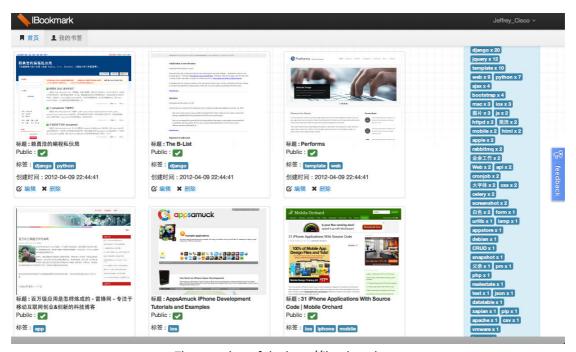
# Django web application development tutorial

Django web application development tutorial	
The goal of this tutorial	
Introduction to Django	4
What is Django	
Django is a high-level Python Web framework	
Django encourages rapid development	
and clean, pragmatic design	
Django philosophy	
Other frameworks	
How to learn Django	
Preparing the development environment	7
Step 1: Install the python and PIP	7
Step 2 Setup Django development environment	
Step 3: Using Django-admin to build the a project	8
Choose an IDE?	9
Inspect the Django project structure	9
Basic request & response	11
Setup the database	11
Add "apps" package directory in "webapp"	12
Defining views for app	13
Define the urls	14
Build the model	16
Django Admin	17
Define our own view functions	20
Define the index view	20
Define the show view	21
Define the delete view	22
Models, Views and Templates	23
The MVC design pattern:	23
User input url address to request something	
Model	25
View	25
Template	
Inheritance	26
Forms	26
Define the "create" method	26
Define "update" method	
Unify "Create" and "Update"	
Static and media files	
Image and other static files	
CSS file	

Bootstrap template	37
User authentication	40
Creating the login page	40
Enabling logout functionality	
User registration	
3 <sup>rd</sup> party apps (django-allauth)	
EC2 (apache, MySQL)	
Todo	

# The goal of this tutorial

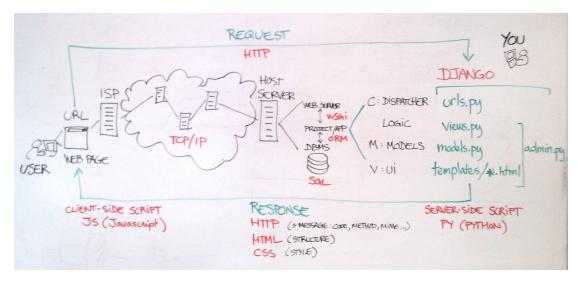
Recently, We developed a Django web app named "iBookmark (http://www.ibookmark.me)", which can help users manage their bookmark online, just like Google and delicious bookmark. Users can import and export their bookmarks from browsers and Google, delicious.



The snapshot of the http://ibookmark.me

The tutorial goal is to show how to use Python/Django to build a small popular web app. I'll use the "iBookmark" as the example for this tutorial.

# **Introduction to Django**



The Django architecture

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

### What is Django

## 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.

## ...Django encourages rapid development...

Regardless of how many powerful features it has, a Web framework is worthless if it doesn't save you time (Do you mean Java?). 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. 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, 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).

### Other frameworks

Compared to other frameworks, Django is quick, not dirty, full-stack and popular in job markets.

## How to learn Django

- 1: Learn Python
- 2: Learn Django concepts and conventions
- 3: Google and Document

# Preparing the development environment

From this chapter, we'll begin our trip to develop a real but tiny web app using Django. First, we should prepare the development environment.

## Step 1: Install the python and PIP

Since Django only supports 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()

PIP is a tool for installing and managing Python packages, such as those found in the Python Package Index. It's a replacement for easy\_install. To install PIP, we can check install document at:

http://www.pip-installer.org/en/latest/installing.html

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

#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 2 Setup 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...

To check the version of Django:

```
>>> import Django
>>> Django.get_version()
'1.4'
```

## 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 database, 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.

If you open browser and visit the server, you can find the server should work.

#### It worked!

Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Here's what to do next:

- If you plan to use a database, edit the DATABASES setting in webapp/settings.py.
- Start your first app by running python manage.py startapp [appname].

