Django web application development tutorial

Django web application development tutorial 1

The goal of this tutorial 2

Introduction to Django 4

Django is a high-level Python Web framework... 4

...that encourages rapid development... 5

...and clean, pragmatic design. 5

Django philosophy 5

Preparing the development environment 7

Step 1: Install the python 7

Step 2: Install the virtualenv 7

Step 3, Using virtualenv to build our django development environment 8

Step 4: Using django-admin to build the a project 9

Django Admin 11

Build the first app: basic request & response 11

Add "apps" package directory in "webapp" 14

Define the list method 17

Define the view method 19

Define the delete method 20

Models, Views and Templates 21

Forms 23

Define the “create” method 23

Define “update” method 27

Static and media files 28

User authentication 28

# The goal of this tutorial

Recently, I developed a django web app named "[iBookmark](http://www.ibookmark.me/) (http://www.ibookmark.me)", which can help users to manage their bookmark online, just like google and delicious bookmark. Users can import and export their bookmarks from browsers and google, delicious.

This simple but useful web app screen shot as below.

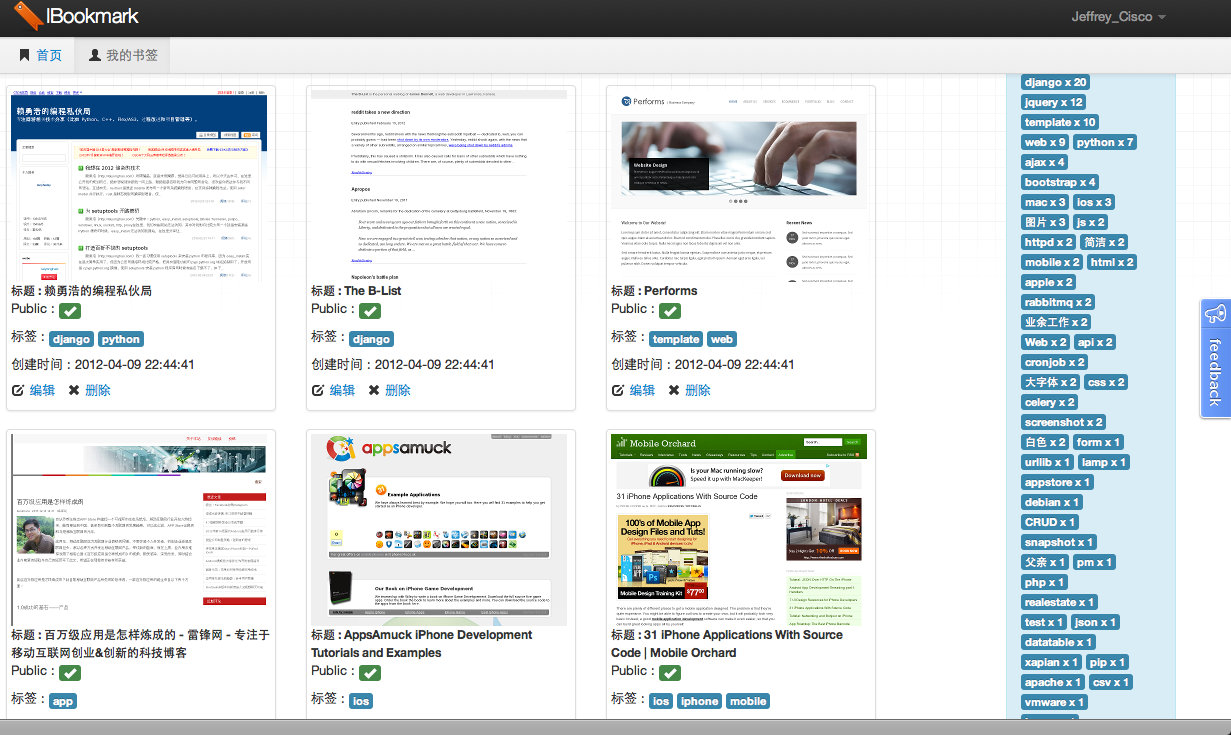


Diagram The snapshot of the http://ibookmark.me

I developed this web app for 1-2 weeks with my spare time. I used many skills as following:

1. Python
2. Django
3. Many django third part apps: django-tagging, pinax, django-oauth, etc.
4. Restful web api with django-piston
5. Ajax
6. jQuery
7. Twitter bootstrap front framework
8. Amazon ec2 server and apache httpd for deployment
9. My own scaffold django\_blank\_project
10. mysql
11. gitlite

I learned those skills of development and deployment from django document, books and google. I decide to record all skills in several blog posts and make them as a tutorial. The tutorial goal is how to use these skills to build a popular web app. I'll use the "iBookmark" as the example for this tutorial.

