Pinax Documentation

Release 0.7.1

James Tauber and Pinax Team

January 14, 2018

Contents

Pinax is an open-source platform built on the Django Web Framework.

By integrating numerous reusable Django apps to take care of the things that many sites have in common, it lets you focus on what makes your site different.

CHAPTER 1

Introduction

Pinax is an open-source platform built on the Django Web Framework.

By integrating numerous reusable Django apps to take care of the things that many sites have in common, it lets you focus on what makes your site different.

While our initial development was focused around a demo social networking site, Pinax is suitable for a wide variety of websites. We are working on number of editions tailored to intranets, event management, learning management, software project management and more.

1.1 Features

At this stage, there is:

- openid support
- email verification
- password management
- site announcements
- a notification framework
- user-to-user messaging
- friend invitation (both internal and external to the site)
- a basic twitter clone
- oembed support
- gravatar support
- interest groups (called tribes)
- projects with basic task and issue management
- threaded discussions

- wikis with multiple markup support
- blogging
- bookmarks
- tagging
- contact import (from vCard, Google or Yahoo)
- photo management

and much more coming…

1.2 History and Background

You can learn more about the history and motivation for Pinax in an interview with James Tauber on This Week in Django as well as his talk on Pinax at DjangoCon 2008 and talk on Pinax at PyCon 2009.

CHAPTER 2

Installation

This covers installation from a release bundle. For information on installing a development version, see http://pinaxproject.com/docs/dev/contributing.html.

The release bundle has almost everything you need to get Pinax up and running. The only things not included are Python itself, the Python Imaging Library (PIL) and a database such as SQLite (which is included in Python 2.5+).

For more information on installing PIL, see ref-pil.

Note: If you are on Mac OS X, make sure you have the Apple developer tools installed before proceeding with Pinax installation.

2.1 Installing Pinax

Pinax makes use of Python virtual environments (or virtualenvs) to isolate the various packages it uses from the rest of your system. Pinax comes with a script that will create the virtualenv for you and install Django and the various applications and libraries that make up Pinax.

To run this script, extract the release bundle, cd into it and run:

```
$ python scripts/pinax-boot.py <path-to-virtual-env-to-create>
```

This will set up the virtualenv and install everything.

For example, if you wanted to create your environment in a directory parallel to where you extracted the bundle you could run:

```
$ python scripts/pinax-boot.py ../pinax-env
```

If you use virtualenvwrapper (which we recommend), this would become:

```
$ python scripts/pinax-boot.py $WORKON_HOME/pinax-env
```

2.2 Activating the virtualenv

Any time you work on a project involving Pinax, you will want to activate the virtualenv.

This is done with:

```
$ source <path-to-virtual-env-created>/bin/activate
```

or, in our example above using `../pinax-env`:

```
$ source ../pinax-env/bin/activate
```

On Windows you would run:

```
$ ..\pinax-env\Scripts\activate.bat
```

With virtualenvwrapper, this becomes:

```
$ workon pinax-env
```

which you can run from anywhere on your filesystem.

Note that you will develop your Pinax-based project in a directory outside your virtualenv. As long as the virtualenv is active, your project will have access to all of the apps and libraries Pinax provides.

2.3 Starting a new Pinax project

The recommended way to start a new Pinax-based project is to clone one of the existing projects. This is done via the `pinax-admin clone_project` command which you can run once you are in your Pinax virtual-env.

You can get a list of available projects with:

```
(pinax-env)$ pinax-admin clone_project -l
```

This will show you a list of projects that you can base your new project on.

Just as quick demonstration, let's start with the social_project. cd into the directory you'd like to create your new project in and run:

```
(pinax-env)$ pinax-admin clone_project social_project mysite
```

This will create a new Pinax project called 'mysite' in your current working directory.

Note: We recommend you don't clone projects into the `pinax-env` (the virtual environment) directory. This directory is best left for only the environment isolated away from your project. This enables you to:

- version your project separately from your Pinax environment
- blow away / upgrade your Pinax environment / try different Pinax versions, etc without affecting your project

Lastly, let's get it running:

```
(pinax-env)$ cd mysite/
(pinax-env)$ python manage.py syncdb
(pinax-env)$ python manage.py runserver
```

Point your browser at http://localhost:8000/ and you should see the Pinax default homepage!

Note that mail and some notifications are queued rather than delivered immediately. See *Sending Mail and Notices* for details.

2.4 What's next?

Look at our *customization* documentation to learn how you might customize your cloned project. If you are ready to deploy your project check out the *deployment* documentation.

CHAPTER 3

Customization

As more sites are built using Pinax, more best practices will emerge, but for now what we recommend is:

- Always work off a stable release. The most current release is 0.7beta3.
- Use the pinax-admin *clone_project* command.
- Make necessary changes to the `settings.py` and `urls.py` files in your copied directory.
- Change the domain and display name of the Site in the admin interface.
- Develop your custom apps under your new project or anywhere on Python path.
- Develop your own templates under your new project.