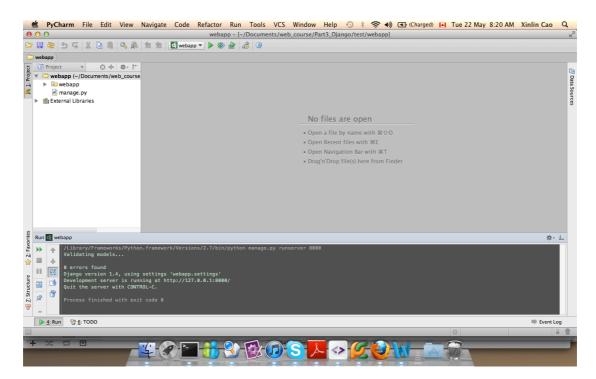
You're seeing this message because you have DEBUG = True in your Django settings file and you haven't configured any URLs. Get to work!

### **Choose an IDE?**

Here are some IDEs which you can choose:

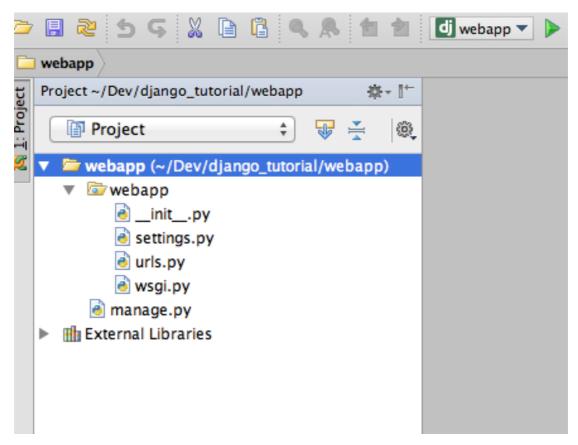
- Text editor
- Eclipse + pyDev
- PyCharm

We suggest user can choice PyCharm, We like this IDE (http://www.jetbrains.com/pycharm/).



## Inspect the Django project structure

Remember the latest web app?



The following is a list of file generated in Django project folder:

- 1. 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.
- 2. 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.
- 3. 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).
- 4. webapp/\_init\_.py: An empty file that tells Python that this directory should be considered a Python package. (Read more about packages in the official Python docs if you're a Python beginner.)
- 5. webapp/settings.py: Settings/configuration for this Django project. Django settings will tell you all about how settings work.
- 6. 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.
- 7. webapp/wsgi.py: An entry-point for WSGI-compatible webservers to serve your project. See How to deploy with 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.

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

# **Basic request & response**

In the last chapter, we have built 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.

### 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_permissions
Creating table auth_user_groups
Creating table auth_user
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.

Note: If you encounter any errors about encoding, you can type "export LC\_ALL="en\_US.UTF-8""(https://code.Djangoproject.com/ticket/16017) in the terminal to solve this issue.

After we have generated database, then we can use some sqlite took, such as sqlitemanager, to debug and manage the database.

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

We used the "apps" directory as the application root path. Please note that "apps" is a Python package directory, it need to 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.

1. 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. The goal is to define your data model in one place and automatically derive things from it.

2. 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.

Then we need enable our created application in 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.messages',
    '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 for app**

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 an option of the owner.

Now we add the following functions in the views.py:

```
from Django.http import HttpResponse

def index(request):
    return HttpResponse("index")
```

```
def create(request):
    return HttpResponse("create")

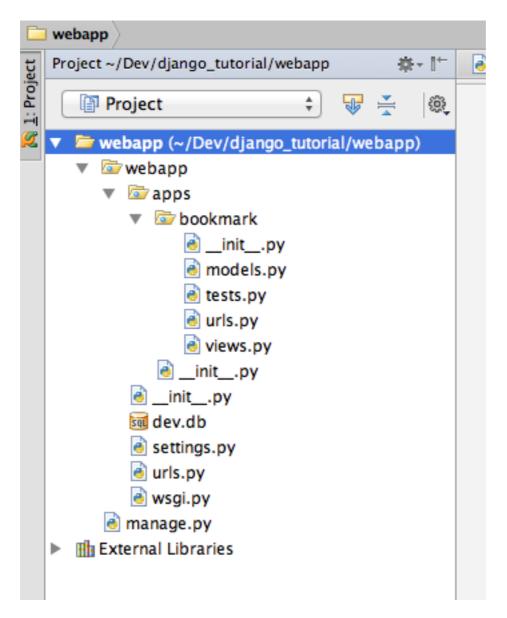
def update(request):
    return HttpResponse("update")

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. We put the urls.py in the bookmark directory.