# Introduction to Django

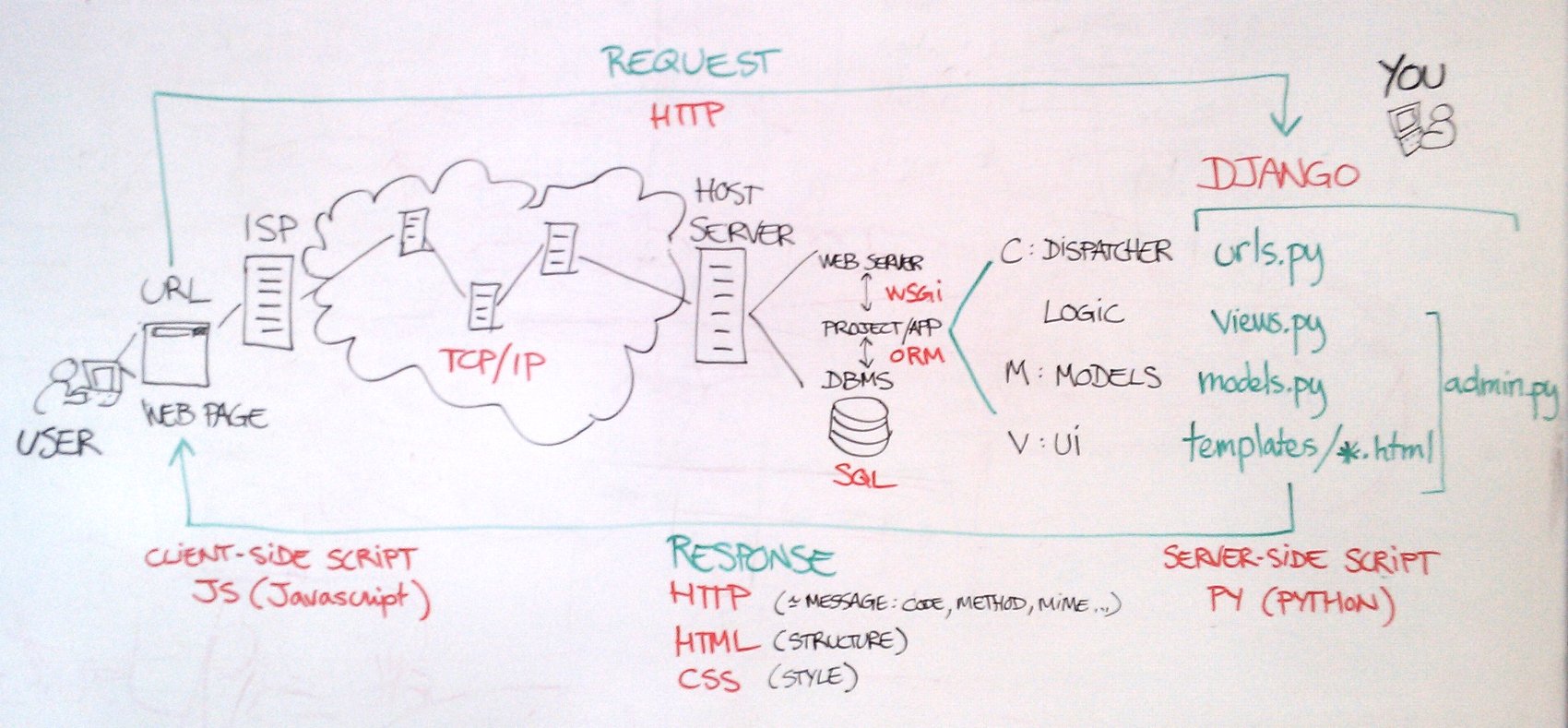


Diagram The django architecture

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

### Django is a high-level Python Web framework...

A Web framework is software that abstracts common problems of Web development and provides shortcuts for frequent programming tasks.

A good web framework finds the “pain points” of web developers and smoothes them over — but never gets in the way! It should let you work at a much higher level of abstraction, so you don’t need to worry about details of HTTP, SQL, or whatever. Again, it shouldn’t get in your way if you need to “step down” a level.

Python itself is another key “feature” of Django. It’s a beautiful, concise, powerful, high-level language with an amazing community; many things are easier in Django simply because of the tools you can build on top of.

Django also makes a point of not changing anything about how Python works; if you learn Django, you’re also learning Python. This helps down the road.

### ...that encourages rapid development...

Regardless of how many powerful features it has, a Web framework is worthless if it doesn't save you time. Django's philosophy is to do all it can to facilitate hyper-fast development. With Django, you build Web sites in a matter of hours, not days; weeks, not years.

This comes directly out of real-world problems. We’re programmers, yes, but we work at a news organization. When a big story breaks, we don’t have the luxury of a long development cycle. Every convenience in Django is there because it makes you more productive.

### ...and clean, pragmatic design.

Django strictly maintains a clean design throughout its own code and makes it easy to follow best Web-development practices in the applications you create.

The philosophy here is to make it easy to do things the "right" way.

### Django philosophy

https://docs.djangoproject.com/en/1.4/misc/design-philosophies/

#### Loose coupling

A fundamental goal of Django’s stack is [loose coupling and tight cohesion](http://c2.com/cgi/wiki?CouplingAndCohesion). The various layers of the framework shouldn’t “know” about each other unless absolutely necessary.

For example, the template system knows nothing about Web requests, the database layer knows nothing about data display and the view system doesn’t care which template system a programmer uses.

Although Django comes with a full stack for convenience, the pieces of the stack are independent of another wherever possible.

#### Less code

Django apps should use as little code as possible; they should lack boilerplate. Django should take full advantage of Python’s dynamic capabilities, such as introspection.

#### Quick development

The point of a Web framework in the 21st century is to make the tedious aspects of Web development fast. Django should allow for incredibly quick Web development.

#### Don’t repeat yourself (DRY)

Every distinct concept and/or piece of data should live in one, and only one, place. Redundancy is bad. Normalization is good.

The framework, within reason, should deduce as much as possible from as little as possible.

See also

The [discussion of DRY on the Portland Pattern Repository](http://c2.com/cgi/wiki?DontRepeatYourself)

#### Explicit is better than implicit

This is a core Python principle listed in [PEP 20](http://www.python.org/dev/peps/pep-0020), and it means Django shouldn’t do too much “magic.” Magic shouldn’t happen unless there’s a really good reason for it. Magic is worth using only if it creates a huge convenience unattainable in other ways, and it isn’t implemented in a way that confuses developers who are trying to learn how to use the feature.

#### Consistency

The framework should be consistent at all levels. Consistency applies to everything from low-level (the Python coding style used) to high-level (the “experience” of using Django).

# Preparing the development environment

From this chapter, we'll begin our trip to develop a real web app using Django.

Frist, we should prepare the development environment.

### Step 1: Install the python

Since the django only support the python 2.x, so we should install the python 2.x.

Download the latest python 2.x to install. After install please check the result as following:

*$ python*

*Python 2.7.1 (r271:86832, Jul 31 2011, 19:30:53)*

*[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin*

*Type "help", "copyright", "credits" or "license" for more information.*

*>>> print "hello world"*

*hello world*

*>>> exit()*

### Step 2 django development environment

*$ pip install django*

Downloading/unpacking django

Downloading Django-1.4.tar.gz (7.6Mb): 7.6Mb downloaded

Running setup.py egg\_info for package django

Installing collected packages: django

Running setup.py install for django

changing mode of build/scripts-2.7/django-admin.py from 644 to 755

changing mode of /Users/jeffrey/Dev/django\_tutorial/env/bin/django-admin.py to 755

Successfully installed django

Cleaning up...

