pinaxtutorial Documentation

Release 0.7.1

Paolo Corti

December 17, 2009

CONTENTS

1 Introduction **3**
- 1.1 Intended audience . . . 3
- 1.2 What is Pinax . . . 4
- 1.3 The Book Store application . . . 5
- 1.4 Tutorial index . . . 7

2 Installing Pinax and making basic customisation **9**
- 2.1 Installing Pinax . . . 9
- 2.2 Creation of the Pinax project . . . 12
- 2.3 Making basic customisation . . . 16

3 Developing the basic application and plugging it in Pinax **19**
- 3.1 Introduction . . . 19
- 3.2 Creating the application . . . 19
- 3.3 Creating the models . . . 20
- 3.4 A note about localization . . . 21
- 3.5 Installing the application in the Pinax project . . . 22
- 3.6 Syncronizing the database . . . 22
- 3.7 Configuring the urls . . . 23
- 3.8 Writing the views . . . 24
- 3.9 Bookstore css and images . . . 27
- 3.10 Plugging the bookstore application into Pinax . . . 28
- 3.11 Writing the templates . . . 29
- 3.12 A quick tour of the basic bookstore application . . . 34
- 3.13 What's next . . . 38
- 3.14 Where can I get the code? . . . 38

4 Internationalization of the application **39**
- 4.1 Using translation strings in Python code and templates . . . 39
- 4.2 Creating and compiling language files . . . 40
- 4.3 Changing settings . . . 42
- 4.4 Notes . . . 43

5 Indices and tables **45**

Index of tutorial:

CHAPTER

ONE

INTRODUCTION

In the last weeks I was studing Pinax , an open source platform for building Django applications. While I enjoyed a lot learning how to develop software with this framework, and I am going to happily use it for a series of projects, I found a bit difficult to get documentation about it, if not reading the source code and the (few - at this time) documentation on the project web site.

When I started, I decided to write a test application for understanding the Pinax philosophy before going for real development projects. I think it can be very useful for other developers new to Pinax to write down my experience. This is why I am going to write this tutorial, hoping it will be really useful for the Django/Pinax community.

While reading my posts, if you find errors or you have suggestions to give, please don't hesitate to put some comment in my blog's posts: I will try to update the tutorial, trying to make it a good resource for people willing to learn to develop with Pinax.

1.1 Intended audience

You may easily follow this tutorial if you have some Python and Django experience. But even if you are new to Python and Django, you may reasonably follow this tutorial picking a bit of Python and Django confidence from the following resources (note, however, that you should have programming experience and you should not be new to web development).

1.1.1 If you are new to Python

According to the Python web site *"Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs. Python runs on Windows, Linux/Unix, Mac OS X, and has been ported to the Java and .NET virtual machines."*

If you are new to Python you can be quickly become productive with it, using these popular resources.

There is really a lot of documentation, I highly recommend this excellent tutorial or the popular Dive Into Python book.

If you want quickly enter Django and then Pinax you may try Instant Python.

1.1.2 If you are new to Django

As written in the Django web site, *"Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design."*.

Many of you will have heard about the Ruby on Rails web framework: we can confidently say that Django is to Python

what Ruby on Rails is to Ruby. Like Ruby on Rails, Django promotes the DRY principles, helping the developers to write web applications in an agile manner. Like Ruby on Rails - but of course in a different fashion - Django provides an object-relational mapper, an elegant URL design system, a powerful template system, an impressive cache system, and other goodness that a modern web framework should provide.

If you want to master Django you should read the excellent documentation.

You may quickly get the a taste of Django, and come back to the Pinax tutorial, using the famous Django's tutorial.

1.2 What is Pinax

According to the Pinax web site: *"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."*

When you are developing a web site, if you choose Django as the web framework you have already a lot of goodness that make your work a lot easier than with other web frameworks.

But many web sites have many common elements, like blogs, wikis, photo galleries, tagging systems and so on and it would be crazy to design and develop them from scratch over and over again.

This is the idea behind the Pinax Project, originally named Django Hot Club like the jazz group founded in France by Django Reinhardt (that gives its name to Django): following the DRY principles, the Pinax Project is a collection of Django application you should use in your web projects for not starting from scratch every time. Pinax try to make you re-use software in the smartest way.

Shortly, these are some of the applications included in Pinax that you could use for your web development:

- **ajax_validation**, a simple application for performing ajax validation of forms created using Django's forms system based on jQuery;
- **announcements**, gives you the ability to manage announcements and to broadcast them into your site (keeping track in session of already displayed ones) or by emailing users;
- **avatar**, allow users to manage avatars for their profile;
- **blog**, a blog application;
- **bookmarks**, let users manage bookmarks;
- **django_openid**, gives to Pinax openid support;
- **django_sorting**, allow easy sorting objects list, and sorting links generation;
- **emailconfirmation**, it's for cases where you don't want to require an email address to signup on your website but you do still want to ask for an email address and be able to confirm it for use in optional parts of your website;
- **flag**, gives the users the possibility to flag contents, for example photo that are considered inappropriate;
- **friends**, let users to ask friendship from other users or from people external to the site;
- **geopy**, a geocoding application for geocoding addresses from Google Maps, Yahoo! Maps, Windows Local Live (Virtual Earth), geocoder.us, GeoNames, MediaWiki pages (with the GIS extension), and Semantic MediaWiki pages. Used from the locations application;
- **gravatar**, for letting the users to use their gravatar as avatar;
- **locations**, gives the user the ability to put geo-locate himself at any time;

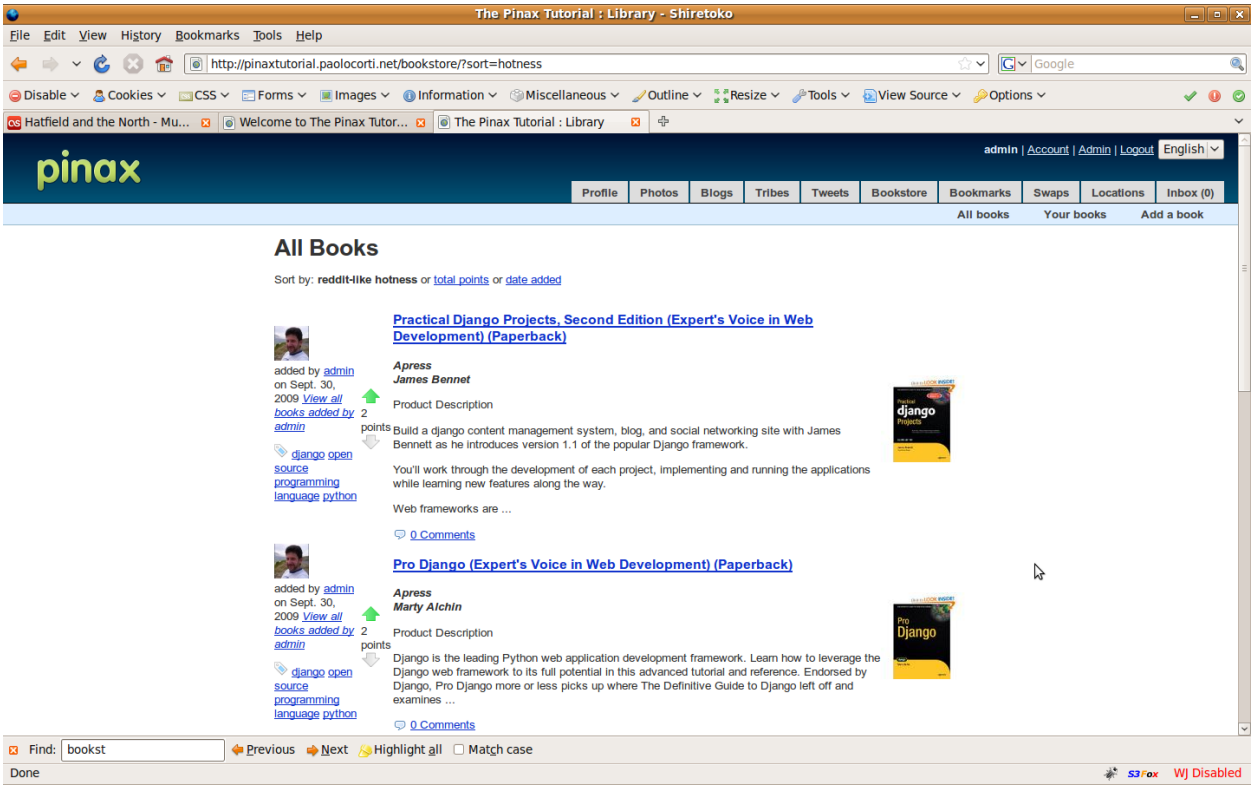
- **mailer**, gives you the ability to queue mail messages and notifications for asynchronous delivery;
- **messages**, gives the user the possibility to send messages to other users. Each user is provided with a message box where messages can be kept or deleted;
- **microblogging**, a twitter clone;
- **notification**, a notification framework;
- **oembed**, gives the site oembed support, for allowing an embedded representation of a URL on third party sites, like Flickr or YouTube;
- **pagination**, let the developer to easily paginate objects list;
- **photologue**, a photo management applications. Users may create photo galleries, and associate them to other contents;
- **projects**, a project application with task and issue management;
- **pygments**, the popular syntax highlighter (to be used, for example, in the wiki and in the blog application);
- **pytz**, for time zone calculations;
- **robots**, for managing web robots access rules to your site;
- **swaps**, for letting a user to swap something with another user;
- **tagging**, let the user to tag any kind of content (ex: a blog post). The user can then browse content by tag;
- **template_utils**, a collection of utilities for text-to-HTML conversion. Supported format are Textile, Markdown, reStructuredText (to be used, for example, in the wiki and in the blog application);
- **threadedcomments**, let the user to write comments on content, even in a threaded way;
- **tribes**, an interest groups management application, where users belonging to that group may take part to topics;
- **uni_form**, provide a simple tag and/or filter that lets you quickly render forms in a div format;
- **vobject**, is a Python package for parsing and generating vCard and vCalendar files, and gives Pinax the ability for example to import contact from GMail or Yahoo;
- **voting**, let the user to vote content in a reddit-like fashion;
- **wiki**, a wiki application for your site;

1.3 The Book Store application

The aim of this tutorial is to teach you Pinax (and Django) with an hands on project. We will build a real application: a book store component for Pinax, following the best practices used from other Pinax projects.

Note that for developing the book store application I was largely inspired by other Pinax applications. In fact you will find in the book store application many similiar elements to the bookmarks application, to the microblogging application, and to other ones. So thanks to the smart developers of these applications!

To get a quick idea of the tutorial final's result, you may have a look at the book store application we will develop: here is a live instance (you need to register to access it).



We want to build a Pinax application to manage the books of a book store (something like in the Amazon fashion).

These are the user stories of the book store application we want to build:

- for every book we want to store the following attributes: title, author, publisher, description, cover art, tags;
- users can browse all of the books;
- books browsing must provide pagination;
- users can add books to the store;
- users can update and delete their books;
- users can vote for a book in a reddit-like fashion;
- users can comment on a book. Comments can be threaded;
- books can be browsed by user and by tag;
- user profile section must provide a book section with every book added by the profile's user;
- users can access RSS feeds of book lists (global list and per user list);
- every book must show which user has added and its avatar (or gravatar if the user has one);
- a user must be notificated (optionally via email) every time a comment is made on a book he has added;
- a user can flag a book added by another user as inappropriate (and an administrator may remove it later);
- the application can be easily localized in different languages to be used in any country.

1.4 Tutorial index

This is the planned tutorial index (there will be a blog post for each paragraph in the next days, as often as i will have the time to post):

- Installing Pinax and making basic customisation
- Create the book store django's application and plugging it into Pinax
- Using pagination, avatars and profiles
- Using the voting application
- Using the tagging application
- Using the feeds application
- Enabling threaded comments for the book store application
- Implementing notifications
- Using the flag application
- Deploying Pinax

Time to start the Pinax Tutorial!