```
from Django.conf.urls.defaults import patterns, url

urlpatterns = patterns(",
 url(r'^/index/$', 'bookmark.views.index', name='bookmark_index'),
 url(r'^/create/$', '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'),
}
```

Then we need to connect our app url with project url mapping file. Find the urls.py in webapp/webapp directory, add the url mapping to bookmark.urls:

from Django.conf.urls import patterns, include, url

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".

### Why two URL configurations?

Loose coupling

#### **Summary**

In this part, we make a real app named "bookmark" and setup the Directory file structure. When users input the url, Django will use the urls.py search corresponding view function to response. View function in views.py can get the user request and return response back to users.

#### **Build the model**

Now let's build our core data structure and business model for our app. 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. Generally, each model maps to a single database table.

Each model is a Python class that subclasses django.db.models.Model.

Each attribute of the model represents a database field.

With all of this, Django gives you an automatically-generated database-access API;

For more information, please see also https://docs.djangoproject.com/en/dev/topics/db/models/.

```
def __unicode__(self):
    return self.url
```

We have used different types of model fields. For more information, please check https://docs.djangoproject.com/en/dev/ref/models/fields/#field-types.

#### Notes:

### DateField.auto\_now

Automatically set the field to now every time the object is saved. Useful for "last-modified" timestamps. Note that the current date is always used; it's not just a default value that you can override.

#### DateField.auto\_now\_add

Automatically set the field to now when the object is first created. Useful for creation of timestamps. Note that the current date is always used; it's not just a default value that you can override.

We can delete the database file and syncdb the database again.

\$python manage.py syncdb

# **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.

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',
)
```

### 2. 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)),
)
```

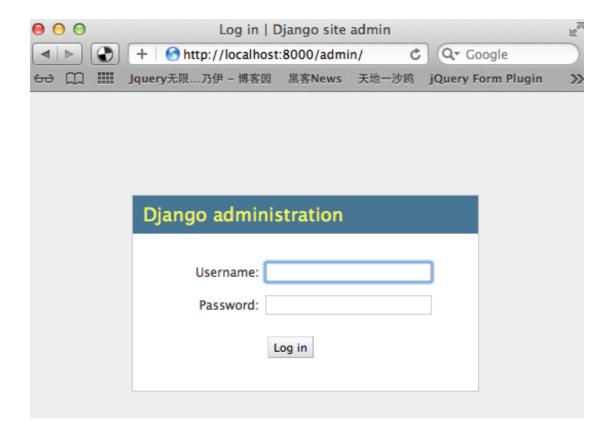
### 3. Add an admin.py (webapp/webapp/apps/bookmark/admin.py)

```
from Django.contrib import admin
from models import Bookmark

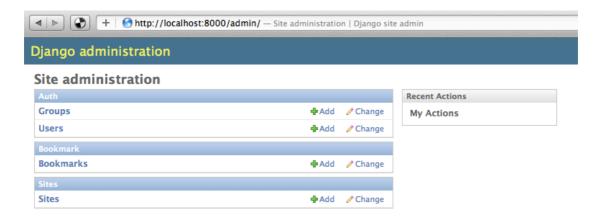
class BookmarkAdmin(admin.ModelAdmin):
    pass

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?) After login, you can see Django have build-in a power admin page let you CRUD data.