You can see the virtualenv installed the pip to help us install django.

To check the version of Django:

*>>> import django*

*>>> django.get\_version()*

*'1.4'*

**What's pip?**

pip is a tool for installing and managing Python packages, such as those found in the [Python Package Index](http://pypi.python.org/pypi). It's a replacement for [easy\_install](http://peak.telecommunity.com/DevCenter/EasyInstall).

We'll put all dependent third python libraries into a txt file, such as requirement.txt.

*#All dependences here#*

*django*

*# If you need a special version #*

*# django==1.4#*

Then we can install all required packages using:

*pip install -r webapp/requirement.txt*

### Step 3: Using django-admin to build the a project

Django provides a tool to do django development tasks, such as build a project, add an app, sync db, etc. We'll introduce the tool later, now we need use it to build a project.

|  |
| --- |
| *$ django-admin.py startproject webapp*  *$ cd webapp/*  *$ python manage.py runserver*  *Validating models...*  *0 errors found*  *Django version 1.3.1, using settings 'webapp.settings'*  *Development server is running at http://127.0.0.1:8000/*  *Quit the server with CONTROL-C.* |

It worked!

**Choose an IDE**

Here are some IDEs which you can choice:

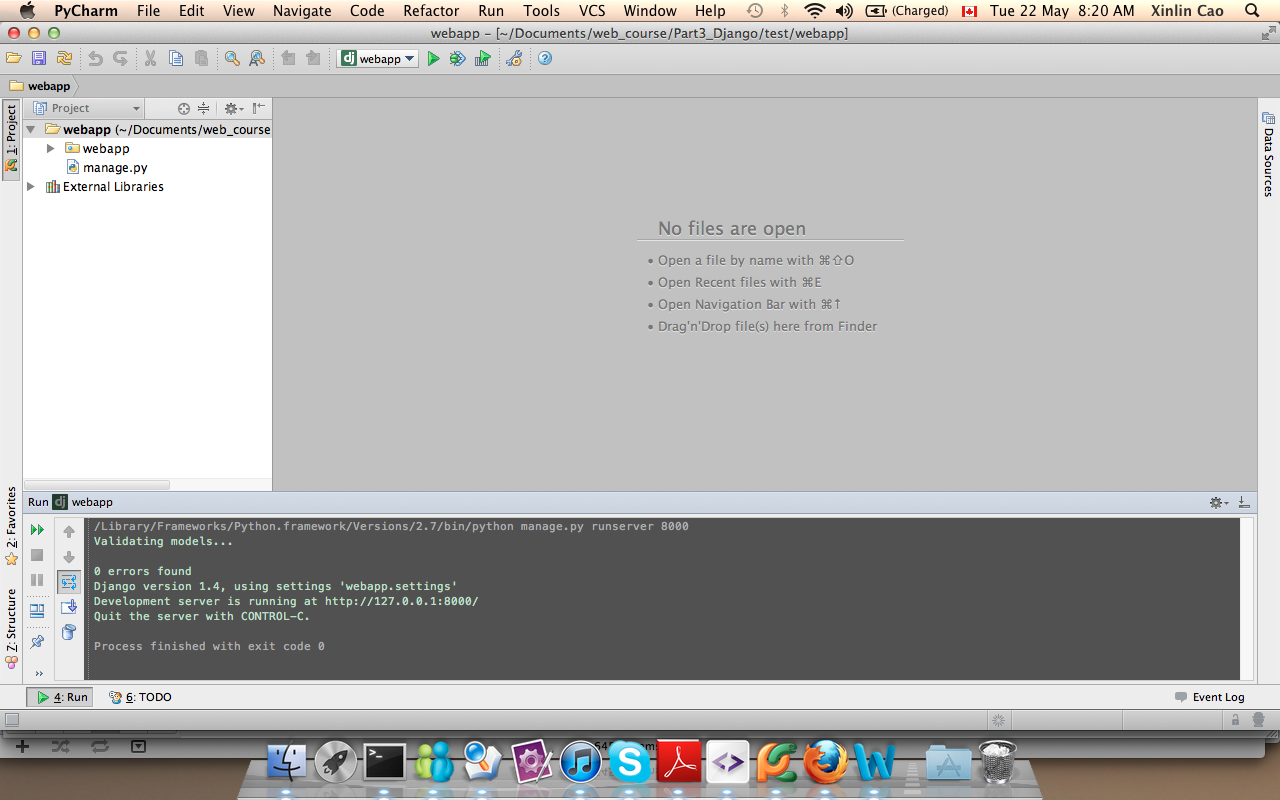
Text editor

Eclipse + pyDev

PyCharm

I suggest user can choice PyCharm, I like this IDE.

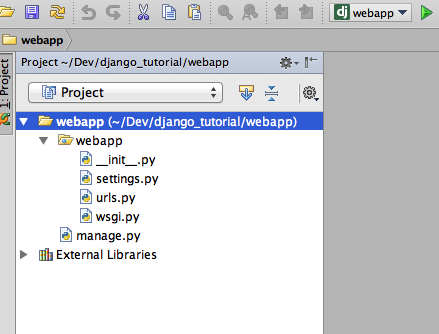
<http://www.jetbrains.com/pycharm/>



# Build the first app: basic request & response

In the last of [chapter 3](http://spidersns.com/wordpress/?p=119) we have build a web app, and run it worked! But it's only for demo and no practical purpose. Now we need go on to build our real application base on it.

Remember the latest web app?



These files are:

* The outer webapp/ directory is just a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
* manage.py: A command-line utility that lets you interact with this Django project in various ways. You can read all the details about manage.py in [django-admin.py and manage.py](https://docs.djangoproject.com/en/1.4/ref/django-admin/).
* The inner webapp/ directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. import webapp.settings).
* webapp/\_\_init\_\_.py: An empty file that tells Python that this directory should be considered a Python package. (Read [more about packages](http://docs.python.org/tutorial/modules.html#packages) in the official Python docs if you're a Python beginner.)
* webapp/settings.py: Settings/configuration for this Django project. [Django settings](https://docs.djangoproject.com/en/1.4/topics/settings/) will tell you all about how settings work.
* webapp/urls.py: The URL declarations for this Django project; a "table of contents" of your Django-powered site. You can read more about URLs in [URL dispatcher](https://docs.djangoproject.com/en/1.4/topics/http/urls/).
* webapp/wsgi.py: An entry-point for WSGI-compatible webservers to serve your project. See [How to deploy with WSGI](https://docs.djangoproject.com/en/1.4/howto/deployment/wsgi/) for more details.Prepare development

What is WSGI?

