get_contacts_for_notification`) can significantly reduce the work for an initial request.

### 8.1.1 Setup

In order to get this set up, you first need to go install and configure django-backgroundtasks yourself. It's fairly straightforward, but exactly how you want the background tasks run is a question only you can answer.

Once it is set up, replace the following:

```
from django_vox.models import VoxModel
```

with this:

```
from django_vox.extra.background import BackgroundVoxModel \
    as VoxModel
```

### 8.1.2 Troubleshooting

If your messages aren't being sent out, there's a good chance that your background tasks just aren't getting run at all. Try running `manage.py process_tasks` or check your queued background tasks in the django admin.

CHAPTER 9

# Comparison with similar projects

Django Vox is great if you:

- Have content authors who are not programmers
- Use a current versions of Django/Python
- Want to be able to send notifications by multiple protocols
- Don't need "web-scale" (tip: if you're using Django, you don't need "web-scale")
- You don't want to maintain a separate web service

That said here's a more in-depth comparison between Django Vox and some specific alternatives.

- Pinax notifications
  - Upsides
    - Less setup
    - Comes with notification settings editor
    - Optional localization
  - Downsides
    - Templates are part of django's template system, so you can't edit them from the admin.
    - Template parameters must be specified whenever sending a message. You need to know ahead of time, what parameters the template authors might want down the road.
    - Notification recipients must be users
    - Only supports email
    - Doesn't support HTML email
  - Neither
    - Notification types can be created anywhere in the code, this means more flexibility, but potentially more confusion too.

- django-courier
  - Upsides:
    - Uses signals, so you don't have to specify your own notification types
    - No code required
  - Downside
    - Only supports emails
    - Doesn't support HTML email
    - The documentation is very lacking
    - Specifying addresses is convoluted
    - Maybe dead
    - Requires `django.contrib.sites`
- universal_notifications
  - Upsides:
    - Supports many backends
    - Built-in unsubscribing API
  - Downsides:
    - Backends have to know about the structure of the user object
    - Notification recipients must be users
    - Email subjects are hard-coded
    - Templates aren't easily editable in the admin
- django-herald
  - Downsides:
    - Templates are part of django's template system, so you can't edit them from the admin.
    - Notification subjects are stored in the code
    - Only supports emails
    - Notification recipients must be users (though its possible to work around this)
- django-notifier
  - Upsides:
    - Easy setup
  - Downsides:
    - Only supports emails by default
    - Doesn't support HTML email
    - Notification recipients must be users
    - Custom backends must make assumptions about structure of user object
- django-notifications
  - Upsides:

        * Supports more backends
        * Supports filters
    - Downsides:
        * Old, and depends on an long-out-of-date version of django
        * Notification recipients must be users
- django-heythere
    - Downsides:
        * Notification types and templates are stored in `settings.py`
        * Only supports emails
        * Doesn't support HTML email
- django-push-notifications
    - Upsides:
        * Supports push notifications
    - Downsides:
        * Only supports push notifications
        * No templates
        * Notification recipients must be users
- django-sitemessage
    - Upsides:
        * Supports many backends
    - Downsides:
        * Configuration is done in code
        * Not possible to specify different messages for different backends
- django-gcm
    - Downsides:
        * Like django-push-notifications but worse
- django-webpush
    - Downsides:
        * Like django-push-notifications but only supports web push
- django-scarface
    - Downsides:
        * Like django-push-notifications but worse (requires Amazon SNS)

## 9.1 Actually Not Similar Projects

There's also a good number of notification frameworks that solve a seeming-ly similar, but different problem: in-app notifications and activity feeds. These are the sort of things that might be a back-end to Django Vox. They're listed here for completion:

- django-notifications-hq
- Stream Django (getstream.io)
- Stream Framework
- django-notify-x
- Django Messages Extends
- django-stored-messages
- django-user-streams
- django-knocker
- django-subscription
- django-offline-messages
- Django webline Notifications
- django-nyt

Also, of honorable mention is Kawasemi which is more of a logging system than anything else.

CHAPTER 10

# Indices and tables

- api/modules
- genindex
- search