3.1 Choosing a Project

Pinax provides several projects to use as a starting point for customization. Depending on your development style, you may prefer one project over the other.

basic_project This project comes with the bare minimum set of applications and templates to get you started. It includes no extra tabs, only the profile and notices tabs are included by default. From here you can add any extra functionality and applications that you would like.

cms_project_company A very simple CMS that lets you set up templates and then edit content, including images, right in the frontend of the site.

The sample media, templates and content including in the project demonstrate a basic company website.

cms_project_holidayhouse A very simple CMS that lets you set up templates and then edit content, including images, right in the frontend of the site.

The sample media, templates and content including in the project demonstrate a basic site for holiday house rentals.

code_project This project demonstrates group functionality and the tasks, wiki and topics apps. It is intended to be the starting point for things like code project management where each code project gets its own wiki, task tracking system and threaded discussions.

intranet_project This project demonstrates a closed site requiring an invitation to join and not exposing any information publicly. It provides a top-level task tracking system, wiki and bookmarks. It is intended to be the starting point of sites like intranets.

private_beta_project This project demonstrates the use of a waiting list and signup codes for sites in private beta. Otherwise it is the same as basic_project.

sample_group_project This project demonstrates group functionality with a barebones group containing no extra content apps as well as two additional group types, tribes and projects, which show different membership approaches and content apps such as topics, wiki, photos and task management.

social_project This project demonstrates a social networking site. It provides profiles, friends, photos, blogs, tribes, wikis, tweets, bookmarks, swaps, locations and user-to-user messaging.

In 0.5 this was called `complete_project`.

3.2 pinax-admin clone_project

Pinax provides you with `pinax-admin`, a command line utility. With `pinax-admin` you can quickly generate a cloned project. For example, if you wanted to clone the basic_project you could simply do the following:

```
(pinax-env)$ pinax-admin clone_project basic_project mysite
```

3.3 Settings You Will (Possibly) Want To Override

Pinax-specific:

- `PINAX_THEME`
- `CONTACT_EMAIL`
- `URCHIN_ID`
- `BBAUTH_APP_ID`
- `BBAUTH_SHARED_SECRET`
- `SITE_NAME`
- `MAILER_PAUSE_SEND`
- `SERVE_MEDIA`
- `ACCOUNT_OPEN_SIGNUP`
- `ACCOUNT_REQUIRED_EMAIL`
- `ACCOUNT_EMAIL_VERIFICATION`
- `EMAIL_CONFIRMATION_DAYS`
- `LOGIN_REDIRECT_URLNAME`

General to Django:

- `DEBUG`
- `TEMPLATE_DEBUG`
- `LOGGING_OUTPUT_ENABLED`

- ADMINS
- MANAGERS
- DATABASE_ENGINE
- DATABASE_NAME
- DATABASE_USER
- DATABASE_PASSWORD
- DATABASE_HOST
- TIME_ZONE
- SECRET_KEY
- DEFAULT_FROM_EMAIL
- SERVER_EMAIL
- SEND_BROKEN_LINK_EMAILS
- EMAIL_HOST
- EMAIL_HOST_USER
- EMAIL_HOST_PASSWORD
- EMAIL_SUBJECT_PREFIX
- LOGIN_URL

3.4 `base.html` versus `site_base.html`

In the sample projects, `templates/base.html` is intended for overall page structure whereas `templates/site_base.html` is intended for adding site-specific content that is to be found on all pages (things like logo, navigation or footers).

If you are writing a theme to be used across multiple sites, you should modify `base.html`, not `site_base.html`. If you want to keep a particular theme but modify content for a specific site, you should modify `site_base.html`.

3.5 Changing Avatar/Gravatar defaults

By default Pinax assigns to users the Gravatar icon and uses the Gravatar icon system. If you want your own personal site avatar default, simply go to the `settings.py` in your project root and add these two lines of code:

```
# avatar controls
AVATAR_DEFAULT_URL =  MEDIA_URL + '<our_custom_avatar.jpg>'
AVATAR_GRAVATAR_BACKUP = False
```

3.6 Adding Tabs

See ref-tabs

CHAPTER 4

Deployment

In short:

- Create a `local_settings.py` alongside `settings.py` for your host-specific settings (like database connection, email, etc).
- Configure mod_wsgi or mod_python.
- Set up `cron` job for mailer and asynchronous notifications.

4.1 Using mod_wsgi

If you are using mod_wsgi, which we recommend, you will need to provide a WSGI script. All projects include a `deploy/` directory which contains this script named `pinax.wsgi`. You may modify this file as it best suits you.