WSGI is the Web Server Gateway Interface. It is a specification for web servers and application servers to communicate with web applications (though it can also be used for more than that). It is a Python standard, described in detail in [PEP 333](http://www.python.org/dev/peps/pep-0333).

For more, see [Learn about WSGI](http://wsgi.readthedocs.org/en/latest/learn.html).

## Setup the database

In settings.py, we should define the database：

|  |
| --- |
| **import** os.path  PROJECT\_ROOT = os.path.abspath(os.path.dirname(\_\_file\_\_))  DATABASES = {  'default': {  'ENGINE': 'django.db.backends.sqlite3',  'NAME': os.path.join(PROJECT\_ROOT, 'dev.db'),  'USER': '', # Not used with sqlite3.  'PASSWORD': '', # Not used with sqlite3.  'HOST': '', # Set to empty string for localhost. Not used with sqlite3.  'PORT': '', # Set to empty string for default. Not used with sqlite3.  }  } |

And need to run "python manage.py syncdb" in webapp directory.

*$python manage.py syncdb*

*Creating tables ...*

*Creating table auth\_permission*

*Creating table auth\_group\_permissions*

*Creating table auth\_group*

*Creating table auth\_user\_user\_permissions*

*Creating table auth\_user\_groups*

*Creating table auth\_user*

*Creating table auth\_message*

*Creating table django\_content\_type*

*Creating table django\_session*

*Creating table django\_site*

*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 'jeffrey'): admin*

*E-mail address: admin@admin.com*

*Password:*

*Password (again):*

*Superuser created successfully.*

*Installing custom SQL ...*

*Installing indexes ...*

*No fixtures found.*

Also, we can run this command directly in Pycharm.

Note: If you encounter any errors about encoding, you can try “**export LC\_ALL="en\_US.UTF-8"**”(https://code.djangoproject.com/ticket/16017)

## Add "apps" package directory in "webapp"

We used the "apps" directory as the application root path.

Please note: apps is a python package directory, it include a file named "\_\_init\_\_.py", please do not forget it.

In "apps" package, we create the first app:

|  |
| --- |
| *$ cd webapp/webapp*  *$ mkdir apps*  *$ cd apps*  *$ touch \_\_init\_\_.py*  *$ python ../../manage.py startapp bookmark*  *$ cd bookmark*  *total 24*  *drwxr-xr-x 6 jeffrey staff 204 4 16 20:26 .*  *drwxr-xr-x 4 jeffrey staff 136 4 16 20:26 ..*  *-rw-r--r-- 1 jeffrey staff 0 4 16 20:26 \_\_init\_\_.py*  *-rw-r--r-- 1 jeffrey staff 57 4 16 20:26 models.py*  *-rw-r--r-- 1 jeffrey staff 383 4 16 20:26 tests.py*  *-rw-r--r-- 1 jeffrey staff 26 4 16 20:26 views.py* |

You can see in webapp/webapp/apps/bookmarks/ package, there are 4 python files.

* models.py: defined the models. A model is the single, definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. Django follows the [DRY Principle](https://docs.djangoproject.com/en/1.3/misc/design-philosophies/#dry). The goal is to define your data model in one place and automatically derive things from it.
* **views.py:** defined the views. A view is a “type” of Web page in your Django application that generally serves a specific function and has a specific template.

settings.py：

|  |
| --- |
| import sys  sys.path.insert(0, os.path.join(PROJECT\_ROOT))  sys.path.insert(0, os.path.join(PROJECT\_ROOT, "apps"))  INSTALLED\_APPS = (  'django.contrib.auth',  'django.contrib.contenttypes',  'django.contrib.sessions',  'django.contrib.sites',  'django.contrib.messages',  'django.contrib.staticfiles',  # Uncomment the next line to enable the admin:  # 'django.contrib.admin',  # Uncomment the next line to enable admin documentation:  # 'django.contrib.admindocs',  'bookmark',  ) |

## Defining views

A view function, or view for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it’s on your Python path. There’s no other requirement–no “magic,” so to speak. For the sake of putting the code somewhere, the convention is to put views in a file called views.py, placed in your project or application directory.

Imaging for the bookmark objects, the owner can create/update/delete/view a bookmark, and list bookmarks.

So we need add some method in views, for every method can correspond a option of the owner.

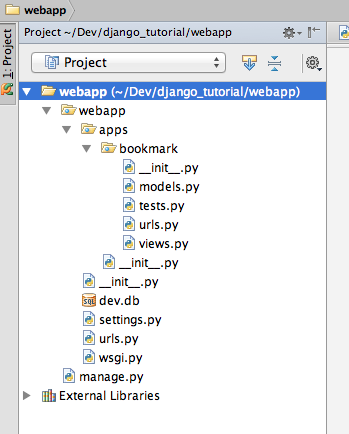
Now we add the views.py

|  |
| --- |
| **from** django.http **import** HttpResponse  **def** index(request):  **return** HttpResponse("index")  **def** create(request):  **return** HttpResponse("create")  **def** update(request):  **return** HttpResponse("update")  **def** delete(request):  **return** HttpResponse("delete")  **def** show(request):  **return** HttpResponse("show") |

## Define the urls

After defining the views.py, we need to define the urls.py. A urls.py is a url mapping to views method.