CHAPTER

TWO

INSTALLING PINAX AND MAKING BASIC CUSTOMISATION

I will assume you are installing Pinax v0.7.1 on a Ubuntu 9.04 box, but this procedure - with a few modifications, should work well on every Linux box. For Windows please refer to the Pinax official site or - rather I highly reccomend to use VirtualBox, and to create an Ubuntu 9.04 Virtual Machine, so you will be able to follow step by step this tutorial. There are ready images like this one, to make things even easier.

As suggested from the official installation procedure, the release bundle has everything you need for running Pinax. What is not included is:

- Python;
- Python Imaging Library (PIL);
- SQLite.

Ubuntu 9.04 includes Python 2.6, PIL 1.1.6 and SQLite 3.6.10, so you have everything. If you are using another Ubuntu version or a different Linux distribution, please verify to have all of this software. You will install and use for this tutorial Pinax v0.7.1 (latest release at the date of this post).

2.1 Installing Pinax

2.1.1 Download Pinax

First step for installing Pinax is to download it. First create a directory named virtualenv, then download the tar.gz bundle and extract it in that directory

```
paolo@ubuntu:~/virtualenv$ wget http://downloads.pinaxproject.com/Pinax-0.7.1-bundle.tar.gz
paolo@ubuntu:~/virtualenv$ tar zxf Pinax-0.7.1-bundle.tar.gz
```

2.1.2 Creation of the virtualenv

The best way to try (and to develop with) Pinax is to use virtualenv. If you want to know more about virtualenv take a look at the official documentation. In short words here I can say that what virtualenv does (and does it well!) is creating isolated Python environments.

There are 3 main benefits using virtualenv:

- It is impossible to create different environments with different versions of Django, Pinax or whatever other Python Packages using the Python on the system (everything would be installed in /usr/lib/python2.5/site-packages and you could not install two different versions of the same component)
- You may be interested to install an application and to leave it in that situation forever, without updating any components on which the application is based
- You may not have - for example in a shared host - the privileges to install packages in site-packages directory

Virtualenv solve all this problems, so using it is definitely a good idea. Plus, it includes the excellent Setuptools, and you will be able to use the easy_install command to install Python packages in your virtualenv in a breeze.

Let's see a concrete example: you are going to create a virtualenv with Python 2.5, Django 1.1 and IPython (if you are new to Python you may wonder about what IPython is: a very complete article describing IPython is here).

First you may probably need the virtualenv command, if you have never used. For installing it (on Debian and Ubuntu):

```
(pinax-env)paolo@ubuntu:~/virtualenv$ sudo apt-get install python-virtualenv
```

Now you can create the virtualenv, naming it myenv:

```
(pinax-env)paolo@ubuntu:~/virtualenv$ virtualenv --python=python2.5 myenv
```

If you are on a scratch Ubuntu 9.04 installation, IPython and Django are not available, plus the default Python is version 2.6.2. To prove this just look the following:

```
paolo@ubuntu:~/virtualenv/myenv$ ipython
The program 'ipython' is currently not installed.  To run 'ipython' please ask your administrator to
bash: ipython: command not found
paolo@ubuntu:~/virtualenv/myenv$ python
Python 2.6.2 (release26-maint, Apr 19 2009, 01:56:41)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named django
```

Now activate the virtualenv you have just created. Activating a virtualenv is easily done by typing:

```
(pinax-env)paolo@ubuntu:~/virtualenv$ cd myenv/
(pinax-env)paolo@ubuntu:~/virtualenv/myenv$ source bin/activate
```

Now install django 1.1 and ipython in myenv, by using the easy_install command:

```
(myenv)paolo@ubuntu:~/virtualenv/myenv$ easy_install http://www.djangoproject.com/download/1.1/tarbal
(myenv)paolo@ubuntu:~/virtualenv/myenv$ easy_install ipython
```

Now you are able to use ipython and you can succesfully import Django 1.1:

```
(myenv)paolo@ubuntu:~/virtualenv/myenv$ ipython
Python 2.5.4 (r254:67916, Apr  4 2009, 17:55:16)
Type "copyright", "credits" or "license" for more information.
...
In [1]: import django
In [2]: django.VERSION
Out[2]: (1, 1, 0, 'final', 0)
```

Note that the IPython is now installed, the default Python here is 2.5.4, and the Django version 1.1.0 is available. You can easily check that nothing has been written to the /usr/lib/python2.5/site-packages directory, but everything is under the site-packages directory of the virtualenv:

```
(myenv)paolo@ubuntu:~/virtualenv/myenv/lib/python2.5/site-packages$ ls
Django-1.1-py2.5.egg  easy-install.pth  ipython-0.10-py2.5.egg  setuptools-0.6c9-py2.5.egg  setuptool
```

Now that you understand what a virtualenv is, you are ready to create the virtual environment, and you will name it pinax-env. For doing so you will use a python script included in the Pinax bundle instead than the virtualenv command:

```
paolo@ubuntu:~/virtualenv$ cd Pinax-0.7.1-bundle/
paolo@ubuntu:~/virtualenv/Pinax-0.7.1-bundle$ python scripts/pinax-boot.py ../pinax-env
```

Wait a while while the virtualenv is created, and then activate it:

```
paolo@ubuntu:~/virtualenv/Pinax-0.7.1-bundle$ cd ../pinax-env/
paolo@ubuntu:~/virtualenv/pinax-env$ source bin/activate
```

Note now what is in the virtualenv site-packages directory (tons of useful stuff, this is the Pinax core!):

```
(pinax-env)paolo@ubuntu:~/virtualenv/pinax-env$ ls lib/python2.6/site-packages/
ajax_validation                                 django_markup                                 Pi
announcements                                   django_markup-0.3-py2.6.egg-info              pi
atom                                            django_messages-0.4.2-py2.6.egg-info          py
atomformat.py                                   django_microblogging-0.1.2-py2.6.egg-info     Py
...
```

To be sure that everything is fine, you may try if your virtualenv can import Django and Pinax:

```
In [1]: import django
In [2]: django.VERSION
Out[2]: (1, 0, 3, 'final', 0)
In [3]: import pinax
In [4]: pinax.VERSION
Out[4]: (0, 7, 0, 'final')
```

2.1.3 The Pinax directory structure

Let's spend just a few lines about the Pinax directory structure.

There are two main places where Pinax live: the site-packages directory of your virtualenv and the project directory. You will see later the project directory structure, now I make some considerations about the site-packages directory: as you have seen a few lines above, this directory contains a ton of Python packages that are included in Pinax: you will find there most of the applications and packages we have been talking in the introduction, like for example the bookmarks, the mailer and the swaps applications.

Also, in the site-packages there is the Pinax directory. This is its structure (we show only 2 levels here):

```
paolo@ubuntu:~/virtualenv/pinax-env/lib/python2.6/site-packages/pinax$ tree -d -L 2
.
|-- apps
|   |-- account
|   |-- analytics
|   |-- authsub
|   |-- autocomplete_app
```

```
|   |-- basic_profiles
|   |-- bbauth
|   |-- blog
|   |-- groups
|   |-- photos
|   |-- profiles
|   |-- projects
|   |-- signup_codes
|   |-- tagging_utils
|   |-- tasks
|   |-- threadedcomments_extras
|   |-- topics
|   |-- tribes
|   |-- voting_extras
|   `-- waitinglist
|-- core
|   |-- management
|   `-- serializers
|-- fixtures
|   `-- generate
|-- media
|   `-- default
|-- middleware
|-- projects
|   |-- basic_project
|   |-- cms_project_company
|   |-- cms_project_holidayhouse
|   |-- code_project
|   |-- intranet_project
|   |-- private_beta_project
|   |-- sample_group_project
|   `-- social_project
|-- templates
|   `-- default
|-- templatetags
|   `-- templatetags
`-- utils
```

A few notes: some of the applications, like the blog, the profile and the projects ones, are developed directly in the Pinax packages (at least at this time), and they are in the apps directory.

The media directory contains all of the css and images needed by Pinax and by its applications.

The projects directory contains the template projects you may use to not start from scratch a Pinax project (more in the next paragraphs about this, but basically for starting you will just need to copy one of them).

The templates directory contains the Pinax themes: at this time there is only one available theme (more are coming in the future), called default. Soon we will see how we can customise the templates.

2.2 Creation of the Pinax project

Now it is time to create a Pinax project. Here the way to go is to clone one of the existing Pinax templates projects, using the pinax-admin clone_project, available to you in your virtualenv.

To see a list of available existing projects, use the -l option like this:

```
(pinax-env)paolo@ubuntu:~/virtualenv/pinax-env$ pinax-admin clone_project -l
Available Projects
------------------
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.

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.

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".

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.

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.

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.

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.
```

Take a while for reading about all the templates project that are provided by Pinax. The easiest and minimal way to start a Pinax project would be to start from the basic_project. But here we want to see more Pinax stuff in action, so we will start from the social_project. You will clone the social_project, creating a project named pinaxtutorial:

```
(pinax-env)paolo@ubuntu:~/virtualenv/pinax-env$ pinax-admin clone_project social_project pinaxtutoria
```

Finally you need to sync the project with the database (we are using the default database engine, Sqlite, but if you wish you may change the settings.py database section and choose another database, for example the really excellent Postgres):

```
(pinax-env)paolo@ubuntu:~/virtualenv/pinax-env$ cd pinaxtutorial/
(pinax-env)paolo@ubuntu:~/virtualenv/pinax-env/pinaxtutorial$ chmod 777 manage.py
(pinax-env)paolo@ubuntu:~/virtualenv/pinax-env/pinaxtutorial$ ./manage.py syncdb
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table notification_noticetype
Creating table notification_noticesetting

...

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'paolo'):
E-mail address: pcorti@gmail.com
Password:
Password (again):
Superuser created successfully.
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for notification.NoticeSetting model

...

Installing json fixture 'initial_data' from '/home/paolo/virtualenv/pinax-env/lib/python2.6/site-pack
Installing json fixture 'initial_data' from '/home/paolo/virtualenv/pinax-env/lib/python2.6/site-pack
Installed 18 object(s) from 2 fixture(s)
```

So, using the sync option of the django-admin command, all the tables needed from Pinax and Django have been created. You can easily test this by querying the sqlite_master table (if you are using that engine, of course):

```
(pinax-env)paolo@ubuntu:~/virtualenv/pinax-env/pinaxtutorial$ sqlite3 dev.db
SQLite version 3.6.10
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from sqlite_master;
```

all the tables just created are listed.

Also, when syncing the database for the first time, you are asked to create the superuser for your application. For example, I created a superuser named like me.