Here is a basic configuration for Apache (assuming you are using Python 2.5):

```
WSGIDaemonProcess mysite-production python-path=/path/to/virtualenvs/pinax-env/lib/
↪python2.5/site-packages
WSGIProcessGroup mysite-production

WSGIScriptAlias / /path/to/project/deploy/pinax.wsgi
<Directory /path/to/project/deploy>
    Order deny,allow
    Allow from all
</Directory>
```

The above configuration will likely need to be modified before use. Most specifically make sure the `python-path` option points to the right Python version. We encourage you to read about WSGIDaemonProcess to learn more about what you can configure.

4.2 Using mod_python

While we highly recommend you use mod_wsgi you may need to use mod_python. In this case we have provided the correct hooks for you to use Pinax. Here is a sample Apache config that you can use:

```
<Location "/">
    SetHandler python-program
    PythonHandler social_project.deploy.modpython
    SetEnv DJANGO_SETTINGS_MODULE social_project.settings
    PythonDebug On
    PythonPath "['/path/to/pinax/projects'] + sys.path"
</Location>
```

Note: It is important to note that you should pay careful attention to the value of `PythonHandler` above. It is *not* using `django.core.handlers.modpython`. It is using a mod_python handler located in your project's `deploy/` directory. The reason why we have our own mod_python handler is because we need to setup the Pinax environment otherwise you will see failing imports.

4.3 Sending Mail and Notices

Both mail messages and (some) notifications are queued for asynchronous delivery. To actually deliver them you need to run:

```
python manage.py send_mail
```

and:

```
python manage.py emit_notices
```

on a frequent, regular basis.

Because failed mail will be deferred, you need an additional, less frequent, run of:

```
python manage.py retry_deferred
```

We recommend setting up some scripts to run these commands within your virtual environment. You can use the following shell script as the basis for each management command:

```
#!/bin/sh

WORKON_HOME=/home/user/virtualenvs
PROJECT_ROOT=/path/to/project

# activate virtual environment
. $WORKON_HOME/pinax-env/bin/activate

cd $PROJECT_ROOT
python manage.py send_mail >> $PROJECT_ROOT/logs/cron_mail.log 2>&1
```

Let's assume the scripts you create from above are stored in `$PROJECT_ROOT/cron`. You can now setup the cron job similar to:

```
* * * * * /path/to/project/cron/send_mail.sh
* * * * * /path/to/project/cron/emit_notices.sh

0,20,40 * * * * /path/to/project/cron/retry_deferred.sh
```

This runs `send_mail` and `emit_notices` every minute and `retry_deferred` every 20 minutes.

4.4 Media files

Pinax makes it very easy to combine all your applications' media files into one single location (see *Media Handling* for details). Serving them more or less comes down again to how you do it with Django itself.

There is an example on how to serve those files with the development server in *Serving static files during development*.

In a production environment you, too, have to merge those files before you can serve them. Regarding actually serving those files then, see Django's deployment documentation for details.

CHAPTER 5

Media Handling

This document explains how Pinax handles media files across external and internal applications and themes.

5.1 Basic media handling

If you want to override default media files, place yours under *<project_name>/media/...* with the same path. For example:

Original file:

```
src/pinax/media/default/pinax/images/logo.png
```

Your file:

```
<project_name>/media/pinax/images/logo.png
```

5.2 Locations of media files

If you want to use Pinax' media handling with your own Django apps, please make sure you put the media files like JavaScript, cascading stylesheets (CSS) and images in the following directory structure:

```
<app_name>/media/<app_name>/(js|img|css)
```

Doubling your <*app_name*> is required to prevent name collision of media files while deploying.

Site specific media files goes to:

```
<project_name>/media/siteExample.js
```

The special static file service view should be able to serve the media files in development.

5.3 build_media management command

The build_media script collects the media files from Pinax and all the installed apps and arranges them under the `<project_name>/site_media/static` folder.

The command:

```
<project_name>/python manage.py build_media --all
```

will collect the media files from Pinax and all the apps and places them in the folder defined in the `STATIC_ROOT` setting.

If you have two apps with the same file and the same relative path it's advised to use the `--interactive` option so the script will prompt you to choose which one to use. This is useful in case you want to overwrite default media files with your custom app for example. Remember to remove the site_media folder before you use this option or the script will prompt you for each file.

Please also refer to the help of the build_media management command by running:

```
<project_name>/python manage.py build_media --help
```

5.4 resolve_media management command

To quickly resolve the full file path of a media file on the filesystem, you can pass its expected URL path(s) to the `resolve_media` management command, e.g.:

```
$ ./manage resolve_media pinax/css/base.css
Resolving css/site_tabs.css:
  /Users/jtauber/virtualenvs/mysite/lib/python2.6/site-packages/Pinax-0.7beta3-py2.6.
↪egg/pinax/media/default/pinax/css/base.css
```

If multiple locations are found which match the given path it will list all of them, sorted by its importance.