Put the urls.py in the bookmark directory.



|  |
| --- |
| **from** django.conf.urls.defaults **import** patterns, url  urlpatterns = patterns('',  url(r'^/index/$', 'bookmark.views.index’, name='bookmark\_list'),  url(r'^/create/$', 'bookmark.views.create', name='bookmark\_create'),  url(r'^/update/$', 'bookmark.views.update', name='bookmark\_update'),  url(r'^/delete/$', 'bookmark.views.delete', name='bookmark\_delete'),  url(r'^/show/$', 'bookmark.views.show', name='bookmark\_show'),  ) |

Find the urls.py in webapp/webapp directory, add the url mapping to bookmark.urls：

|  |
| --- |
| from django.conf.urls import patterns, include, url  urlpatterns = patterns('',  url(r'^bookmark', include('bookmark.urls')),  ) |

Now we run the app by using "python manage.py runserver". And input the url for "http://localhost:8000/bookmark/index", you will find the page show the text "index".

**Summary**

In this part, we did a real app named "bookmark" and setup the Directory file structure.

User input the url, django will use the urls.py to mapping the method

Views.py defined the methods which get the user request and return response

Need make sure your app added in "INSTALLED\_APPS" in settings.py

## Build the model

|  |
| --- |
| from django.contrib.auth.models import User  from django.db import models  class Bookmark(models.Model):  url = models.URLField()  title = models.CharField(blank=True, max\_length=100, help\_text='Limit of 100 characters')  public = models.BooleanField(verbose\_name='Public this bookmark', blank=True, default=True)  create\_time = models.DateTimeField(auto\_now\_add=True)  last\_modified\_time = models.DateTimeField(auto\_now=True)  def \_\_unicode\_\_(self):  return self.url |

## Define the index view

In the apps/bookmark/views.py, let’s define a new list method which can show all bookmarks data from db to page.

The steps are:

Get a bookmarks list from db

Render the bookmarks list to page(template)

That’s all, now we code:

|  |
| --- |
| **from** django.http **import** HttpResponse  **from** django.shortcuts **import** render\_to\_response  **from** models **import** Bookmark  **def** index(request):  bookmarks = Bookmark.objects.all()  **return** render\_to\_response('index.html',  {  'bookmarks': bookmarks,  }) |

bookmarks = Bookmark.objects.all() can get the all bookmarks list from db, there is no need to write any sql, but it actually worked as you do in other language:

Create db connection

Write a sql and execute the sql then get a result set

Get the list from the result set

return render\_to\_response('index.html', {…}) just tell us django will render the bookmarks list to the template “index.html”, so we can use this “bookmarks” in the template page directly.

Let’s look the “index.html” (in bookmarks/ templates directory)

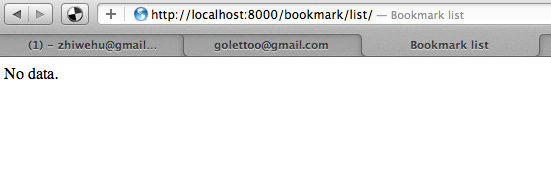
|  |
| --- |
| <**html**>  <**head**>  <**title**>Bookmark list</**title**>  </**head**>  <**body**>  {% for bookmark in bookmarks %}  {{ bookmark }}  {% empty %}  No data.  {% endfor %}  </**body**>  </**html**> |

In the “index.html” we can see the django template markup language:

{% %} can execute some logic code, such as if, for,etc.

{{ }} can display some variable to page.

In this example, we use {% for %} {% empty %} {% endfor %} markup language, you can get the meaning from the literal meaning. It’s means “loop the bookmarks list and print every bookmark, if the bookmarks is empty, just tell user no data.” Simple? Yes, but powerful!



## Define the show view

For the show method, we just need to do are:

Define the show template (show.html):

|  |
| --- |
| <html>  <head>  <title>Show Bookmark</title>  </head>  <body>  {{ bookmark }}  </body>  </html> |

Get a bookmark which user want to show

Render the bookmark to page

Let’s look how to implement in code:

|  |
| --- |
| …  **from** django.http **import** HttpResponse, Http404  …  **def** show(request, bookmark\_id):  **try**:  bookmark = Bookmark.objects.get(id=bookmark\_id)  **except** Bookmark.DoesNotExist:  **raise** Http404  **return** render\_to\_response('show.html',  {  'bookmark': bookmark,  }) |

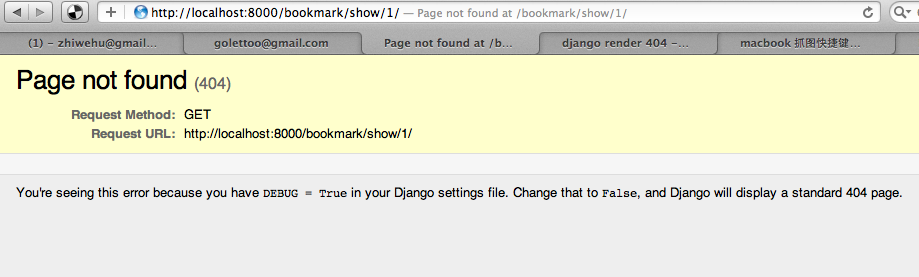
bookmark = Bookmark.objects.get(id=bookmark\_id), **This line can get a bookmark object from db by bookmark\_id, and the question is, where get the bookmark\_id?**

Please look the urls.py, and see how we define the “show” method url mapping:

|  |
| --- |
| url(r'^/show/(?P<bookmark\_id>\d+)/$', 'bookmark.views.show', name='bookmark\_show'), |

You can see, we have defined a url which can get the bookmark\_id from url input, for example, if the user input the url like this: <http://localhost:8000/bookmark/show/1/>, it tell the django urls.py to mapping this url to “bookmark.views.show” method, and pass the “bookmark\_id” parameter to this method.

Using try…except… to protect if the user input a non-exists data, if the user input an id which not exists in db, it’ll raise a HTTP404 error.



## Define the delete view

Just like the “show” method, the delete method need a parameter let django to know which data need to delete, and also, if the data does not exists, it’ll raise a HTTP404 error.

We can change the url of this method to:

|  |
| --- |
| url(r'^/delete/(?P<bookmark\_id>\d+)/$', 'bookmark.views.delete', name='bookmark\_delete'), |

|  |
| --- |
| def delete(request, bookmark\_id):  try:  bookmark = Bookmark.objects.get(id=bookmark\_id)  bookmark.delete()  except Bookmark.DoesNotExist:  raise Http404  return index(request) |

bookmark.delete() method will delete this data in db.