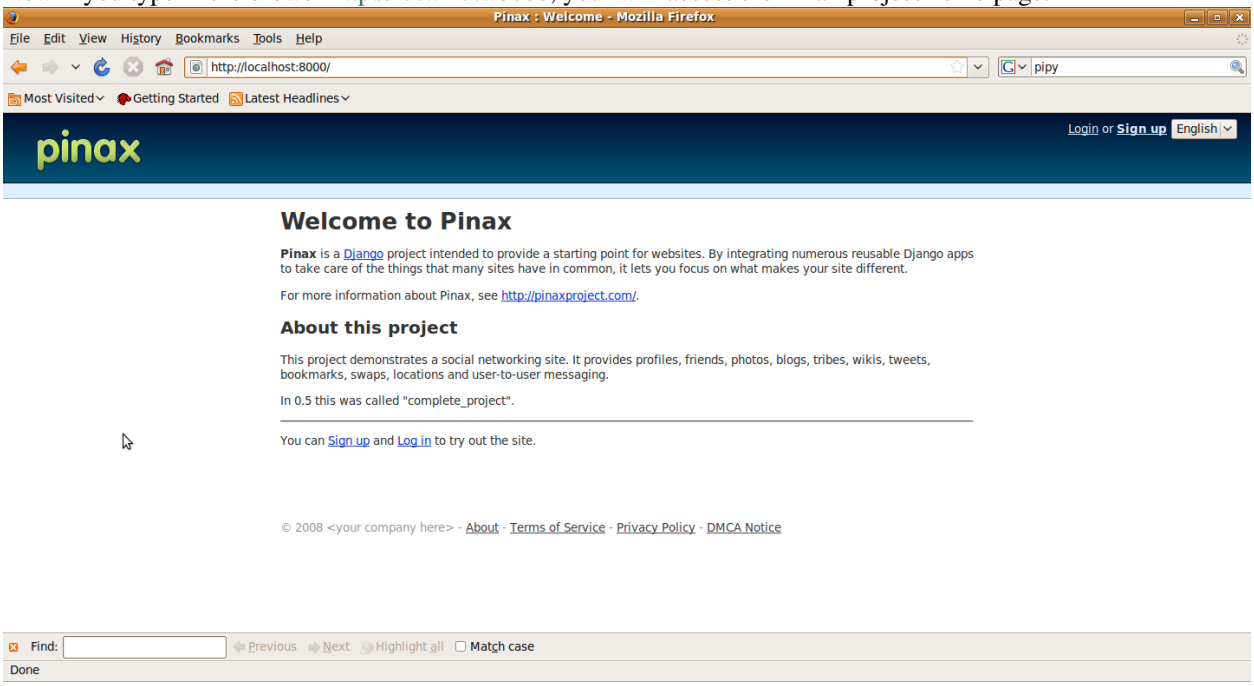
2.2.1 First look at the Pinax project

Now you are ready to start and try the project we just created:

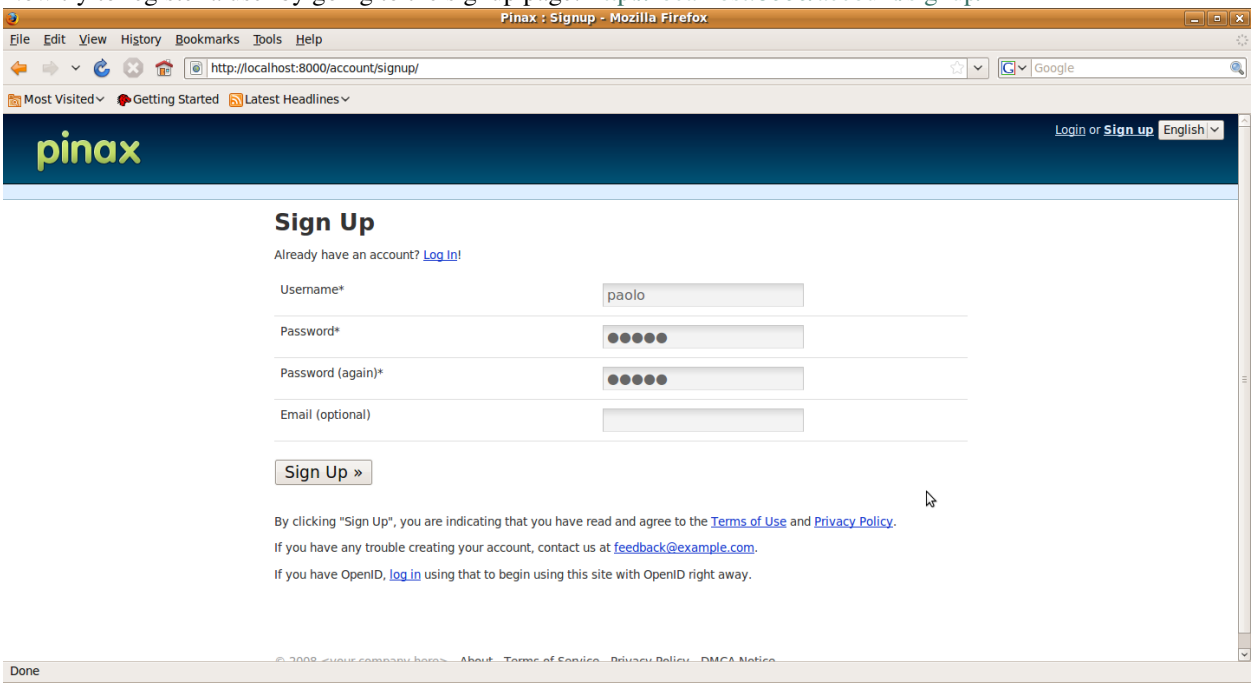
```
(pinax-env)paolo@ubuntu:~/virtualenv/pinax-env/pinaxtutorial$ ./manage.py runserver
Validating models...
0 errors found

Django version 1.0.3, using settings 'pinaxtutorial.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

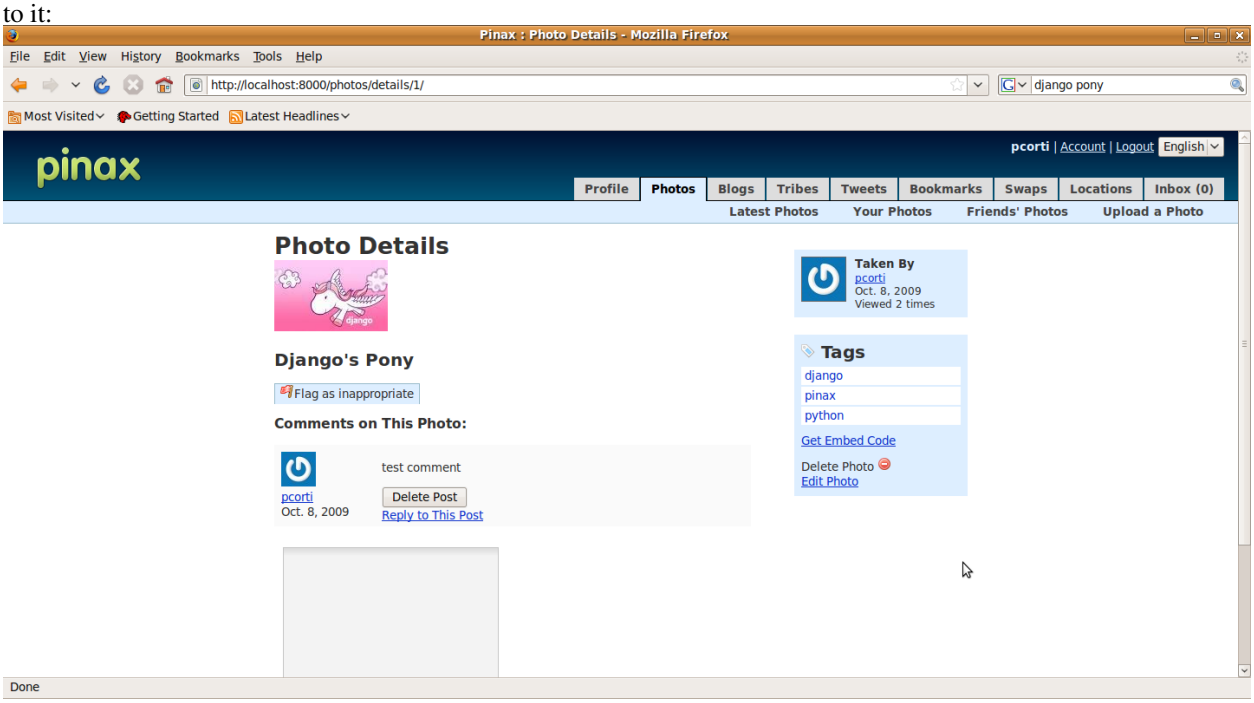
Now if you type in the brower http://localhost:8000, your will access the Pinax project home page:



Now try to register a user by going to the signup page: http://localhost:8000/account/signup/



After registering the user try uploading a photo: http://localhost:8000/photos/upload/ and try to add some test comment to it:



Now take some more time to learn what the pinax social_project offers: there is a ton of stuff already set to be used in your production environment, if you wish. Have fun digging the profiles, blogs, tribes, tweets, bookmarks, swaps and locations sections. Also be delighted by the production-ready messaging system, and by the authentication system.

2.3 Making basic customisation

Now that you are ready with your Pinax environment, you are going to make the last little step at this time: you will learn how easy is in Pinax to make (very) basic customisation.

Keep in mind that, according to the settings.py of the project you created, templates are loaded from two different directories:

```
(settings.py)
TEMPLATE_DIRS = (
        os.path.join(PROJECT_ROOT, "templates"),
        os.path.join(PINAX_ROOT, "templates", PINAX_THEME),
)
```

in my Ubuntu box, the templates directories are in the project directory: ~/virtualenv/pinax-env/pinaxtutorial and in the pinax directory: ~/virtualenv/pinax-env/lib/python2.6/site-packages/pinax.

For the sake of brevity, I will refer to this two directory as PROJECT_ROOT and PINAX_ROOT in the following sections.

Now, for obvious reasons, storing your basic customisation by modifying the templates in the PINAX_ROOT would not be a good choice: as soon as you would decide to update your Pinax installation with a new release of Pinax or from new updates from the trunk if you are living on the edge using the Pinax development version, your html modified templates would be replaced by the new ones and you would miserably loose your customisation.

The best way to go, as with many pythonic things, is to override the templates of the PINAX_ROOT templates directory with the templates of the PROJECT_ROOT templates directory.

Let's see a concrete sample. Suppose you want to change the text of the about page (it is here: http://localhost:8000/about/). According to the central urls.py file:

```
(PROJECT_ROOT/urls.py)
(r'^about/', include('about.urls')),
```

The urls for the about/ views of our project are included in the PROJECT_ROOT/apps/about/urls.py file:

```
urlpatterns = patterns('',
        url(r'^$', direct_to_template, {"template": "about/about.html"}, name="about"),

        url(r'^terms/$', direct_to_template, {"template": "about/terms.html"}, name="terms"),
        url(r'^privacy/$', direct_to_template, {"template": "about/privacy.html"}, name="privacy"),
        url(r'^dmca/$', direct_to_template, {"template": "about/dmca.html"}, name="dmca"),

        url(r'^what_next/$', direct_to_template, {"template": "about/what_next.html"}, name="what_nex
)
```

It is easy to understand that you need to change the about/about.html template for modifying your about page. But wait! there is not the PROJECT_ROOT/templates/about/about.html template. There is, instead, the PINAX_ROOT/templates/default/about/about.html: your project is feeding the about view by that file. According to what we have written a few lines before, you DO NOT change this file. But you override it recreating this file in your PROJECT_ROOT/templates/about directory.

So this is what you have to do: first create the PROJECT_ROOT/templates/about/about.html template file. Then put this few lines in it (I am copying - with little modifications - the PINAX_ROOT/templates/default/about/about.html file):

```
{% extends "site_base.html" %}
{% load i18n %}
{% block head_title %}{% trans "About" %}{% endblock %}
{% block body %}
        {% blocktrans %}
            <p>My new <b>About Page</b></p>
        {% endblocktrans %}
        {% blocktrans %}
            Pinax includes the <a href="http://www.famfamfam.com/lab/icons/silk/">Silk icon set 1.3</
        {% endblocktrans %}
{% endblock %}
```

If you now go to: http://localhost:8000/about/ you will see your new about page, rendered from the about.html template you created in the PROJECT_ROOT/templates/about directory.

CHAPTER

THREE

DEVELOPING THE BASIC APPLICATION AND PLUGGING IT IN PINAX

3.1 Introduction

In this part of the tutorial you are going to create the core of the bookstore application, with all the pages that gives access to the CRUD (Create, Read, Update, Delete) features. And you are goint to plugin this basic application into Pinax.

After finishing with this part you will have the core of the bookstore application working as desired. You will be able to:

- see a list with all the books in the bookstore
- add a new book
- update and delete your books
- see a list of all books added by you
- see a list of all books added by a user

In the following parts you will add some more goodness like localization, pagination, avatars, profiles, voting, tagging, feeds, comments, notifications, and flags for contents, but basically this part is the most important one and it is the way you can create from scratch an application and make it enabled for Pinax.

3.2 Creating the application

Before doing anything, give execute permissions to the manage.py command:

```
chmod 777 manage.py
```

Now create the bookstore application:

```
./manage.py startapp bookstore
```

Now that you have created the application, it it time to add the models, installing it in the Pinax project and syncing the database.

3.3 Creating the models

For creating the models you need to edit the models.py that has been created for you when creating the application. Our application will be composed by a single model, named Book.

PROJECT_ROOT/bookstore/models.py