5.5 Serving static files during development

Pinax provides the static file serving view `staticfiles.views.serve` to handle the app and theme media as well as other media files found in the `MEDIA_ROOT` directory. Make sure your projects' urls.py contains the following snippet below the rest of the url configuration:

```
from django.conf import settings
if settings.SERVE_MEDIA:
    urlpatterns += patterns('',
        (r'^site_media/', include('staticfiles.urls')),
    )
```

CHAPTER 6

Group support

Group support in Pinax allows you to define any type of group. Pinax comes bundled with two types of groups:

- tribes — used in social_project
- projects — used in code_project

A group app can have any content object associated with it. Pinax includes several apps that are group aware:

- tasks
- photos
- wiki
- topics

The idea is a group aware app has the ability to work with or without a group association. This is done using a nullable generic foreign key. Pinax comes with an app to do much of the work to make this all happen.

6.1 Writing your own group aware domain objects

If you want to write your own domain object that is group aware start an app for it. Follow the guidelines below and you'll be all set.

6.1.1 Models

Let's look at a basic model that stores data for a blog:

```
class Blog(models.Model):

    name = models.CharField(max_length=140)
```

To enable group support for this minimal domain object add in a nullable generic foreign key:

```
from django.db import models

from django.contrib.contenttypes import generic
from django.contrib.contenttypes.models import ContentType


class Blog(models.Model):

    name = models.CharField(max_length=140)

    object_id = models.IntegerField(null=True)
    content_type = models.ForeignKey(ContentType, null=True)
    group = generic.GenericForeignKey("object_id", "content_type")
```

We use a nullable generic foreign key to enable it to be optional and we don't know what group model it will point to.

6.1.2 Views

The views you write for your app need to be aware there may be a group association. This is to ensure you properly work with the right subset of data from your models.

Let's take a look at a view that would be a bit naive:

```
def blog_list(request):

    blogs = Blog.objects.all()

    return render_to_response("blog/blog_list.html", {
        "blogs": blogs,
    }, context_instance=RequestContext(request))
```

Assuming `Blog` is the first model presented above this will work fine. However, once you introduce the generic foriegn key you will potentially be selecting objects that don't belong.

To deal with situation we introduced a `ContentBridge` object. This object is passed to your view from the layer above. Let's see how to work with it:

```
from django.http import Http404
from django.template import RequestContext
from django.shortcuts import render_to_response
from django.core.exceptions import ObjectDoesNotExist


def blog_list(request, group_slug=None, bridge=None):

    if bridge is not None:
        try:
            group = bridge.get_group(group_slug)
        except ObjectDoesNotExist:
            raise Http404
    else:
        group = None

    if group:
        blogs = group.content_objects(Blog)
    else:
        blogs = Blog.objects.all()
    return render_to_response("blog/blog_list.html", {
        "group": group,
        "blogs": blogs,
    }, context_instance=RequestContext(request))
```

Pretty straight-foward code to handle both group and no group association. If you are writing an app that can guarantee group association you can definitely make it simpler.

Checking for user membership

In many cases you might want to check if the authenticated user has membership in the group. To do this:

```
if not request.user.is_authenticated():
    is_member = False
else:
    is_member = group.user_is_member(request.user)
```

6.1.3 URLs

The `urls.py` file of your app will not need anything special. Most of that is handled by Pinax. However, URL reversal needs to be group aware. We have some helpers to help you work with this easily.

Let's say you have the following `urls.py`:

```
from django.conf.urls.defaults import *


urlpatterns = patterns("",
    url(r"^blogs/$", "blog.views.blog_list", name="blog_list"),
    url(r"^blog/(?P<slug>[-\w]+)/$", "blog.views.blog_detail", name="blog_detail"),
)
```

To ensure URLs to `blog_list` are correctly generated you will need to use `reverse` located on the `ContentBridge` object:

```
def some_view_with_redirect(request, bridge=None):
    ...
    return HttpResponseRedirect(bridge.reverse("blog_list", group))
```

The `reverse` method work almost identical to Django's `reverse`. It is essentially a wrapper. To reverse the `blog_detail` URL:

```
blog = Blog.objects.get(pk=1)
bridge.reverse("blog_detail", group, kwargs={"slug": blog.slug})
```

Note: You should be aware that **only** `kwargs` work with the bridge `reverse`. This is significant because URLs with `args` mapping will fail reversal. The reason behind this is because Django does not allow mixing of `args` and `kwargs` when performing URL reversal.

There are some cases when you don't have easy access to the `ContentBridge`. You may only have access to a domain object instance. You can get access to the `ContentBridge` from the instance. For example:

```
blog = Blog.objects.get(pk=1)
blog.content_bridge.reverse(...)
```

URL reversal in templates

In Django you may be familiar with the `{% url %}` templatetag. This is basically a wrapper around `reverse`. We provide a similar tag, but works with our `ContentBridge.reverse`. Here is how you might use it:

```
{% load group_tags %}

<a href="{% groupurl blog_detail group slug=blog.slug %}">{{ blog.name }}</a>
```

The `{% groupurl %}` templatetag will fall back to normal Django URL reversal if the value of the passed in `group` is `None`. This enables the ability to work with no group association.