After delete, we just call return index(request) method, so we can go to the index page again.

# Django Admin

One of the most powerful parts of Django is the automatic admin interface. It reads metadata in your model to provide a powerful and production-ready interface that content producers can immediately use to start adding content to the site. In this document, we discuss how to activate, use and customize Django’s admin interface.

There are seven steps in activating the Django admin site:

* Add 'django.contrib.admin' to your [INSTALLED\_APPS](https://docs.djangoproject.com/en/1.4/ref/settings/#std:setting-INSTALLED_APPS) setting.
* The admin has four dependencies - [django.contrib.auth](https://docs.djangoproject.com/en/1.4/topics/auth/#module-django.contrib.auth), [django.contrib.contenttypes](https://docs.djangoproject.com/en/1.4/ref/contrib/contenttypes/#module-django.contrib.contenttypes), [django.contrib.messages](https://docs.djangoproject.com/en/1.4/ref/contrib/messages/#module-django.contrib.messages) and [django.contrib.sessions](https://docs.djangoproject.com/en/1.4/topics/http/sessions/#module-django.contrib.sessions). If these applications are not in your [INSTALLED\_APPS](https://docs.djangoproject.com/en/1.4/ref/settings/#std:setting-INSTALLED_APPS) list, add them.
* Add django.contrib.messages.context\_processors.messages to [TEMPLATE\_CONTEXT\_PROCESSORS](https://docs.djangoproject.com/en/1.4/ref/settings/#std:setting-TEMPLATE_CONTEXT_PROCESSORS) and [MessageMiddleware](https://docs.djangoproject.com/en/1.4/ref/middleware/#django.contrib.messages.middleware.MessageMiddleware) to [MIDDLEWARE\_CLASSES](https://docs.djangoproject.com/en/1.4/ref/settings/#std:setting-MIDDLEWARE_CLASSES). (These are both active by default, so you only need to do this if you’ve manually tweaked the settings.)
* Determine which of your application’s models should be editable in the admin interface.
* For each of those models, optionally create a ModelAdmin class that encapsulates the customized admin functionality and options for that particular model.
* Instantiate an AdminSite and tell it about each of your models and ModelAdmin classes.
* Hook the AdminSite instance into your URLconf.

After you’ve taken these steps, you’ll be able to use your Django admin site by visiting the URL you hooked it into (/admin/, by default). Without writing a line of code, you can have a powerful web admin page.

1. Add django.contrib.admin into your installed apps list in settings.py

|  |
| --- |
| INSTALLED\_APPS = (  'django.contrib.auth',  'django.contrib.contenttypes',  'django.contrib.sessions',  'django.contrib.sites',  'django.contrib.messages',  'django.contrib.staticfiles',  # Uncomment the next line to enable the admin:  'django.contrib.admin',  # Uncomment the next line to enable admin documentation:  # 'django.contrib.admindocs',  'bookmark',  ) |

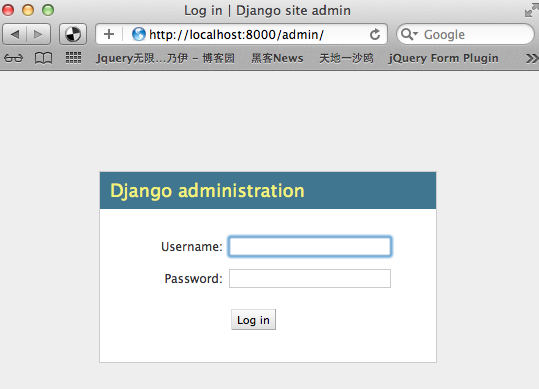
1. Add admin in urls.py

|  |
| --- |
| from django.conf.urls import patterns, include, url  # Uncomment the next two lines to enable the admin:  from django.contrib import admin  admin.autodiscover()  urlpatterns = patterns('',  url(r'^bookmark', include('bookmark.urls')),  url(r'^admin/', include(admin.site.urls)),  ) |

1. Add an admin.py (webapp/webapp/apps/bookmark/admin.py)

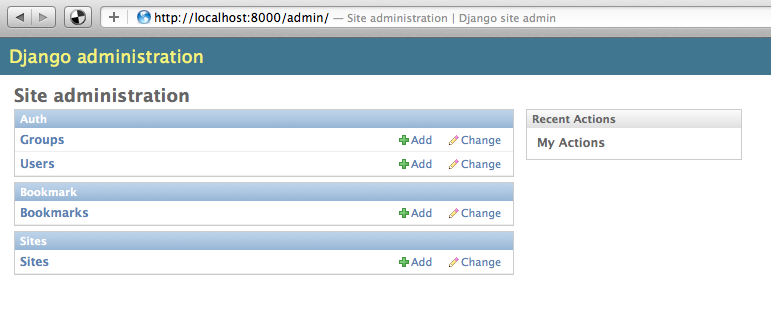
|  |
| --- |
| from django.contrib import admin  from models import Bookmark  class BookmarkAdmin(admin.ModelAdmin):  pass  admin.site.register(Bookmark, BookmarkAdmin) |

Now check the url <http://localhost:8000/admin/> you will see



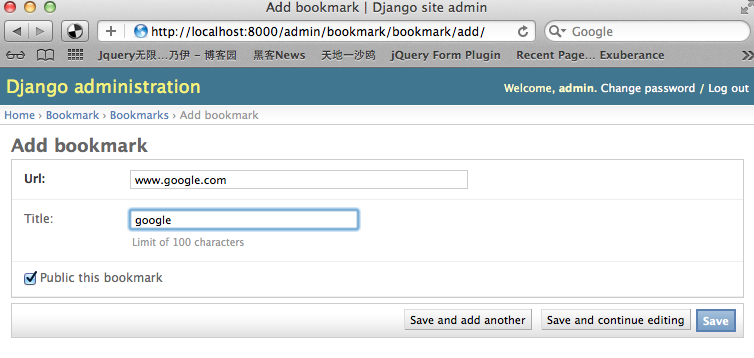
Input the admin user and password(remember we syncdb first time and let me input the user name and password?)