```
# python
import os
from os import path
from datetime import datetime

# django imports
from django.conf import settings
from django.db import models
from django.contrib.auth.models import User
from django.utils.translation import ugettext_lazy as _

class Book(models.Model):
    """
    Book Model: title, publisher, author, description, coverart, adder, added
    """
    title = models.CharField(_('title'), max_length=255)
    publisher = models.CharField(_('publisher'), max_length=255)
    author = models.CharField(_('author'), max_length=255)
    description = models.TextField(_('description'), blank=True)
    coverart = models.ImageField(upload_to="bookstore", blank=True, null=True)

    adder = models.ForeignKey(User, related_name="added_books", verbose_name=_('adder'))
    added = models.DateTimeField(_('added'), default=datetime.now)

    def get_absolute_url(self):
        return ("describe_book", [self.pk])
    get_absolute_url = models.permalink(get_absolute_url)

    def __unicode__(self):
        return self.title

    class Meta:
        ordering = ('-added', )

    def _get_thumb_url(self, folder, size):
        """ get a thumbnail giver a folder and a size. """
        if not self.coverart:
            return '#'
        upload_to = path.dirname(self.coverart.path)
        tiny = path.join(upload_to, folder, path.basename(self.coverart.path))
        tiny = path.normpath(tiny)
        if not path.exists(tiny):
            import Image
            im = Image.open(self.coverart.path)
            im.thumbnail(size, Image.ANTIALIAS)
            im.save(tiny, 'JPEG')
        return path.join(path.dirname(self.coverart.url), folder, path.basename(self.coverart.path))

    def get_thumb_url(self):
        return self._get_thumb_url('thumb_100_100', (100,100))
```

```
    def thumb(self):
        """ Get thumb <a>. """
        link = self.get_thumb_url()
        if link is None:
            return '<a href="#" target="_blank">NO IMAGE</a>'
        else:
            return '<img src=%s />' % (link)
    thumb.allow_tags = True

    def fullpicture(self):
        """ Get full picture <a>. """
        link = "%s%s" % (settings.MEDIA_URL, self.coverart)
        if link is None:
            return '<a href="#" target="_blank">NO IMAGE</a>'
        else:
            return '<img src=%s />' % (link)
    thumb.allow_tags = True
```

A few notes about the Book's model:

- The basic application of the tutorial implements these fields: title, publisher, author, description, coverart, adder, added
- Also an identifier field, named id, will be created by Django behind the scenes
- The adder field is a foreign key to the user table. This is why you need to import User from django.contrib.auth.models
- The default ordering will be based on the time each book has been added, from the newest to the oldest
- The thumb method returns the url of the thumbnail of the coverart
- The fullpicture method returns the url of the coverart

If you wish that the bookstore application may be managed also from the Django Admin application (this is mainly useful for letting the system administrator to interact with the system), create an admin.py file like this one:

PROJECT_ROOT/bookstore/admin.py

```
from django.contrib import admin
from bookstore.models import Book

class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'description', 'added', 'adder', 'coverart')

admin.site.register(Book, BookAdmin)
```

3.4 A note about localization

One of the aims of this tutorial is also to make the developer confident with the powerful Django's localization features. One of the part of the tutorial will teach you how to develop a localization-ready application.

Meanwhile just be aware that we will make the strings of the application easily translatable by using two mechanism:

- in the models (as you may have noticed) you will use the ugettext_lazy class from django.utils.translation
- in the templates you will use the {% blocktrans %} block

3.5 Installing the application in the Pinax project

Add the Bookstore application in the settings.py file. Do the same with the Admin application if you want to manage the bookstore application also from there:

PROJECT_ROOT/settings.py

```
INSTALLED_APPS = (
    # included
    'django.contrib.auth',
    'django.contrib.contenttypes',
    ...

    # external
    'notification', # must be first
    'django_openid',
    'emailconfirmation',
    ...

    # internal (for now)
    'analytics',
    'profiles',
    'account',
    ...

    'django.contrib.admin', # add this line

    # my apps
    'bookstore', # also add this line
)
```

3.6 Syncronizing the database

Now that you created a new model, you need to syncronize the database:

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

Note that only a new table is generated, and its named bookstore_book. This is the SQL that has been ran by syncdb:

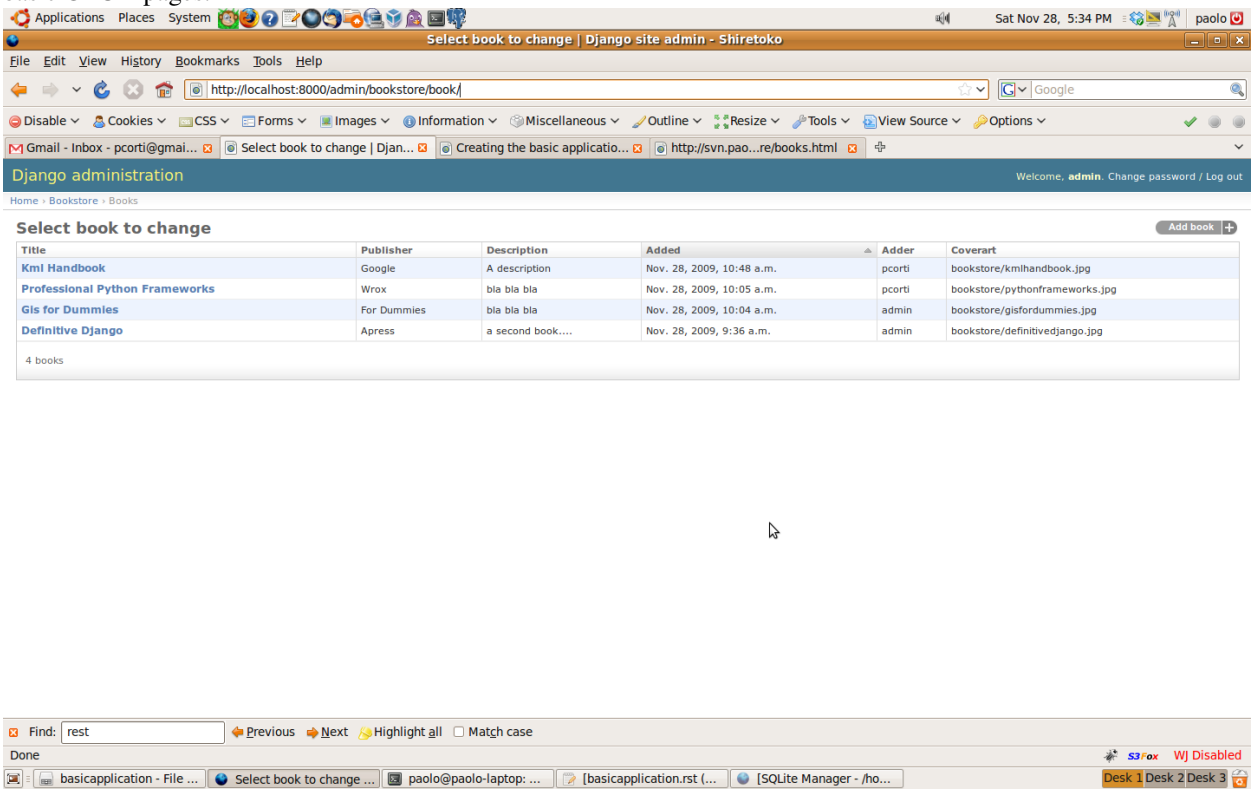
```
CREATE TABLE "bookstore_book" (
    "id" integer NOT NULL PRIMARY KEY,
    "title" varchar(255) NOT NULL,
    "publisher" varchar(255) NOT NULL,
    "author" varchar(255) NOT NULL,
    "description" text NOT NULL,
    "coverart" varchar(100),
    "adder_id" integer NOT NULL REFERENCES "auth_user" ("id"),
    "added" datetime NOT NULL
)
```

Consider that I am using Sqlite, if you are using a different database like Postgres, MySql... the SQL may be slightly different.

Time to run the server and test your new application:

```
./manage.py runserver
```

Now if you access the admin interface, at: http://localhost:8000/admin you will be able to manage your bookstore with basic CRUD pages.



To integrate this CRUD pages into Pinax, you are going to write the views of these pages in the next sections, so keep going with the tutorial.

3.7 Configuring the urls

Configuring the urls means to design how to map urls to pages. Any web developers with good habits should plan the pages composing the application that is developing, and mapping them to corresponding urls.

For the Bookstore application this is what is planned to be:

- A page (the home page of the bookstore application) for getting a list of all books: http://localhost:8000/bookstore/
- A page for getting a list of your books: http://localhost:8000/bookstore/your_books/
- A page for getting a list of the books added by a different users: http://localhost:8000/bookstore/user_books/an_user/
- A page for adding a book: http://localhost:8000/bookstore/add/
- A page for viewing a book, given its identifier: http://localhost:8000/bookstore/4/book/ (it this case the book with id=4)
- A page for updating a book, given its identifier: http://localhost:8000/bookstore/4/update/ (it this case the book with id=4)

Now that we planned the urls needed by the bookstore application, you add the mappings to the views in the urls.py file. Instead than adding these mappings to the central urls.py file of the project, for clearity you create a urls.py file in the bookstore application directory. Now you need to include this directory in the project urls.py file.

This is where to add the file in the urls.py project file:

PROJECT_ROOT/urls.py

```
urlpatterns = patterns('',

    ...

    (r'^feeds/tweets/(.*)/$', 'django.contrib.syndication.views.feed', tweets_feed_dict),
    (r'^feeds/posts/(.*)/$', 'django.contrib.syndication.views.feed', blogs_feed_dict),
    (r'^feeds/bookmarks/(.*)/?$', 'django.contrib.syndication.views.feed', bookmarks_feed_dict),

    # bookstore urls.py file
    (r'^bookstore/', include('bookstore.urls')),

)
```

and this is the urls.py file you need to create in the application directory that translate to django the mappings we have planned before:

PROJECT_ROOT/bookstore/urls.py

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

# from bookstore
from bookstore.models import Book

urlpatterns = patterns('',
        url(r'^$', 'bookstore.views.books', name="all_books"),
        url(r'^(\d+)/book/$', 'bookstore.views.book', name="describe_book"),
        url(r'^your_books/$', 'bookstore.views.your_books', name="your_books"),
        url(r'^user_books/(?P<username>\w+)/$', 'bookstore.views.user_books', name="user_books"),
        # CRUD urls
        url(r'^add/$', 'bookstore.views.add_book', name="add_book"),
        url(r'^(\d+)/update/$', 'bookstore.views.update_book', name="update_book"),
        url(r'^(\d+)/delete/$', 'bookstore.views.delete_book', name="delete_book"),
    )
```

A few notes about the urls:

- each url maps a uri (in a regular expression form) to a Django view. For example in the case of the url named your_books the request made to Django will be managed by a view named your_books
- note that in some case there is a parameter (an integer or a string) in the regular expression: that is for the identifier of a book or for the name of the user. These parameters are passed to the view

3.8 Writing the views

Now that you have planned all the views composing the application, it is time to actually write them!