## **Define our own view functions**

#### Define the index view

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

The steps are:

- 1. Get a bookmarks list from database.
- 2. Render the bookmarks list to page (template)

That's all, now we code:

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

- 1. Create database connection
- 2. Write a sql and execute the sql then get a result set
- 3. Get the list from the result set

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)

```
{% 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!



No data.

#### Define the show view

In this section, we would like to define a show function to display the detail of a bookmark record.

#### Steps:

- 1. Get a bookmark which user want to show
- 2. Render the bookmark to page

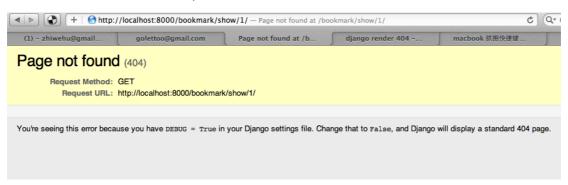
Let's look how to implement in code:

```
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,
        })
```

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

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 so to know which data need to delete, and also, if the data does not exists, it'll raise an HTTP404 error.

We can change the url of this method to:

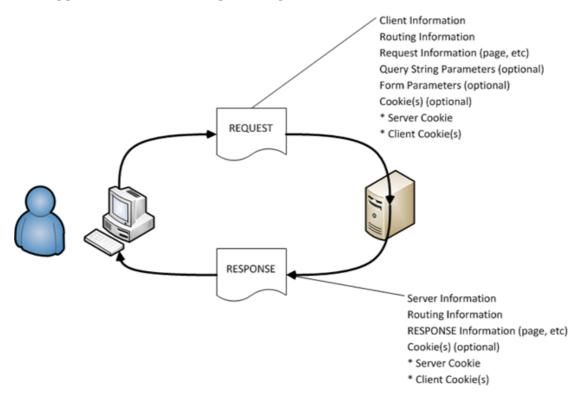
#### The view function code:

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

# **Models, Views and Templates**

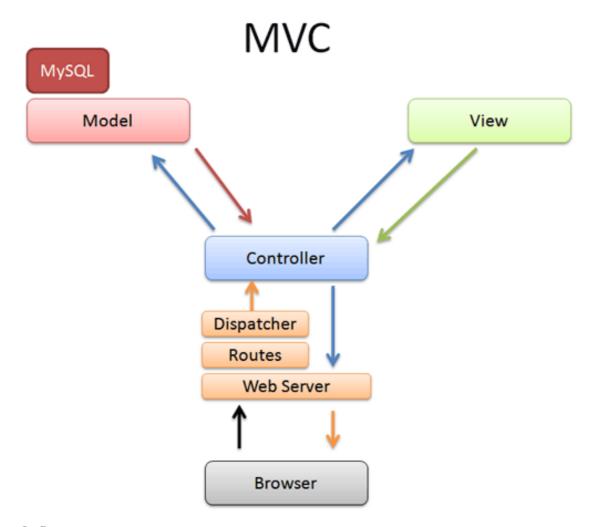
Web application is base on request-response mode.



# The MVC design pattern:

Model-View-Controller (MVC) is a design pattern for software that divides an application into three areas of responsibility:

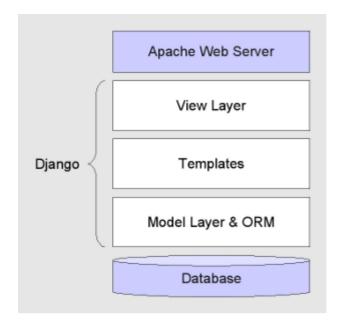
- Model: the domain objects or data structures that represent the application's state.
- View: which observes the state and generates output to the users.
- Controller, which translates user input into operations on the model.



#### 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.



## 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 database, 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).

#### Model

Data structure and mathematic model

#### View

Controller

## **Template**

https://docs.Djangoproject.com/en/dev/ref/templates/

Django's template engine provides a powerful mini-language for defining the user-facing layer of your application, encouraging a clean separation of application and presentation logic. Templates can be maintained by anyone with an understanding of HTML; no knowledge of Python is required.

### **Inheritance**

https://docs.Djangoproject.com/en/dev/topics/templates/#template-inheritance

## **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 it can pass validation, then Django will 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 which fields in the Bookmark model need to show in the form.

For first time user input the "create" url, it'll create an empty form and render it to the "create.html" template. Now we need to define the "create.html", this html need render an empty form in the page.

<html>

We defined a <form>, which action=".", it's mean the action is same with the current page, just like action="/bookmark/create/", the method 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

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()
                       HttpResponseRedirect(reverse('bookmark_index'))
             return
Redirect after POST
    else:
         form = BookmarkForm() # An unbound form
    return render_to_response('create.html',
         RequestContext(request, {
             'form': form,
```

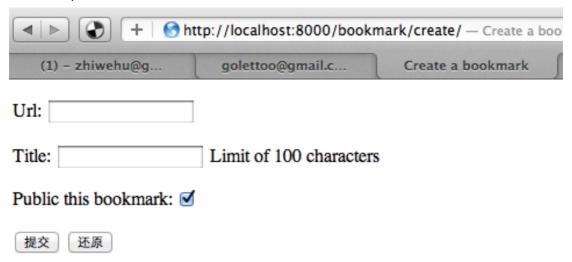
If a user submits the form with the "post" method, the method will check it and handle the data in the form.