6.2 Writing your own group app

6.2.1 Hooking up content objects

CHAPTER 7

Settings

Depending on what profile and which apps out of it you're using with your Pinax project you have a collection of options for your `settings.py` at your disposal. This listing only includes those that are supported by the internal applications or are used in the sample project settings (excluding those that are already described by Django's settings reference. If you want to know the available settings for any of the external apps, please refer to its docoumentation.

7.1 ACCOUNT_OPEN_SIGNUP

Applications `pinax.apps.signup_codes`

This setting lets you configure whether the site should allow new users to register accounts if they don't provide any kind of signup code.

7.2 BBAUTH_APP_ID

Applications `pinax.apps.bbauth`

This setting is used to allow auth through Yahoo!'s Browser-Based Authentication service.

7.3 BBAUTH_SHARED_SECRET

Applications `pinax.apps.bbauth`

This setting is used to allow auth through Yahoo!'s Browser-Based Authentication service.

7.4 BEHIND_PROXY

Applications `pinax.apps.blog`

If your site is behind a proxy server, set this setting accordingly so that the users' real IP addresses are stored when they create blog posts. When activated the blog takes the user's IP addresse from `request.META['HTTP_X_FORWARDED_FOR']` instead of `request.META['REMOTE_ADDR']`.

7.5 COMBINED_INBOX_COUNT_SOURCES

Applications `pinax.apps.misc`

With this setting you can specify a list applications that write to the users inbox in order to get the current inbox count for the current user. In fact you don't specify the application but a context processor function which you specify here. If you for example create a project based on the `social_project` template, the `messages`, `notification` and `friends_app` all provide information that affects the inbox count.

Such a context processor should return a dictionary with one or more entries for the additional inbox count.

7.6 EMAIL_CONFIRMATION_DAYS

Applications `emailnotification` (external)

This way you can configure the number of days, confirmation email should be valid. For further details please see the documentation of the application itself.

7.7 FEEDUTIL_SUMMARY_LEN

Applications `feedutil` (external)

Default 150

Number of characters used for summary-attributes in feeds.

7.8 FORCE_LOWERCASE_TAGS

Applications `tagging` (external)

Default `False`

If set to `True` all tags will first be converted to lowercase before they are saved to the database.

7.9 LOGIN_REDIRECT_URLNAME

Applications `pinax.apps.account`

This setting is used by `pinax.apps.account` and allows you to specify the name of a named URL as redirection target. If it is not set, `LOGIN_REDIRECT_URL` will get used instead.

7.10 NOTIFICATION_LANGUAGE_MODULE

Application `notification` (external)

Default `False`

This way you can specify what model holds the language selection of a specific user – e.g. `account.Account`. The model has to have a foreign key to the the user model (`user`) and also provide a `language` field, which is then used by the `notification` application.

7.11 PINAX_ITEMS_PER_FEED

Applications `pinax.apps.blog`

Default `20`

With this option the number of posts that should be served in the feeds generated by Pinax' blogging application can be configured.

7.12 PINAX_ROOT

Application `pinax.apps.staticfiles`

Normally you shouldn't need to change this setting. It's a reference to where Pinax itself is installed so that you can easily re-use for instance templates from that location or work with the original static files like the `build_media` command does. There this settng is used to find the media files of those internal applications used in your project, which are then copied into one central location.

For more on this topic take a look at *Media Handling*.

It is also used by default in project settings for determining a template directory.

7.13 PINAX_THEME

Applications

Default `"default"`

With Pinax your site can have multiple themes available. This option now determines, which one of these should be used. In practice the value of `PINAX_THEME` becomes part of the file-paths the `build_media` command is looking for when trying to combine all your media files into one single location. A small example:

```
src/pinax/media/default/pinax/images/logo.png
```

is a file that is specific to the "default" theme for Pinax while:

```
src/pinax/media/new_hotness/pinax/images/logo.png
```

would only be available in the "new_hotness" theme.

This setting is also used for the core templates that are provided with Pinax by default. The default `settings.py` files provided by Pinax for instance load templates from following locations:

```
TEMPLATE_DIRS = (
    os.path.join(os.path.dirname(__file__), "templates"),
    os.path.join(PINAX_ROOT, "templates", PINAX_THEME),
)
```

Note that this setting only applies to Pinax' core media files and templates and is not used either in the internal nor the external apps by default.

7.14 RESTRUCTUREDTEXT_FILTER_SETTINGS

Applications `pinax.apps.blog`

Default `{}`

Using this option you can pass additional settings as dictionary through the `restructuredtext` template library to the underlying `docutils.core.publish_parts` function.

7.15 SERVE_MEDIA

This option is used in the standard projects' URLconf to determine, if `django.views.static.serve` should be used to serve static files. By default this settings is bound to the `DEBUG` setting in the default `settings.py`.