PROJECT_ROOT/bookstore/views.py

```
#from django
from django.shortcuts import render_to_response, get_object_or_404
from django.template import RequestContext
from django.contrib.auth.decorators import login_required
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _
from django.contrib.auth.models import User

#from bookstore
from bookstore.models import Book
from bookstore.forms import BookForm

def books(request):
    """ Return the all books list, ordered by added date. """
    books = Book.objects.all().order_by("-added")
    return render_to_response("bookstore/books.html", {
        "books": books,
        "list": 'all',
    }, context_instance=RequestContext(request))

def user_books(request, username):
    """ Return an user books list. """
    user = get_object_or_404(User, username=username)
    userbooks = Book.objects.filter(adder=user).order_by("-added")
    return render_to_response("bookstore/books.html", {
        "books": userbooks,
        "list": 'user',
        "username": username,
    }, context_instance=RequestContext(request))

def book(request, book_id):
    """ Return a book given its id. """
    isyours = False
    book = Book.objects.get(id=book_id)
    if request.user == book.adder:
        isyours = True
    return render_to_response("bookstore/book.html", {
        "book": book,
        "isyours": isyours,
    }, context_instance=RequestContext(request))

@login_required
def your_books(request):
    """ Return the logged user books list. """
    yourbooks = Book.objects.filter(adder=request.user).order_by("-added")
    return render_to_response("bookstore/books.html", {
        "books": yourbooks,
        "list": 'yours',
    }, context_instance=RequestContext(request))

@login_required
def add_book(request):
    """ Add a book to the bookstore. """
    # POST request
    if request.method == "POST":
        book_form = BookForm(request.POST, request.FILES)
        if book_form.is_valid():
```

```
            # from ipdb import set_trace; set_trace()
            new_book = book_form.save(commit=False)
            new_book.adder = request.user
            new_book.save()
            request.user.message_set.create(message=_("You have saved book '%(title)s'") %  {'title'
            return HttpResponseRedirect(reverse("bookstore.views.books"))
    # GET request
    else:
        book_form = BookForm()
        return render_to_response("bookstore/add.html", {
            "book_form": book_form,
            }, context_instance=RequestContext(request))
    # generic case
    return render_to_response("bookstore/add.html", {
        "book_form": book_form,
    }, context_instance=RequestContext(request))

@login_required
def update_book(request, book_id):
    """ Update a book given its id. """
    book = Book.objects.get(id=book_id)
    if request.method == "POST":
        book_form = BookForm(request.POST, request.FILES, instance=book)
        book_form.is_update = True
        if request.user == book.adder:
            #from ipdb import set_trace; set_trace()
            if book_form.is_valid():
                book_form.save()
                request.user.message_set.create(message=_("You have updated book '%(title)s'") %  {'t
                return HttpResponseRedirect(reverse("bookstore.views.books"))
    else:
        book_form = BookForm(instance=book)
        return render_to_response("bookstore/update.html", {
            "book_form": book_form,
            "book": book,
            }, context_instance=RequestContext(request))

@login_required
def delete_book(request, book_id):
    """ Delete a book given its id. """
    book = get_object_or_404(Book, id=book_id)
    if request.user == book.adder:
        book.delete()
        request.user.message_set.create(message="Book Deleted")

    return HttpResponseRedirect(reverse("bookstore.views.books"))
```

Some notes on the views.py file you just created:

- some of the views can be accessed without authenticating to the system: books, user_books, book
- some of the views need authentication to the system: your_books, add_book, update_book, delete_book. In this case you need to use the @login_required decorator
- some of the views have additional input parameter to the request. For example add_book, update_book, delete_book need a book_id parameter to know which book to process
- all of the views define a response defined by a template and some variable that the template must process. For

example the user_books renders the response with the books.html template and these variables processed by the response: books, list, username

Note that you also need to create a forms.py file and to add the BookForm that is used from the add_book and update_book views:

PROJECT_ROOT/bookstore/forms.py

```
#from django
from django import forms
from django.utils.translation import ugettext_lazy as _

#from bookstore
from bookstore.models import Book

class BookForm(forms.ModelForm):
    """
    Book Form: form associated to the Book model
    """

    def __init__(self, *args, **kwargs):
        super(BookForm, self).__init__(*args, **kwargs)
        self.is_update = False

    def clean(self):
        """ Do validation stuff. """
        # title is mandatory
        if 'title' not in self.cleaned_data:
            return
        # if a book with that title already exists...
        if not self.is_update:
            if Book.objects.filter(title=self.cleaned_data['title']).count() > 0:
                raise forms.ValidationError(_("There is already this book in the library."))
        return self.cleaned_data

    class Meta:
        model = Book
        fields = ('coverart', 'publisher', 'author', 'description', 'title', 'tags')
```

Here the form is generated by the model (DRY, Don't Repeat Yourself!). Note that the clean method implements some validation stuff. At this time we do not want a book to be created (or updated) if:

- title is empty
- there is already a book with that title

It is now time to write the application templates, but before doing so let's fix the way the Bookstore application css and images are managed. Also, before writing the templates, it is finally time to plugin the Bookstore application into Pinax!

3.9 Bookstore css and images

Create a bookstore directory in the PROJECT_ROOT/media directory. Inside the bookstore directory create a css directory and an img directory. Create a bookstore.css file in the css directory like this one:

PROJECT_ROOT/media/bookstore/css/bookstore.css

```
/* BOOKSTORE */

table.narrow {
    width: 500px;
}
table.bookstore td {
    vertical-align: top;
    padding: 5px;
}
table.bookstore td h2 {
    margin: 0;
    padding: 0;
}
table.bookstore td.vote {
    width: 80px;
    text-align: center;
    vertical-align: middle;
}
.bookstore .even {
    background-color: #FAFAFA;
}
.bookstore .odd {
    background-color: #F3F3F3;
}
div.url {
    color: #666;
    font-size: 90%;
    font-style: italic;
}
```

3.10 Plugging the bookstore application into Pinax

To to so, you just need to modify site_base.html to include the the bookstore.css and the bookstore application (you just need to add two lines):

PROJECT_ROOT/templates/site_base.html:

```
{% block extra_head_base %}
    <link rel="stylesheet" href="{{ STATIC_URL }}css/site_tabs.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/avatar.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/blogs.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/comments.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/friends.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/groups.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/locations.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/messages.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/microblogging.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/pagination.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/photos.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/tabs.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/topics.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/wiki.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}pinax/css/jquery.autocomplete.css" />
    <link rel="stylesheet" href="{{ STATIC_URL }}bookmarks/css/bookmarks.css" />
    <!-- here you insert the booookstore css-->
```

```
    <link rel="stylesheet" href="{{ STATIC_URL }}bookstore/css/bookstore.css" />
    <!-- end -->
    {% block extra_head %}{% endblock %}
{% endblock %}
....
{% block right_tabs %}
    {% if user.is_authenticated %}
        <ul class="tabs">{% spaceless %}
            <li id="tab_profile"><a href="{% url profile_detail user %}">{% trans "Profile" %}</a></l
            <li id="tab_photos"><a href="{% url photos %}">{% trans "Photos" %}</a></li>
            <li id="tab_blogs"><a href="{% url blog_list_all %}">{% trans "Blogs" %}</a></li>
            <li id="tab_tribes"><a href="{% url tribe_list %}">{% trans "Tribes" %}</a></li>
            <li id="tab_tweets"><a href="{% url tweets_you_follow %}">{% trans "Tweets" %}</a></li>
            <li id="tab_bookmarks"><a href="{% url all_bookmarks %}">{% trans "Bookmarks" %}</a></li
            <!-- here you insert the booookstore tab-->
            <li id="tab_bookstore"><a href="{% url all_books %}">{% trans "Bookstore" %}</a></li>
            <!-- end -->
            <li id="tab_swaps"><a href="{% url offer_list_all %}">{% trans "Swaps" %}</a></li>
            <li id="tab_locations"><a href="{% url loc_yours %}">{% trans "Locations" %}</a></li>
            <li id="tab_inbox"><a href="{% url messages_inbox %}">{% trans "Inbox" %} ({{ combined_in
        {% endspaceless %}</ul>
    {% endif %}
{% endblock %}
```

3.11 Writing the templates

3.11.1 Writing the base template

First you create a base template for the Bookstore application. This base template will be extend by all the Bookstore application templates.

PROJECT_ROOT/templates/bookstore/base.html:

```
{% extends "site_base.html" %}

{% load i18n %}

{% block rtab_id %}id="bookstore_tab"{% endblock %}

{% block subnav %}
    <ul>
        <li><a href="{% url all_books %}">{% trans "All books" %}</a></li>
        <li><a href="{% url your_books %}">{% trans "Your books" %}</a></li>
        <li><a href="{% url add_book %}">{% trans "Add a book" %}</a></li>
    </ul>
{% endblock %}
```

Note that this base template extends the site_base.html template you have edited a while ago. As you can easily deduct, the Bookstore application will have 3 sub menus: “All books”, “Your books” and “Add a book”.

3.11.2 Creating a template for listing the books

This is the template that the application will use to render the books, your_books and user_books views. Before writing any template, you need to create the directory that will actually contain them: create a bookstore directory in the templates directory of the project. Also you need to create a PROJECT_ROOT/site_media/media/bookstore/thumb_100_100 directory to manage thumbnails, according to the way the get_thumb_url method is written (you could make things easily customisable with a variable in settings.py).

Now it is time to create the books.html, the template for listing the books:

PROJECT_ROOT/templates/bookstore/books.html template

```
{% extends "bookstore/base.html" %}

{% load i18n %}

{% block head_title %}{% blocktrans %}Library{% endblocktrans %}{% endblock %}

{% block body %}

    <h1>
        {% ifequal list 'all' %}
            {% trans "All Books" %}
        {% endifequal %}
        {% ifequal list 'user' %}
            {{ username }} {% trans "books" %}
        {% endifequal %}
        {% ifequal list 'yours' %}
            {% trans "Your Books" %}
        {% endifequal %}
    </h1>

    <!-- alternate -->
    {% if books %}

        <table class="bookstore">
            {% for book in books %}
                <tr class="{% cycle odd,even %}">
                    <!-- meta -->
                    <td class="meta" >
                        <div class="details">{% blocktrans %}added by{% endblocktrans %} <a href="{%
                        {% blocktrans %}on{% endblocktrans %} {{ book.added|date }}
                        <a href="/bookstore/user_books/{{ book.adder.username }}"><i>{% blocktrans %
                    </td>
                    <!-- book info -->
                    <td>
                        <h3><a href="/bookstore/{{ book.id }}/book/">{{ book.title }}</a></h2>
                        <div class="body">
                            <strong><i>{{ book.publisher }}<br />{{ book.author }}<br /></strong></i>
                            {{ book.description|linebreaks|truncatewords:50 }}
                        </div>
                        {% ifequal list 'yours' %}
                            <table>
                                <tr>
                                    <td>
                                        <!-- udpate book -->
                                        <form method="GET" action="{% url update_book book.id %}">
                                            <input type="submit" value="{% trans "Update Book" %}" />
                                        </form>
```

```
                                    </td>
                                    <td>
                                        <!-- delete book -->
                                        <form method="POST" action="{% url delete_book book.id %}">
                                            <input type="submit" value="{% trans "Delete Book" %}" />
                                        </form>
                                    </td>
                                </tr>
                            </table>
                        {% endifequal %}
                    </td>
                    <!-- cover art -->
                    <td>
                        <div class="coverart" >{{ book.thumb|safe }}</div>
                    </td>
                </tr>
            {% endfor %}
        </table>


    {% else %}
        <p>{% trans "No books yet." %}</p>
    {% endif %}

{% endblock %}
```

Notes about this template:

- the 'list' variable is used to define the title of the page ("All books", "Your books", "An user books")
- the 'books' collection is iterated to create a table with a row for each book (the collection may be related to the whole book set, to your book set or to a user book set
- each row contains the following sections (one for each column): meta, book info and coverart
- the meta section displays the user that added the book (with a link to her/his profile), the date when the book has been added and a link to the full book list added by that user
- the book info section displays the book's title with a link to its page, the publisher, the author and the description (truncated after 50 characters). If the template is rendering the your_books view (thus if 'list' == 'Your books'), a link for updating/deleting that book is displayed
- if the books collection is empty the table is not created an the user is warned with a "No books yet." message

3.11.3 Creating a template for viewing a book

This is maybe the easiest of the templates you need to create. Create the book.html template using the following code:

PROJECT_ROOT/templates/bookstore/book.html template

```
{% extends "bookstore/base.html" %}

{% load i18n %}
{% load uni_form %}

{% block head_title %}{% blocktrans %}Book Description{% endblocktrans %}{% endblock %}
```

```
{% block body %}
    <h1>{{ book.title }}</h1>
        <p>
             <div class="coverart" >{{ book.fullpicture|safe }}</div>
        </p>
        <p>
            <!-- book info -->
            <h3>{{ book.title }}</h2>
            <div class="body">
                <strong><i>{{ book.publisher }}<br />{{ book.author }}<br /></strong></i>
                            {{ book.description|linebreaks }}
            </div>
        </p>
        <p>
            <!-- book action -->
            {% if isyours %}
                <table>
                    <tr>
                        <td>
                            <!-- udpate book -->
                            <form method="GET" action="{% url update_book book.id %}">
                                <input type="submit" value="{% trans "Update Book" %}" />
                            </form>
                        </td>
                        <td>
                            <!-- delete book -->
                            <form method="POST" action="{% url delete_book book.id %}">
                                <input type="submit" value="{% trans "Delete Book" %}" />
                            </form>
                        </td>
                    </tr>
                </table>
            {% endif %}
        </p>
        <p class="meta">
            <!-- meta -->
            <div class="details">{% blocktrans %}added by{% endblocktrans %} <a href="{% url profile_

{% endblock %}
```

It is a bit like the books.html template, but here Django is rendering only one book. You may note that:

- the variable book is used to gain information about the book to render
- the variable isyours is needed to let the user to update or delete the book if she/he is the user who has added it
- there are 3 sections in the book representation: book info, book action, meta
- book info displays the book's cover art, title, publisher, author and description
- book action displays the buttons to update or delete the book if the user is the one who has added it in the system
- meta displays the user that added the book (with a link to his/her profile) and the date when the book has been added

3.11.4 Creating a template for adding a book

Now create the add.html template, for adding new books to the library:

PROJECT_ROOT/templates/bookstore/add.html template

```
{% extends "bookstore/base.html" %}

{% load i18n %}
{% load uni_form %}

{% block head_title %}{% blocktrans %}Add Book{% endblocktrans %}{% endblock %}

{% block body %}
    <h1>{% trans "Add Book" %}</h1>

    <form enctype="multipart/form-data" method="POST" action="" class="uniForm">
        <fieldset class="inlineLabels">
            {{ book_form|as_uni_form }}
            <div class="form_block">
                <input type="submit" value="{% trans 'add book' %}">
            </div>
        </fieldset>
    </form>
{% endblock %}
```

Here there is nothing much to explain, you are rendering the template with a BookForm variable.

3.11.5 Creating a template for updating an existing book

Time to create the update template, for updating a book in the library:

PROJECT_ROOT/templates/bookstore/update.html template

```
{% extends "bookstore/base.html" %}

{% load i18n %}
{% load uni_form %}

{% block head_title %}{% blocktrans %}Add Book{% endblocktrans %}{% endblock %}

{% block body %}
    <h1>{% trans "Update Book: " %}{{ book.title }}</h1>
        <form enctype="multipart/form-data" method="POST" action="" class="uniForm">
            <p align='center'>{{ book.fullpicture|safe }}</p>
            <fieldset class="inlineLabels">
                {{ book_form|as_uni_form }}
                <div class="form_block">
                    <input type="submit" value="{% trans 'update book' %}">
                </div>
            </fieldset>
        </form>
{% endblock %}
```

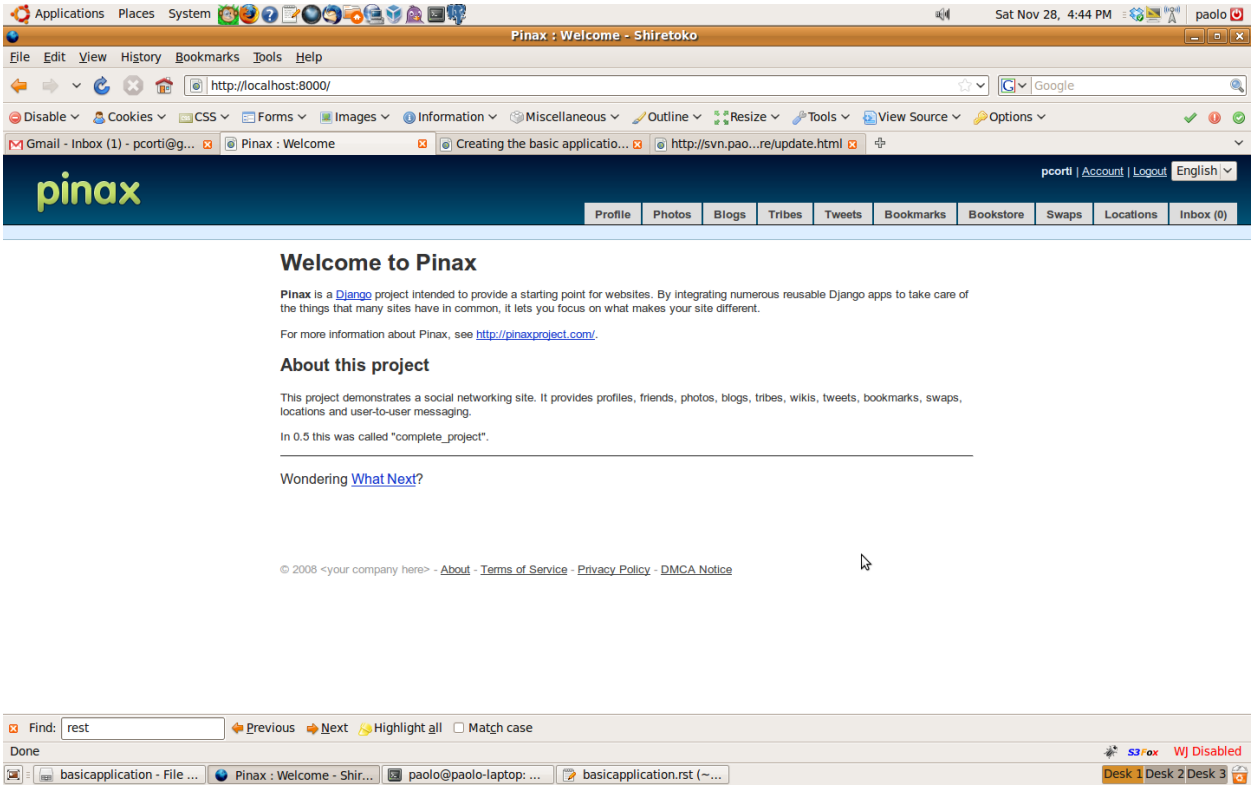
Basically this template is like the add.html ones, with another variable - the book - coming from the view.

3.12 A quick tour of the basic bookstore application

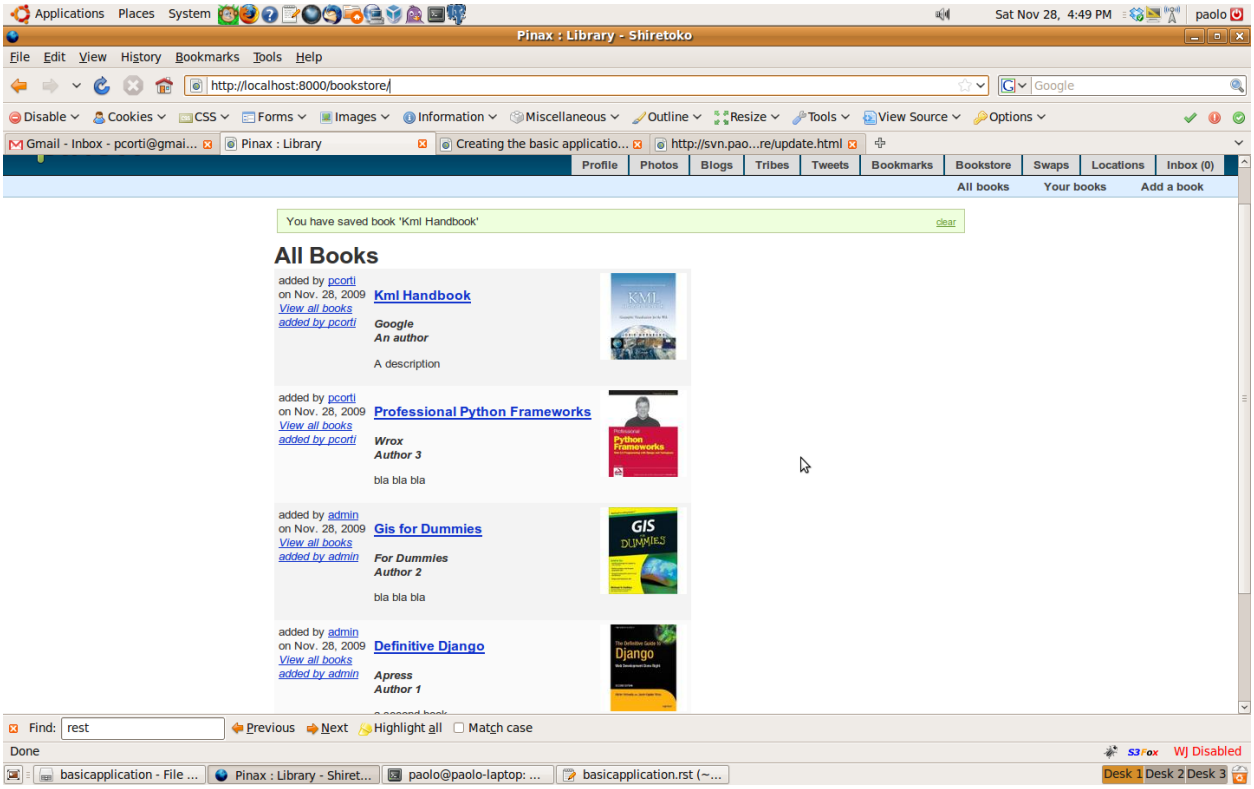
Now that you are over writing the models, the view and the templates of the application, you may finally test if everything is working properly.

If you still didn't do so, start the development server and try visiting the pages and access the features you have implemented.

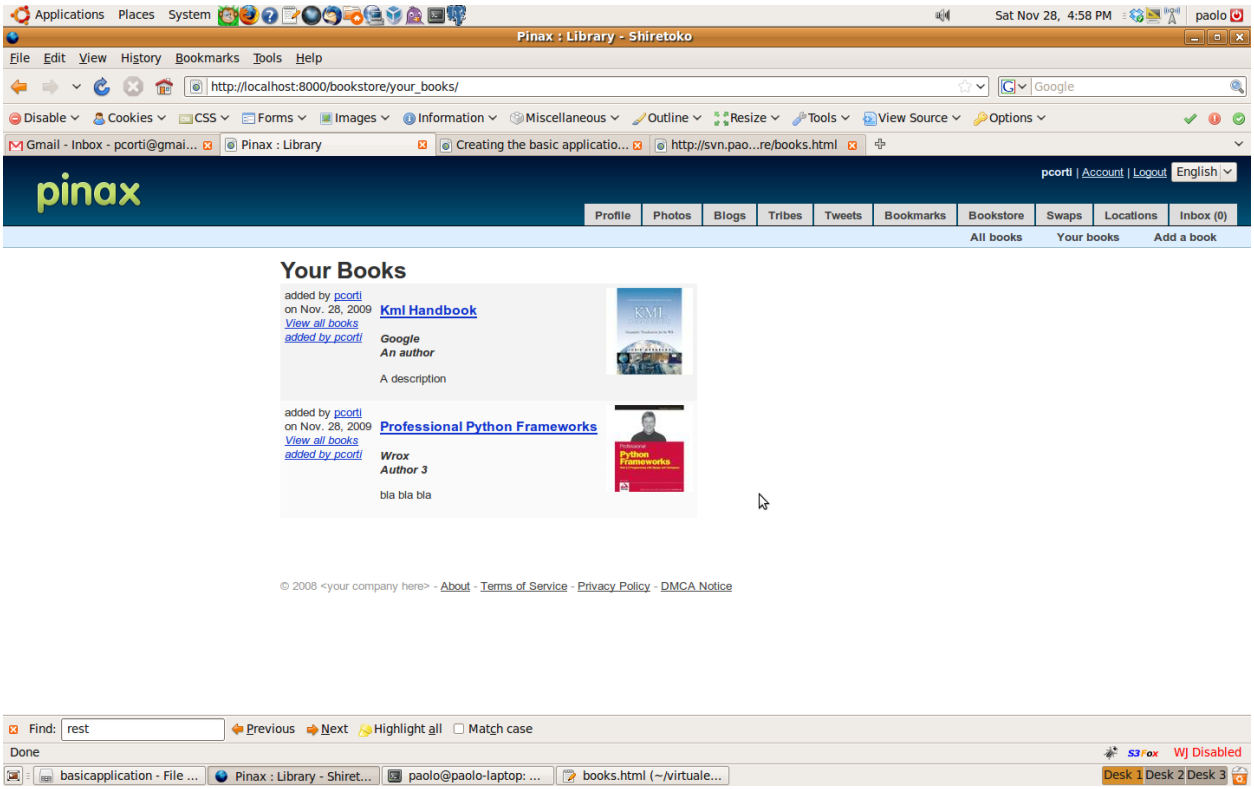
1. Visit the Pinax home page: http://localhost:8000