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 bound to the model, if the form doesn't bind the model, it needs you to write the method by self. For example:

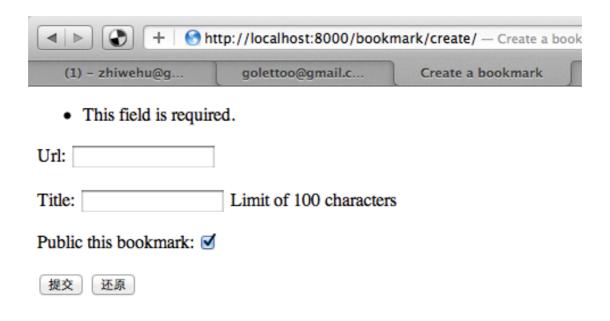
```
url=form.cleaned_data['url']
title=form.cleaned_data['title']
public= form.cleaned_data['public']
bookmark = Bookmark(url=url, title=title, public=public)
bookmark.save()
```

#### That's ok, let me check the result:



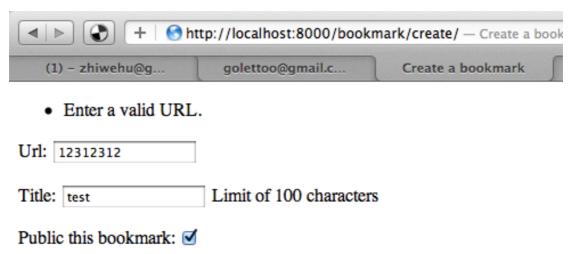
You can see the page display a form with labels, and input text, for the "public" field of Bookmark model, since this field is a Boolean type, it can use a checkbox to display it automatically! You just use the {{ form.as\_p}}, and the Django form template tag "as\_p" will do it. Easy?

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:

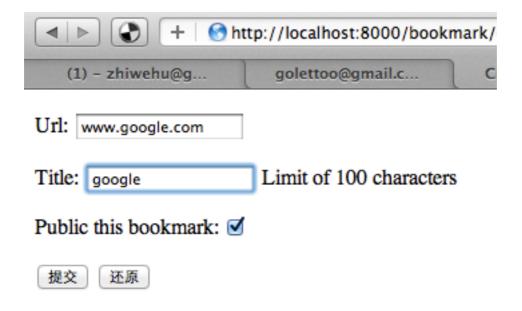


It cannot pass the form.is\_valid() method again, and this time, the error is "Enter a valid URL", cool? Without write one line of code to validate the form data, but Django help you do it!

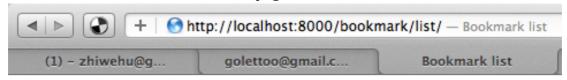
At the end, let us input some valid data.

提交

还原



After submit, it'll redirect to "index" page



http://www.google.com/

#### Tips:

We used HttpResponseRedirect(reverse('bookmark\_index')) to redirect to another page. Reverse method can 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 last method is how we update an existing 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>
```

#### Steps to update an existing record:

- Get the bookmark by bookmark\_id parameter
- 2. Render the bookmark form with the bookmark data
- 3. User can update the bookmark data in the form and then submit
- 4. If the data in form is valid, it'll update the bookmark and save it to db

### Let's define the url for update method:

#### 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()
                       HttpResponseRedirect(reverse('bookmark_index'))
             return
Redirect after POST
    else:
        form = BookmarkForm(instance=bookmark)
    return render_to_response('update.html',
         RequestContext(request, {
             'form': form,
             }))
```