7.16 STATICFILES_DIRS

Default `[]`

This setting defines the additional locations the `staticfiles` app will traverse when looking for media files, e.g. if you use the *build_media* or *resolve_media* management command or use the *static file serving view*.

It should be defined as a sequence of `(label, path)` tuples, e.g.:

```
STATICFILES_DIRS = (
    ('pinax', os.path.join(PINAX_ROOT, 'media', PINAX_THEME)),
    ('my_project', os.path.join(PROJECT_ROOT, 'media')),
)
```

7.17 STATICFILES_PREPEND_LABEL_APPS

Default `('django.contrib.admin',)`

A sequence of app paths that have the media files in `<app>/media`, not in `<app>/media/<app>`, e.g. `django.contrib.admin`.

7.18 STATICFILES_MEDIA_DIRNAMES

Default `('media',)`

A sequence of directory names to be used when searching for media files in installed apps, e.g. if an app has its media files in `<app>/static` use:

```
STATICFILES_MEDIA_DIRNAMES = (
    'media',
    'static',
)
```

7.19 URCHIN_ID

Applications `pinax.apps.analytics`

Used by `pinax.apps.analytics` as part of your account information on Google Analytics. Based on this setting the JavaScript is generated that is then embedded into your website to allow Google Analytics to track your traffic.

7.20 MARKUP_CHOICES

Applications `pinax.apps.blog`, `pinax.apps.tasks`

Default `(('restructuredtext', u'reStructuredText'), ('textile', u'Textile'), ('markdown', u'Markdown'), ('creole', u'Creole'),)`

The actual origin of this setting is django-wikiapp which is one of the external applications Pinax integrates. `pinax.apps.blog` uses it to determine, how a post's content should be converted from plain text to HTML.

7.21 WIKI_REQUIRES_LOGIN

Applications `wiki` (external)

Default `False`

With this setting you configure the Wiki to be only accessible to people who are logged in.

CHAPTER 8

Dependencies

This documents what apps use what other apps and what external libs.

account uses apps misc, emailconfirmation, friends, profiles, timezones, microblogging

ajax_validation uses library simplejson

announcements uses app notification and library atomformat

authsub uses library gdata

avatar uses library PIL

basic_profiles uses app notification

bbauth uses library ybrowserauth

blog uses apps friends, notification, tagging, threadedcomments and library atomformat

bookmarks uses apps tagging, voting and library atomformat

misc uses apps blog, bookmarks, mailer, tribes, microblogging, voting

django_extensions uses a wide range of libraries depends on command

django_openid uses libraries openid, yadis

djangologging uses library pygments

emailconfirmation uses app mailer

friends uses apps emailconfirmation, mailer, notification and libraries gdata, simplejson, vobject, ybrowserauth

friends_app uses apps account, friends, notification

messages uses app mailer, notification

notification uses app mailer and library atomformat

oembed uses library simplejson

photologue uses app tagging and library PIL

photos uses apps photologue, projects, tagging, tribes

profiles uses apps account, friends, gravatar, notification, photos, timezones, microblogging

projects uses apps friends, notification, photos, tagging, things, threadedcomments, wiki

swaps uses apps notification, tagging, threadedcomments

tag_app uses apps blog, bookmarks, photos, projects, tagging, tribes, wiki

timezones uses library pytz

tribes uses apps friends, notification, photos, tagging, things, threadedcomments, wiki, microblogging

wiki uses apps notification, tagging and libraries atomformat, creoleparser, diff_match_patch, docutils

microblogging uses apps account, notification, tribes and libraries atomformat, twitter

CHAPTER 9

Frequently asked questions

9.1 Does Pinax work on Django 1.1?

Yes. Pinax 0.7 ships with Django 1.0.4 by default. Django 1.1 came too late in our 0.7 release cycle. However, we tested it on Django 1.1 to ensure it works and it works well. To use Django 1.1 in your Pinax project simply follow our *installation* documentation and once you are in the virtual environment run:

```
pip install -U Django==1.1.1
```

This will install Django 1.1.1 over 1.0.4.

9.2 Does Pinax work on Django 1.2?

Yes and no. Our stable release 0.7.X (at 0.7.1) does not support Django 1.2. We will attempt to make 0.7.2 compatible to run on Django 1.2. Installing Django 1.2 over 1.0 that ships with 0.7 will work the same way as seen above. We hope to release 0.7.2 soon. There are no timelines.