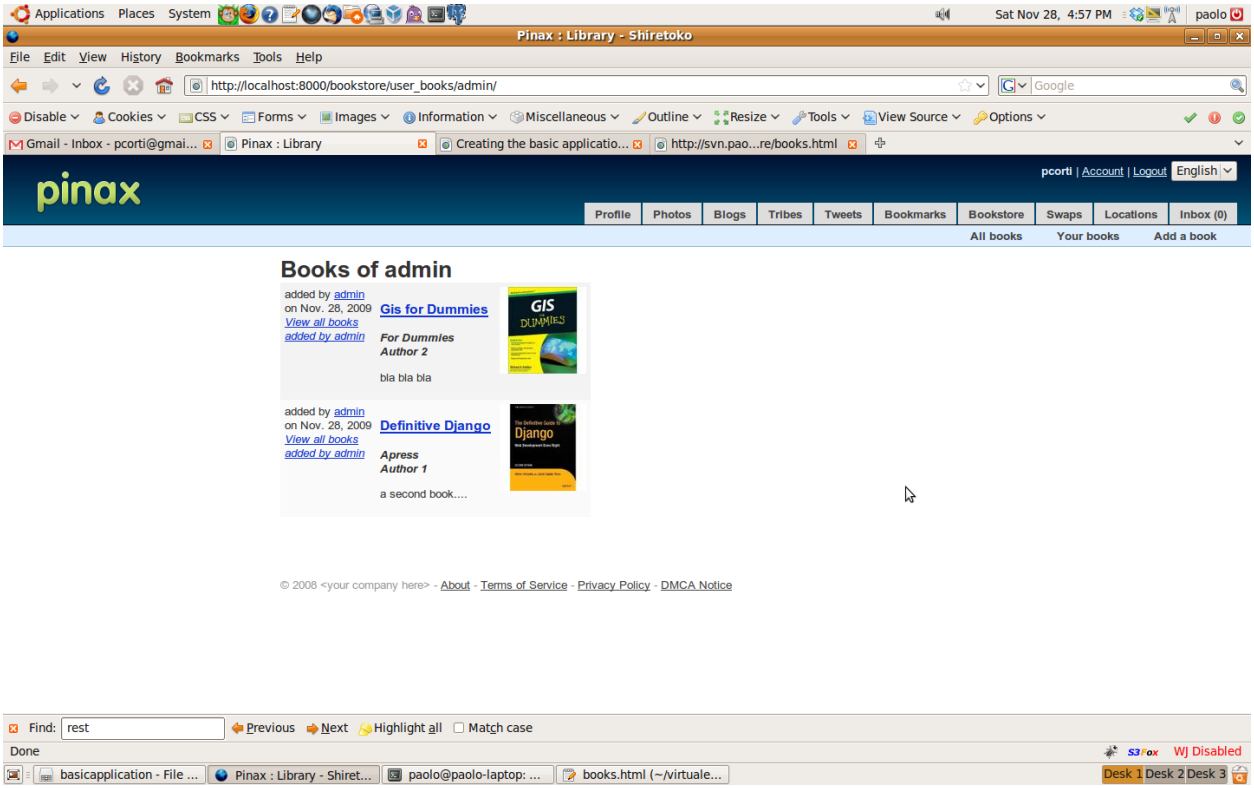
1. Visit the Bookstore all books page: http://localhost:8000/bookstore/



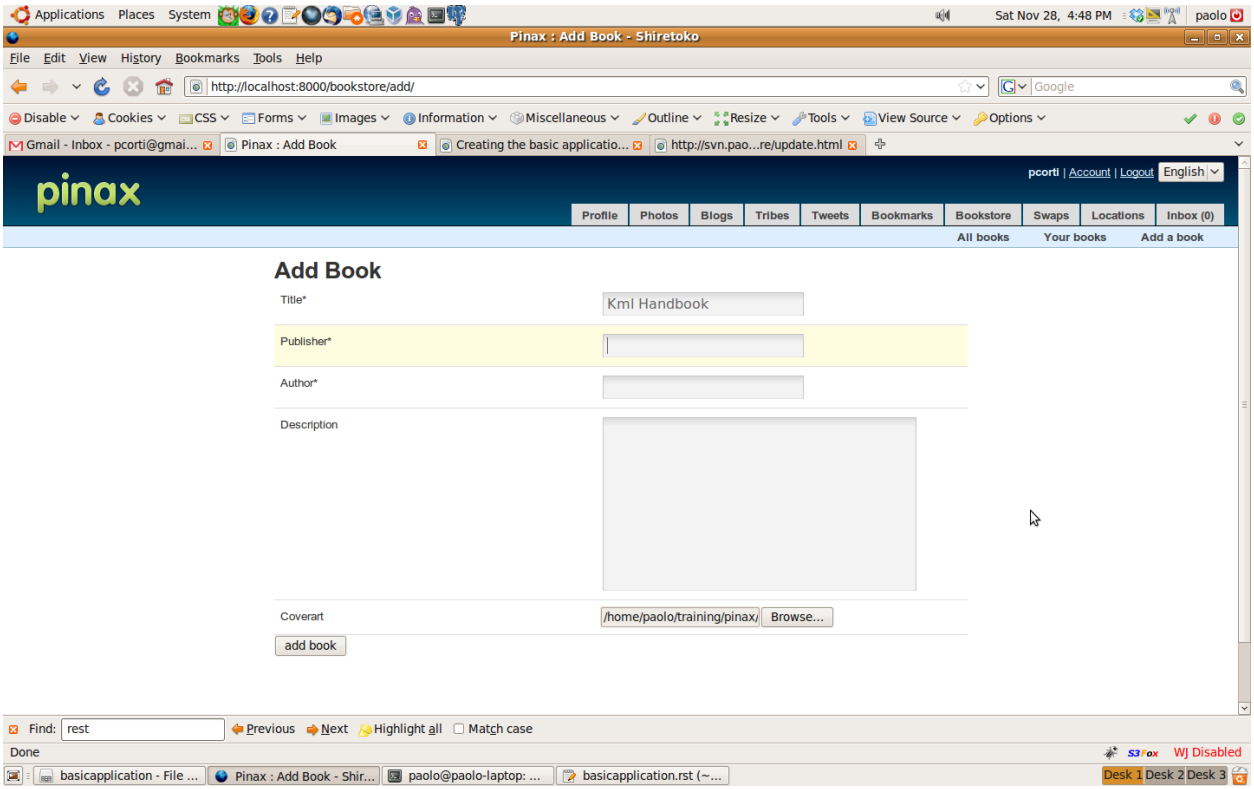
1. Consult your books: http://localhost:8000/bookstore/your_books/



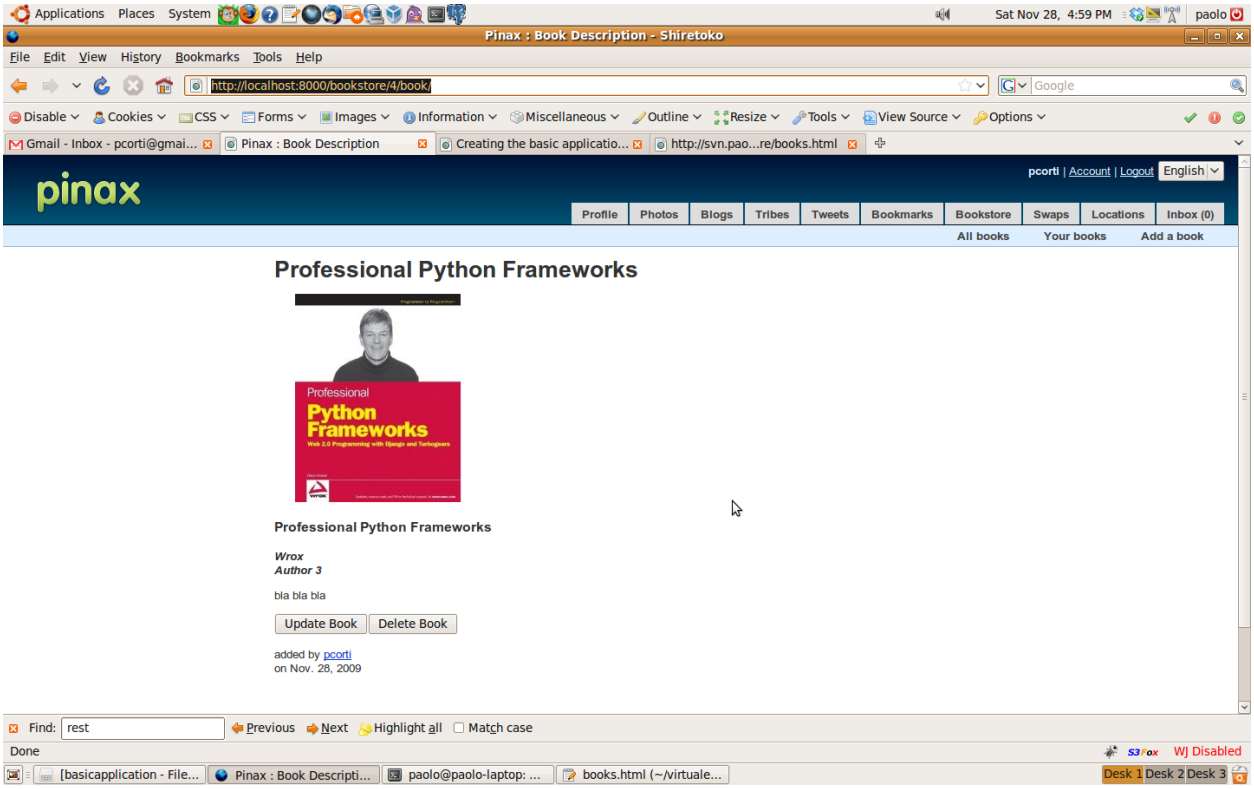
1. Consult the books added by a different users: http://localhost:8000/bookstore/user_books/admin/