## Unify "Create" and "Update"

Now you can see, the code of "update" is very similar with the "create" method, even the html template. A Django philosophy is "do not repeat yourself", so we'll consider use a common method can handle "create" and "update".

```
def foo(request, obj_id):
    form = ObjectForm(request.POST or None, request.FILES or None
        , instance= obj_id and SomeObject.objects.get(id=obj_id))

if request.method == 'POST' and form.is_valid():
    form.save()
    return HttpResponseRedirect("/")

variables = RequestContext(request, {'form': form})
    return render_to_response('create_update_template.html',variables)
```

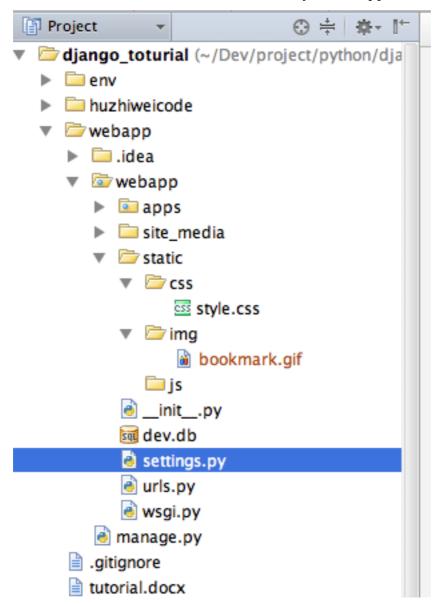
## Static and media files

### Image and other static files

To configure the static files, such as javascripts, images and css, we need to define some configuration in settings.py

```
STATICFILES_FINDERS = (
    'Django.contrib.staticfiles.finders.FileSystemFinder',
    'Django.contrib.staticfiles.finders.AppDirectoriesFinder',
)
```

Now we need to create the "static" directory in webapp directory.



You can see in static directory we defined 3 sub-directoies:

- img: store the images for the webapp
- css: store the css files
- js: store the javascript files

And we need to use these static files in our template html files (index.html). For example:

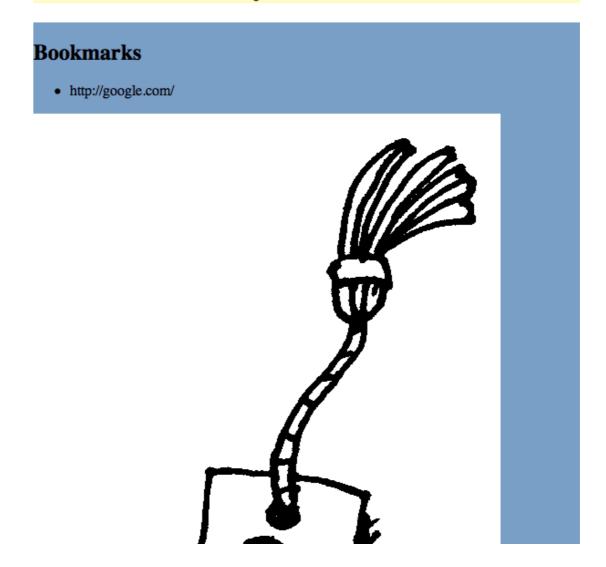
```
{% extends "base.html" %}
{% load static %}
```

You can see we should add {% load static %} tag to let template know you want to use static, and use {% static 'img/bookmark.gif' %} to load static files.

And let's see the result in browser.

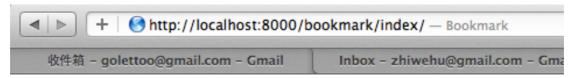


# IBookmark Share your IBookmark.



#### It works!

But please notice we just using "debug=True" mode in our webapp development now, if we set the "debug=False" in production mode, and let we check the result:



# IBookmark Share your IBookmark.

## **Bookmarks**

http://google.com/



# **Tags**

Copy right 2012.

#### Why?

Since the Django will automatically call a view to serve static files in dev mode(debug=True).

So if we set the mode to production, it cannot work. In production, we'll use such as httpd web server to handle the static files request. We need to use Django admin tool to collect our static files to /site-media/static/

\$ source env/bin/activate
(env)\$ cd webapp/
(env)\$ python manage.py collectstatic

You have requested to collect static files at the destination
location as specified in your settings.

This will overwrite existing files!
Are you sure you want to do this?