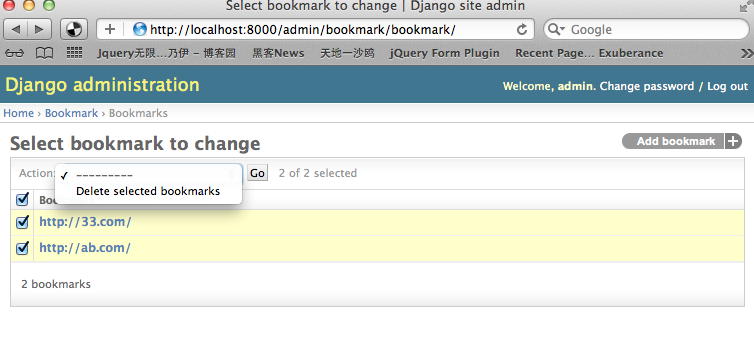
1. After login, you can see django have build-in a power admin page let you CRUD data.



And you can see the Bookmarks link, let’s see what operations we can do?

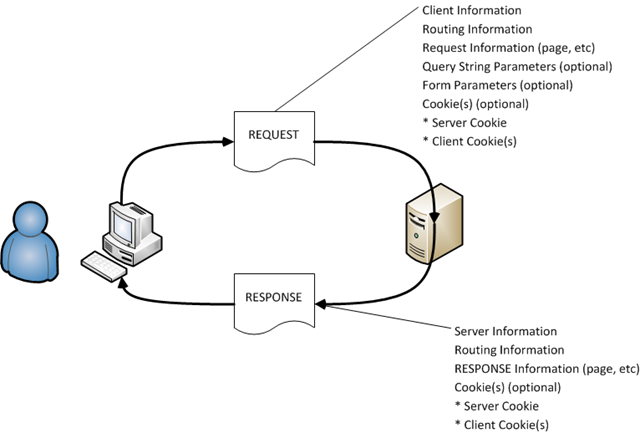
We can add/edit/delete/batch delete/search bookmarks!





# Models, Views and Templates

Web application is base on request-response mode.



## The MVC design pattern:

Model–View–Controller (MVC) is a [design pattern](http://en.wikipedia.org/wiki/Design_pattern) for computer [user interfaces](http://en.wikipedia.org/wiki/User_interface) that divides an application into three areas of responsibility:

the Model: the [domain objects](http://en.wikipedia.org/wiki/Domain_objects) or [data structures](http://en.wikipedia.org/wiki/Data_structures) that represent the application's state.

the View, which observes the state and generates output to the users.

the Controller, which translates user input into operations on the model.

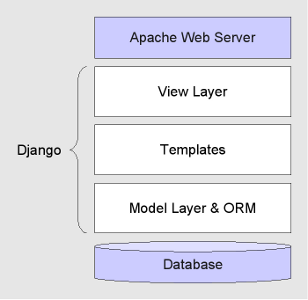


In django, MVC=MVT

Controller = django Views

View = django templates

In Django, the Python method called when a URL is requested is called a view, and the page loaded and rendered by the view is called a template. Because of this, the Django team refers to Django as an MVT (model-view-template) framework. TurboGears, on the other hand, calls its methods controllers and their rendered templates views so that they can fit squarely into the MVC acronym. The difference is largely semantic, as they accomplish the same things.



## User input url address to request something

The urls.py will mapping the url to python method which defined in views.py

views.py methods will filter users’ request, get parameters from the request.GET or request.POST, then do some query from db, get the models from models.py

views.py will use the templates to show the data to user, django will render the data (include models data) to template page.(html)

# Forms

Form is used to get input form the user. In Django, Form framework is easy. We can ask Django to generate form semi-automatically. We will use Form to create and modify data.

## Define the “create” method

Let’s make a create method, first let user input a form, and submit the form, the method can handle the data which post in a form, then validate the data first

If validate pass, save it to db.

First, define a forms.py in bookmark directory, we used to write all forms in this file:

|  |
| --- |
| **from** django **import** forms  **from** models **import** Bookmark  **class** BookmarkForm(forms.ModelForm):  **class** Meta:  model = Bookmark  fields = ('url','title','public') |

The BookmarkForm is a subclass of django.forms.ModelForm, this from need can bind a model automatically. Simple? Powerful! But if you need customize the form, maybe you need to use the django.forms.Form, it’ll not bind the model, you should bind the data in the form to a model by yourself.

Model=Bookmark in “class Meta” tell us the form bind the model Bookmark

fields tell us which fields in the Bookmark model need to show in the form.

Let’s look the create method in views.py

For first time user input the create url, it’ll create an empty form and render it to the “create.html” template.

If the user submit the form with the “post” method, the method will check it and handle the data in the form

form.is\_valid() method will check if the data are all ok for this form. It can defined some valid method in the “BookmarkForm”, we left it empty, but since the “BookmarkForm” is bind with the model “Bookmark”, so for example, the Bookmark.url is not allow null or empty, so if the user didn’t input the url field in the form, it’ll get an error, we’ll look later.

bookmark = form.save() can save the data just use the form.save() since the form is bind to the model, if the form doesn’t bind the model, it need you write the method by self. For example:

*url=form.cleaned\_data[‘url’]*

*title=form.cleaned\_data[‘title’]*

*public= form.cleaned\_data[‘public’]*

*bookmark = Bookmark(url=url, title=title, public=public)*

*bookmark.save()*

Now we need to define the “create.html”, this html need render an empty form in the page.

|  |
| --- |
| <**html**>  <**head**>  <**title**>Create a bookmark</**title**>  </**head**>  <**body**>  <**form** action="." method="post">  {% csrf\_token %}  {{ form.as\_p }}  <**input** type="submit" />  <**input** type="reset" />  </**form**>  </**body**>  </**html**> |

We defined a <form>, which action=”.”, it’s mean the action is same with the current page, just like action=”/bookmark/create/”, the mehod is “post”