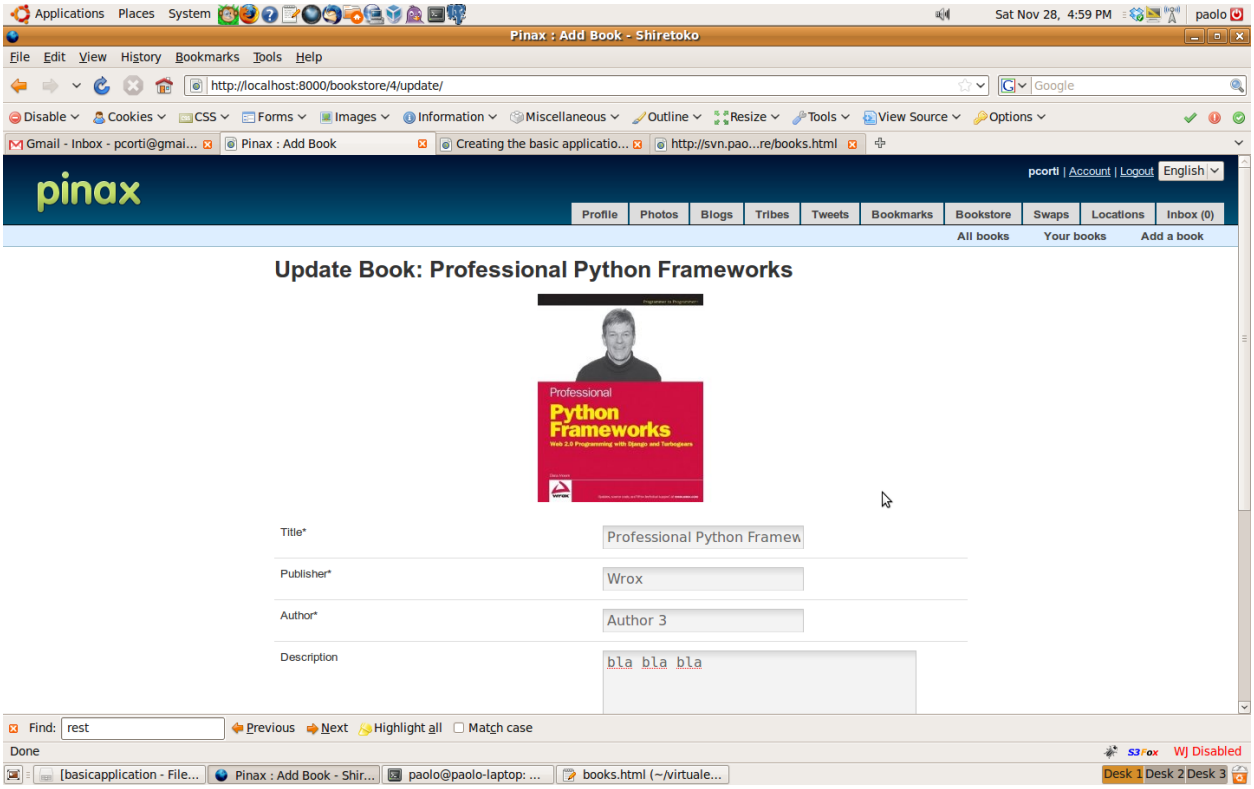
1. Add a book: http://localhost:8000/bookstore/add/



1. View a book: http://localhost:8000/bookstore/4/book/



1. Update a book added by you: http://localhost:8000/bookstore/4/update/



3.13 What's next

Now that you are ready with the basic bookstore application, you may read the next tutorial parts (as soon as they will appear at this blog) in order to implement other features like localization, pagination, avatars, profiles, voting, tagging, feeds, comments, notifications, and flags for contents.

3.14 Where can I get the code?

The whole tutorial is at gitHub: http://github.com/capooti/pinaxtutorial

Code for this part is here: http://github.com/capooti/pinaxtutorial/commits/PinaxTutorial-2

There you can download tutorial's code, and the tutorial text in restructured text format (Sphinx ready).

CHAPTER

FOUR

INTERNATIONALIZATION OF THE APPLICATION

(Note that, if you did the other parts of this tutorial before the date of this post, you need to move the PROJECT_ROOT/templates/bookstore directory in PROJECT_ROOT/bookstore/templates: so now you will have a PROJECT_ROOT/bookstore/templates/bookstore directory. I did so for a better deployment experience, and doing so I am following the Django best practices. If you are reading the pdf version of the tutorial or the reST documentation, then this documentation is already updated with the templates directory in the right place).

In this part of the tutorial I will show how easy it is to enable your application for internationalization with Pinax.

Enabling internationalization for a Pinax application is just the same procedure for enabling it for a Django application.

You can find a complete discussion about Django's internationalization mechanism here Be sure to read this discussion before going on with this tutorial.

To enable internationalization for your application you will need to complete three steps:

- use translation strings in Python code and templates
- create language files
- change settings

Let's see how to apply them to the Bookstore application you are developing.

4.1 Using translation strings in Python code and templates

In the Bookstore application we have already used translation strings in Python code and templates with the help of the django.utils.translation.ugettext_lazy() function and the {% trans %} (and {% blocktrans %}) template tag. For example:

In the model: PROJECT_ROOT/bookstore/models.py:

```
...
from django.utils.translation import ugettext_lazy as _

class Book(models.Model):
        """
        Book Model: title, publisher, author, description, coverart, adder, added
        """
        title = models.CharField(_('title'), max_length=255)
        publisher = models.CharField(_('publisher'), max_length=255)
```

```
        author = models.CharField(_('author'), max_length=255)
        ....
```

In the views: PROJECT_ROOT/bookstore/views.py:

```
...
from django.utils.translation import ugettext_lazy as _
...
@login_required
def add_book(request):
        """ Add a book to the bookstore. """
        # POST request
        if request.method == "POST":
            book_form = BookForm(request.POST, request.FILES)
            if book_form.is_valid():
                new_book = book_form.save(commit=False)
                new_book.adder = request.user
                new_book.save()
                request.user.message_set.create(message=_("You have saved book '%(title)s'") %  {'tit
                return HttpResponseRedirect(reverse("bookstore.views.books"))
        ...
...
```

In the templates: PROJECT_ROOT/bookstore/templates/bookstore/base.html:

```
{% extends "site_base.html" %}

{% load i18n %}

{% block rtab_id %}id="bookstore_tab"{% endblock %}

{% block subnav %}
        <ul>
            <li><a href="{% url all_books %}">{% trans "All books" %}</a></li>
            <li><a href="{% url your_books %}">{% trans "Your books" %}</a></li>
            <li><a href="{% url add_book %}">{% trans "Add a book" %}</a></li>
        </ul>
{% endblock %}
```

Note that I did not use translation strings for everything in the previous chapters, so if you have time try to cover the totality of the code (or download the source code of this tutorial part at gitHub, as suggested at the ending).

4.2 Creating and compiling language files

Once you have tagged your strings for being translated, you need to write the languages files for any language you wish the application needs to be translated.

4.2.1 Language files for the Bookstore application

For example create a message file for Italian by using the django-admin.py makemessages tool. First we are going to create the message file in the application directory to enable internationalization for the Bookstore Application (note that we need to create a locale directory before):

```
(pinax-env)paolo@paolo-laptop:~/virtualenv/pinax-env/pinaxtutorial/bookstore$ django-admin.py makemes
Error: This script should be run from the Django SVN tree or your project or app tree. If you did inc
(pinax-env)paolo@paolo-laptop:~/virtualenv/pinax-env/pinaxtutorial/bookstore$ mkdir locale
(pinax-env)paolo@paolo-laptop:~/virtualenv/pinax-env/pinaxtutorial/bookstore$ django-admin.py makemes
processing language it
```

Open the django.po file that has been created, and change the msgstr for each msgid.

PROJECT_ROOT/bookstore/locale/it/LC_MESSAGES/django.po:

```
...
#: forms.py:25
msgid "There is already this book in the library."
msgstr "Esiste già questo libro nella libreria."

#: models.py:16
msgid "title"
msgstr "titolo"

#: models.py:17
msgid "publisher"
msgstr "editore"
...
```

4.2.2 Language files for the Pinax project

If you didn't get Pinax packaged with the message files of the language you wish, you may need to do the same procedure for your project. To do so, create the message file in the project directory to enable internationalization for this Pinax Project:

```
(pinax-env)paolo@paolo-laptop:~/virtualenv/pinax-env/pinaxtutorial$ mkdir locale
(pinax-env)paolo@paolo-laptop:~/virtualenv/pinax-env/pinaxtutorial$ django-admin.py makemessages -l i
processing language it
```

Edit the msgstr for each msgid also for this file: PROJECT_ROOT/locale/it/LC_MESSAGES/django.po:

```
#: templates/site_base.html:59
msgid "Bookmarks"
msgstr "Segnalibri"

#: templates/site_base.html:61
msgid "Bookstore"
msgstr "Libreria"

#: templates/site_base.html:63
msgid "Swaps"
msgstr "Scambi"

#: templates/site_base.html:64
msgid "Locations"
msgstr "Località"
```

4.2.3 Compiling the language files

Now you need to compile the messages you modified in django.po by using the django-admin.py tool with the compilemessages option:

```
(pinax-env)paolo@paolo-laptop:~/virtualenv/pinax-env/pinaxtutorial/bookstore$ django-admin.py compile
processing file django.po in /home/paolo/git/pinaxtutorial/bookstore/locale/it/LC_MESSAGES
(pinax-env)paolo@paolo-laptop:~/virtualenv/pinax-env/pinaxtutorial$ django-admin.py compilemessages
processing file django.po in /home/paolo/virtualenv/pinax-env/pinaxtutorial/locale/it/LC_MESSAGES
```

If you examinate both the LC_MESSAGES directory you should find a binary file named django.mo.

Remember to restart the development server every time you compile the messages, so do it now.

4.3 Changing settings

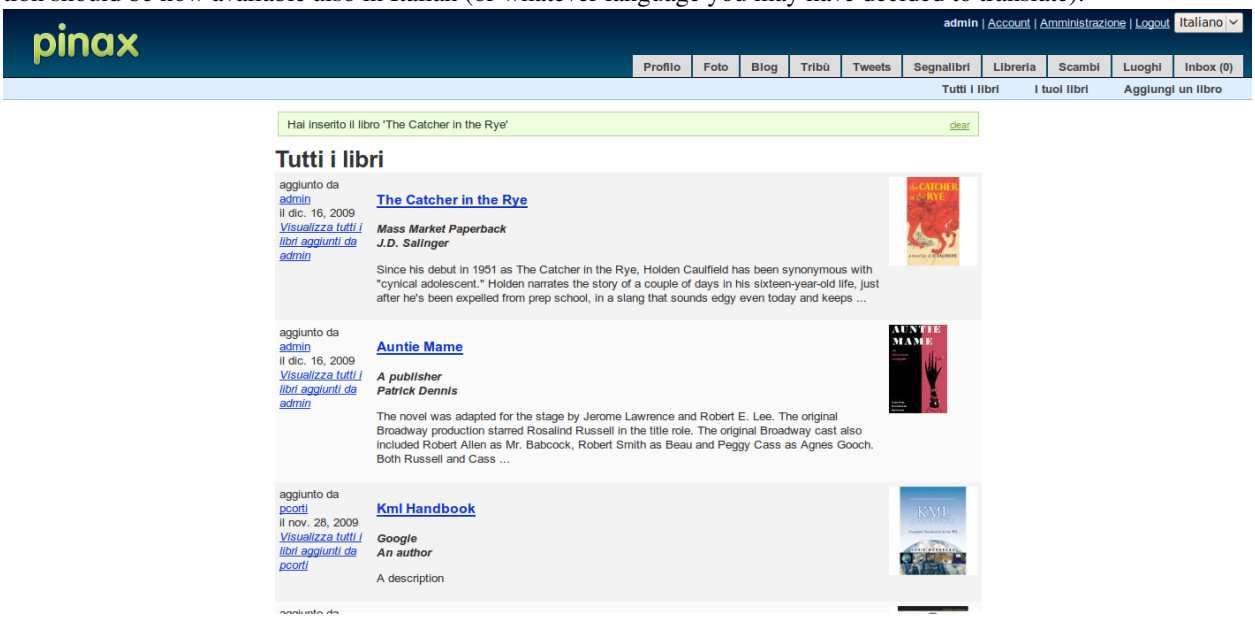
If now you try the Bookstore application, you will realize that your translations are still not there.

This is because the bookstore application has now the Italian translation, but you need to make Django aware of the languages supported by the application.

Open your settings.py file and make sure you have the following settings:

```
...
# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
USE_I18N = True
...
ugettext = lambda s: s
LANGUAGES = (
        ('en', u'English'),
        ('it', u'Italiano'),
)
```

Now you can try your project (change the Pinax language by using the dropdown list in the upper right). The application should be now available also in Italian (or whatever language you may have decided to translate):



4.4 Notes

As usual I have updated the gitHub repository for this project (the tutorial) with all the stuff you many need to go along with it:

- You can find the code of this part of the tutorial here
- You can find updated documentation in reST format here
- You can download a pdf copy of this tutorial here

CHAPTER

FIVE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*