Type 'yes' to continue, or 'no' to cancel: yes
Copying

```
'/Users/jeffrey/Dev/project/python/Django_toturial/webapp/webapp/static/c ss/style.css'
......
Copying
'/Users/jeffrey/Dev/project/python/Django_toturial/env/lib/python2.7/site-pa ckages/Django/contrib/admin/static/admin/js/admin/RelatedObjectLookups.js
'
74 static files copied.
```

### **CSS file**

Let's write a css file.

```
header {
    background-color: #fffccf;
}
section {
    background-color: #7CA0C7;
    width: 80%;
    float: left;
}
aside {
    background-color: #c9dbed;
    width: 20%;
    float: left;
}
footer {
    background-color: #f6f6f6;
    clear: both;
```

And load css in html files.

```
... k href="{% static 'css/style.css' %}" rel="stylesheet">
...
```

# **Bootstrap template**

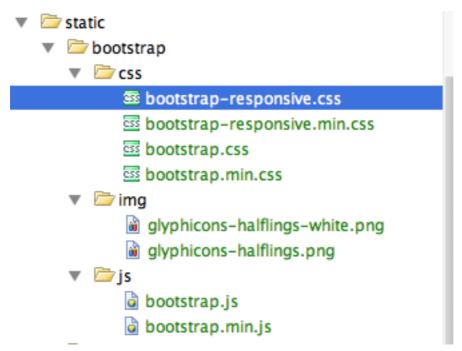
Twitter boostrap is a frontend framework which helps user build web app. Simple and flexible HTML, CSS, and Javascript for popular user interface components and interactions.

In this chapter, I'll introduce how to integrate with twitter bootstrap in our webapp.

- 1. First, we need to download the bootstrap lib from http://twitter.github.com/bootstrap/,now the version is 2.0.4
- 2. Open the bootstrap.zip, we can found the architecture is same with our static directory.



3. Copy the bootstrap directory to our static directory.



4. Create a base.html using bootstrap.

```
<!DOCTYPE HTML>
{% load static %}
{% load i18n %}
```

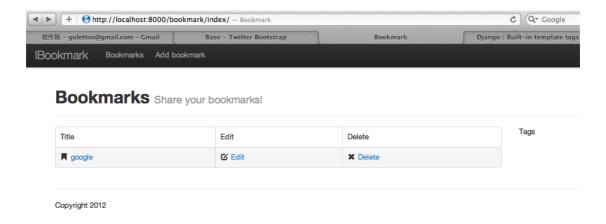
```
<html>
<head>
                                                                    %}"
    link
            href="{%
                         static
                                  'bootstrap/css/bootstrap.min.css'
rel="stylesheet">
                                                                    %}"
    k href="{% static 'bootstrap/css/bootstrap-responsive.min.css'
rel="stylesheet">
    <title>Bookmark</title>
</head>
<body>
<header>
    <div class="navbar">
        <div class="navbar-inner">
            <div class="container">
                <a class="brand" href="{% url bookmark_index %}">{% trans
"IBookmark" %}</a>
                ul class="nav">
                     <a href="{% url bookmark_index %}">{% trans
"Bookmarks" %}</a>
                     <a href="{% url bookmark_create %}">{% trans "Add"
bookmark" %}</a>
                </div>
        </div>
    </div>
</header>
<div class="container">
    <section>
        <div class="page-header">
            <h1>Bookmarks
                <small>Share your bookmarks!</small>
            </h1>
        </div>
        <div class="row">
                    class="span8">{%
            <div
                                         block
                                                  main_content
                                                                  %}{%
endblock %}</div>
            <div class="span4">Tags</div>
        </div>
    </section>
    <hr />
    <footer class="footer">
        Copyright 2012
```

```
</footer>
</div>
<!-- Le javascript
<!-- Placed at the end of the document so the pages load faster -->
<script src="{% static 'bootstrap/js/bootstrap.min.js' %}"></script>
</body>
</html>
```

#### 5. And update our index.html

```
{% extends "base.html" %}
{% load static %}
{% load i18n %}
{% block main_content %}
<thead>
   {% trans "Title" %}
   {% trans "Edit" %}
   {% trans "Delete" %}
   </thead>
   {% for bookmark in bookmarks %}
   class="icon-bookmark"></i>
      <i
                                                       <a
href="{{ bookmark.url }}">{{ bookmark.title }}</a>
      <i class="icon-edit"></i> <a href="{% url bookmark_update
bookmark.id %}">{% trans "Edit" %}</a>
      <i class="icon-remove"></i> <a href="{% url bookmark_delete
bookmark.id %}">{% trans "Delete" %}</a>
   {% empty %}
   {% trans "No data" %}
   {% endfor %}
   {% endblock %}
```

And now let's see the result.