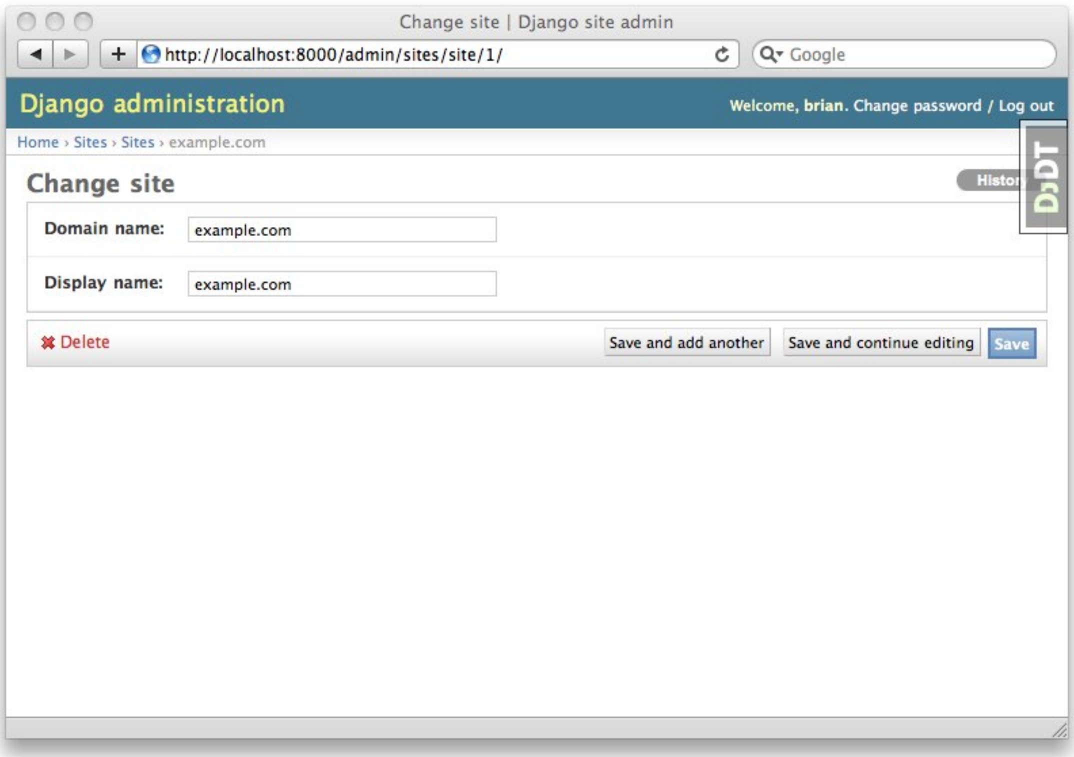
On the other hand our development version of Pinax will ship with Django 1.2. Stay tuned for more news regarding the releases of 0.9.

9.3 How do I change the references to example.com

`example.com` is the default value for `Site.objects.get(pk=settings.SITE_ID).domain`. This comes from the Django contrib app named `sites`. It is enabled in Pinax by default. Pinax uses this value to construct URLs back to your site in e-mails, for example. There are two ways to change this value. First, you can modify it in the shell (using `python manage.py shell`):

```
>>> from django.conf import settings
>>> from django.contrib.sites.models import Site
>>> site = Site.objects.get(pk=settings.SITE_ID)
>>> site.domain = "localhost:8000"
>>> site.name = "Development site"
>>> site.save()
```

Alternatively, you can perform the same action through the admin interface.

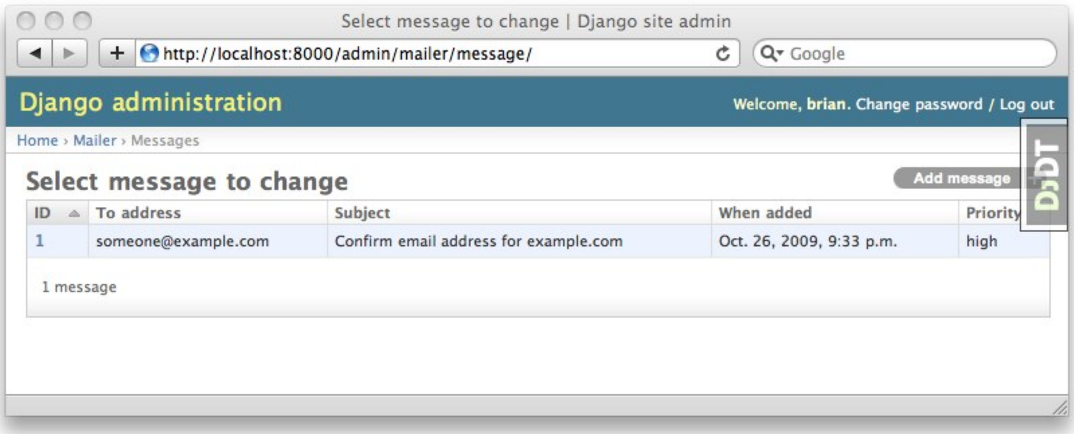

Change site | Django site admin
http://localhost:8000/admin/sites/site/1/
Google
Django administration
Welcome, brian. Change password / Log out
Home › Sites › Sites › example.com
History
DjDT
Delete
Save and add another
Save and continue editing
Save

Change site

Domain name:	example.com
Display name:	example.com

9.4 Why won't my e-mail send?

Pinax queues all e-mail for delivery. This is the behavior of django-mailer. All messages are stored in the database. This enables you to view what will be sent via the admin during development.