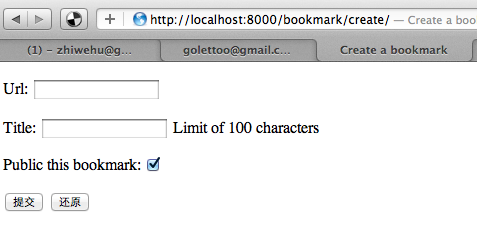
{% csrf\_token %} is django build-in plugin which provide easy-to-use protection against “Cross site request forgeries”. For the detail, please check here: <https://docs.djangoproject.com/en/dev/ref/contrib/csrf/>

{{ form.as\_p }} is a quick way to display the form with <p></p>

Then, let’s define the view function for “create”:

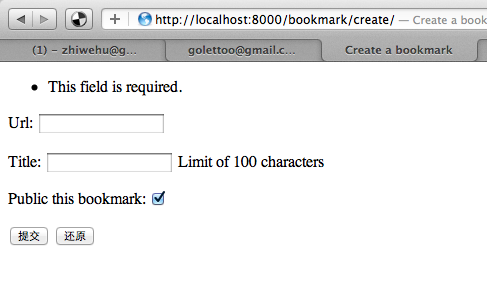
|  |
| --- |
| from django.template.context import RequestContext  from forms import \*  from django.http import \*  from django.core.urlresolvers import reverse  def create(request):  if request.method == 'POST': # If the form has been submitted...  form = BookmarkForm(request.POST) # A form bound to the POST data  if form.is\_valid(): # All validation rules pass  bookmark = form.save()  return HttpResponseRedirect(reverse('bookmark\_list')) # Redirect after POST  else:  form = BookmarkForm() # An unbound form  return render\_to\_response('create.html',  RequestContext(request, {  'form': form,  })) |

That’s ok, let me check the result:



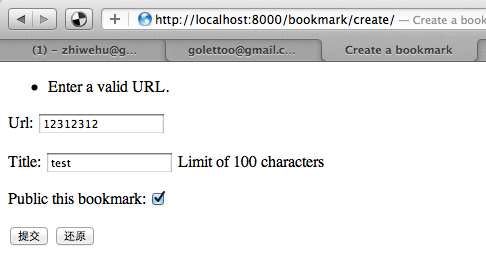
You can see the page display a from with labels, and input text, for the “public” field of Bookmark model, since this field is a Boolean type, it can use a checkbox to display it automatically! You just use the {{ form.as\_p }}, and the django form template tag “as\_p” will do it. Easy?

Then, let us chick the “input” button without input any data, and you will see:



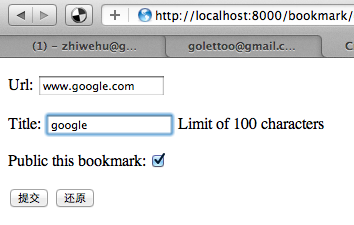
Yeah! Django checked the url field is not allow empty, so, if you forget to input this field, django will valid the data, and return errors.

Now, we input something, for the url input, we just input some text but not a valid url, and you can see:

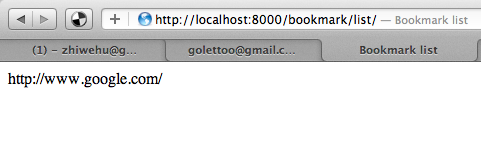


It cannot pass the form.is\_valid() method again, and this time, the error is “Enter a valid URL”, cool? Without write 1 code to validate the form data, but django help you to do it!

At the end, let us input some valid data.



After submit, it’ll redirect to “index” page



Tips:

We used the return HttpResponseRedirect(reverse('bookmark\_list')) # Redirect after POST, it call a reverse method, this method will get the url by name, please check the urls.py, we defined the url like:

url(r'^/index/$', 'bookmark.views.index, name='bookmark\_index'),

Define a url name can make django reuse it by name, even you changed the url.

## Define “update” method

Ok, now the latest method is how we update an exist bookmark?

Let’s define the template for “update” (update.html):

|  |
| --- |
| <html>  <head>  <title>Update a bookmark</title>  </head>  <body>  <form action="." method="post">  {% csrf\_token %}  {{ form.as\_p }}  <input type="submit" />  <input type="reset" />  </form>  </body>  </html> |

Get the bookmark by bookmark\_id parameter

Render the bookmark form with the bookmark data

User can update the bookmark data in the form and then submit

If the data in form valid, it’ll update the bookmark and save it to db

Define the url:

|  |
| --- |
| url(r'^/update/(?P<bookmark\_id>\d+)/$', 'bookmark.views.update', name='bookmark\_update'), |

Let’s see how the views.py looks:

|  |
| --- |
| **def** update(request, bookmark\_id):  **try**:  bookmark = Bookmark.objects.get(id=bookmark\_id)  **except** Bookmark.DoesNotExist:  **raise** Http404  **if** request.method == 'POST': # If the form has been submitted...  form = BookmarkForm(request.POST, instance=bookmark)  **if** form.is\_valid(): # All validation rules pass  bookmark = form.save()  **return** HttpResponseRedirect(reverse('bookmark\_list')) # Redirect after POST  **else**:  form = BookmarkForm(instance=bookmark)  **return** render\_to\_response('update.html',  RequestContext(request, {  'form': form,  })) |

form = BookmarkForm(request.POST, instance=bookmark), the instance is the bookmark which has been exist in db.

Now you can see, the code is very similar with the “create” mehod, even the html template. A Django philosophy is “do not repeat yourself”, so we’ll consider use a common method can handle “create” and “updata” in later chapter.

Summary:

This chapter just implement the very basic concepts for django how to hander request and response data(render data to template). We defined a model, a view which have some methods, some templates to display render data. The urls.py mapping the url and method with some regular expression. And we used the django ModelForm in help us handle user input data in a form.

# Static and media files

## Image and other static files

todo

## CSS file

todo

# User authentication

User registration and account management are universal features found in every web application. Users need to identify themselves to the application before they can post and share content with other users. User accounts are also required for activities such as online discussions and friend networks.

The Django authentication system is available in the django.contrib.auth package. It is installed by default as part of Django and projects created with the django-admin.py utility have it enabled by default.

## Creating the login page

Todo

## Enabling logout functionality

Todo

## User registration

Todo

# Message

Todo

# Ajax

Todo

# Bootstrap

Todo

# 脚手架

todo

# EC2

todo