Cool now?

And update others html templates also to extend the base.html

## 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.

https://docs.Djangoproject.com/en/dev/topics/auth/

## Creating the login page

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(username=username, password=password)
    if user is not None:
        if user.is_active:
            login(request, user)
            # Redirect to a success page.
        else:
```

# Return a 'disabled account' error message

else:

# Return an 'invalid login' error message.

## **Enabling logout functionality**

from django.contrib.auth import logout

def logout\_view(request):
 logout(request)

# Redirect to a success page.

### **User registration**

The same procedure with "Create" method of bookmark.

# 3<sup>rd</sup> party apps (django-allauth)

https://github.com/pennersr

# EC2 (apache, MySQL)

In this chapter, we'll deploy our webapp on amazon ec2 host, with apache httpd web server, and use the mysql database.

```
-- Login ec2
```

\$ ssh -i your\_ec2.pem ec2-user@your\_ec2\_ip\_or\_hostname

-- Update ec2

\$ sudo yum update -y

\$ sudo yum install gcc python-devel -y

-- Install git

\$ sudo yum install git -y

-- Check out code from git server

\$ cd /home/ec2-user

```
$ git clone git://github.com/zhiweihu/Django_toturial.git
-- Install apache httpd and mod_wsgi, mod_ssl
$ sudo yum install httpd mod_wsgi mod_ssl -y
-- Config httpd server
$ sudo vi /etc/httpd/conf.d/wsgi.conf
[Add the below content to the end of the file]
#-----#
LoadModule wsgi_module modules/mod_wsgi.so
WSGISocketPrefix run/wsgi
WSGIDaemonProcess
                                                           Django_toturial
python-path=/home/ec2-user/Django_toturial
/env/lib64/python2.6/site-packages
WSGIProcessGroup Django_toturial
#-----#
$ sudo vi /etc/httpd/conf/httpd.conf
[Change "User apache" to "User ec2-user"]
[Add the below content to the end of the file]
#----#
WSGIScriptAlias / /home/ec2-user/ES_V2/webapp/wsgi.py
Alias /site_media/ /home/ec2-user/ES_V2/webapp/site_media/
#-----#
-- Install mysql and create user/password
$ sudo yum install mysql mysql-server mysql-devel -y
$ sudo service mysqld start
$ sudo mysqladmin -u root password Your_password
$ mysql -u root -p
[Input the password, then we use the mysql console]
mysql> CREATE DATABASE tutorial db;
Query OK, 1 row affected (0.00 sec)
mysql> GRANT ALL PRIVILEGES ON tutorial_db.* TO "youruser"@"localhost"
IDENTIFIED BY "password";
Query OK, 0 rows affected (0.00 sec)
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
mysql> EXIT
Bye
```

```
-- Install virtualenv for python
$ sudo easy_install virtualenv
-- Use virtualenv to install Django
$ cd /home/ec2-user/Django_tutorial
$ virtualenv env
$ source env/bin/activate
[Now we use the virtualenv to do something, do not print the (env)$ on your
console]
(env)$ pip install -r webapp/requirement.txt
-- Settings for production
[Edit "settings_local.py" to save the db settings for production]
DATABASES = {
    'default': {
         'ENGINE': 'Django.db.backends.mysql',
         'NAME': 'tutorial_db',
         'USER': 'youruser',
         'PASSWORD': 'password',
         'HOST': ",
         'PORT': ",
    }
}
-- Sync db and collect static files
(env)$ python manage.py syncdb
(env)$ python manage.py collectstatic
-- Restart the apache http server
$ sudo /etc/init.d/httpd restart
```

# Todo

第二次迭代:

- 1. 简单介绍如何用命令行和控制台调试 model,包括如何 get objects 等queryset 的用法等等
- 2. 增加 account,message 等章节