Select message to change | Django site admin
http://localhost:8000/admin/mailer/message/
Google
Django administration
Welcome, brian. Change password / Log out
Home › Mailer › Messages
Add message
DjDT

Select message to change

ID	To address	Subject	When added	Priority
1	someone@example.com	Confirm email address for example.com	Oct. 26, 2009, 9:33 p.m.	high

1 message

To send the messages that are queued you should use the send_mail management command. To invoke this you would run:

```
python manage.py send_mail
```

Be sure you have set the appropriate EMAIL_* settings. A full list of these settings can be found in Django settings documentation. Our *deployment* documentation gives instructions on how to set this up on a cron.

Also, some e-mail may occur as a result of notifications. Some notifications are queued. Be sure you run:

```
python manage.py emit_notices
```

to clear the notification queue and get those e-mails queued.

CHAPTER 10

Contributing to Pinax

We are always looking for people wanting to improve Pinax itself. This document outlines the necessary bits to begin contributing to Pinax.

10.1 Getting started

The Pinax source code is hosted on GitHub. This means you must have git installed locally. We recommend you create an account on GitHub allowing you to watch and fork the Pinax source code.

You will want to be sure that your git configuration is set for making commits to a repository. Check the following:

```
git config user.name
git config user.email
```

If the output of any of the two commands above are not entirely correct you can easily correct them:

```
git config --global user.name "First Last"
git config --global user.email "email@somewhere.com"
```

It is critical you set this information up correctly. It helps us identify who you are when you start giving us those awesome patches.

10.1.1 Grabbing the source code

Once you have forked the Pinax source code you can now make a clone of it to your local disk. To do this:

```
git clone git@github.com:<username>/pinax.git
```

This will create new directory named `pinax` which now contains the Pinax source tree ready for you to get started.

10.1.2 Setting up your environment

Now that you've cloned the source code you are ready to get your environment setup to work on Pinax. This section also applies if you are looking to just run off the latest code. We'll assume that your current working directory is from within the clone (the `pinax` directory):

```
python scripts/pinax-boot.py --development --source=. ../pinax-dev
source ../pinax-dev/bin/activate
```

If you use virtualenvwrapper you could alternatively do:

```
python scripts/pinax-boot.py --development --source=. $WORKON_HOME/pinax-dev
workon pinax-dev
```

Finally, you need to install the dependencies for the development version:

```
pip install --requirement requirements/external_apps.txt
```

10.2 Committing code

The great thing about using a distributed versioning control system like git is that everyone becomes a committer. When other people write good patches it makes it very easy to include their fixes/features and give them proper credit for the work.

We recommend that you do all your work on Pinax in a separate branch. When you are ready to work on a bug or a new feature create yourself a new branch. The reason why this is important is you can commit as often you like. When you are ready you can merge in the change. Let's take a look at a common workflow:

```
git checkout -b task-1-work
... do work and git commit often ...
git push origin task-1-work
git checkout -b task-1
git merge --squash --no-commit task-1-work
git commit -m "Fixed #1 -- added a great new feature"
git push origin task-1
```

The reason we have created two new branches is to stay off of `master`. Keeping master clean of only upstream changes makes yours and ours lives easier. You can then send us a pull request for the fix/feature from the "ready" branch. Then we can easily review it and even take a look at the individual commits for why you may have done something. If we say that you've done something slightly wrong you can now go back to the `task-1` branch and correct it. Let's see how we might do this:

```
git checkout -b task-1-work
... fix and git commit often ...
git push
git branch -D task-1
git checkout -b task-1
git merge --squash --no-commit task-1-work
git commit -m "Fixed #1 -- added a great new feature"
git push
```

Send another pull request and we can review the fix.

10.2.1 Writing commit messages

Writing a good commit message makes it simple for us to identify what your commit does from a high-level. We are not too picky, but there are some basic guidelines we'd like to ask you to follow.

```
Fixed #1 -- added some feature
```

We ask that you indicate which task you have fixed (if the commit fixes it) or if you are working something complex you may want or be asked to only commits parts:

```
Refs #1 -- added part one of feature X
```

As said earlier we are not too picky (some core developers may change commit messages before pulling in your changes), but as you get the basics down you make the process of getting your patch into core faster.

Another critical part is that you keep the **first** line as short and sweet as possible. This line is important because when git shows commits and it has limited space or a different formatting option is used the first line becomes all someone might see. If you need to explain why you made this change or explain something in detail use this format:

```
Fixed #13 -- added time travel

You need to be driving 88 miles per hour to generate 1.21 gigawatts of
power to properly use this feature.
```

CHAPTER 11

Release Notes

- current release notes
- all release notes

CHAPTER 12

Documentation for Individual Apps

12.1 External Apps

The majority of functionality in Pinax is provided by external, reusable Django apps.