Django tutorial

##

Quick procedures of django deployment

1. setup database server

2. get files

2. create models in models.py

3. include app in the setting under INSTALLED_APPS

4. python manage.py makemigrations my_app

5. python manage.py migrate

6. create folder my_app/static/my_app

7. create templates in my_app/templates/my_app; create style.css in static/my_app

8. create views in views.py

9. create url patterns in urls.py

8.

Admin

1. import and register created models in my_app/admin.py

##

##

Quick procedures of creating an app

1. python manage.py startapp my_app

2. create models in models.py

3. include app in the setting under INSTALLED_APPS

4. python manage.py makemigrations my_app

5. python manage.py migrate

6. create folder my_app/static/my_app

7. create templates in my_app/templates/my_app; create style.css in static/my_app

8. create views in views.py

9. create url patterns in urls.py

8.

Admin

1. import and register created models in my_app/admin.py

##

Django is dynamic website framework

Django design principle

MVC Model View Controller

MVC is a software architectural pattern for implementing user interfaces on computers。

  ## Model: Interface to your data

      ### The model is the central component of the pattern

    ### Each model is class with class variables that represent fields in Database

    ### We create columns names and datatypes and allow the database to create tables

      ## View: User Interface

          ### A view can be any output representation of information

      ## Controller: accepts input and converts it to commands for the model or view

DRY Do not repeat yourself

APP: MVC DRY Design Principle //dynamic

Default DB is SQLite3

##

$ python -m pip install django

doc

$ django-admin startproject "Name of Project"

$ django-admin startproject mysite

Configure apache with django

$ sudo apt-get install libapache2-mod-wsgi-py3

##

FILE STRUCTURE

$ cd {"Name of Project"}

there are multiple files in the folder

mysite/

manage.py

mysite/

__init__.py

settings.py

urls.py

wsgi.py

1. manage.py: A command-line utility that lets you interact with this Django project in various ways.

2. The inner mysite/ directory is the actual Python package for your project. Its name is the Python package name you will need to use to import anything inside it (e.g. mysite.urls).

3. mysite/__init__.py: An empty file that tells Python that this directory should be considered a Python package.

4. mysite/settings.py: Settings/configuration for this Django project. Django settings will tell you all about how settings work.

settings.py

A settings file is just a Python module with module-level variables.

set timezone

TIME_ZONE = 'UTC'

set database

DATABASE = {

}

Add a path for static files

STATIC_ROOT = os.path.join(BASE_DIR, 'static/')

mysite/urls.py: The URL declarations for this Django project; a table of contents of your Django-powered site.

url(r'^regex$', views.my_view_function, kwargs, name='url_name')

The url() function is passed four arguments, two required:

regex, view,

two optional:

kwargs, name

always use 'include()', except admin

When Django finds a regular expression match, Django calls the specified view function, with an HttpRequest object as the first argument and any “captured” values from the regular expression as other arguments. If the regex uses simple captures, values are passed as positional arguments; if it uses named captures, values are passed as keyword arguments.

mysite/wsgi.py: An entry-point for WSGI-compatible web servers to serve your project.

##

DEVELOPMENT SERVER

$ python manage.py runserver

Built-in Django development server

$ python manage.py runserver 8080

change to other port instead of default 8000

$ python manage.py runserver 0.0.0.0:80

accessible from outside

##

APP

A project can contain multiple apps. An app can be in multiple projects.

$ python manage.py startapp polls

That will create an app directory polls, which is laid out like this:

polls/

  __init__.py

  admin.py

  apps.py

  migrations/

    __init__.py

  models.py

  tests.py

  views.py

##

$ cd mysite/mysite

1. Create functions in views.py

2. Add url in urls.py

3. Test in browser

Basically speaking, you link the url to your python codes and when the url is called, the server run the python script and return its result.

##

Model:

0. Create models

    # Each model is represented by a class that subclasses django.db.models.Model. Each model has a number of class variables, each of which represents a database field in the model.

    # Each field is represented by an instance of a Field class – e.g., CharField for character fields and DateTimeField for datetimes. This tells Django what type of data each field holds.

The name of each Field instance (e.g. question_text or pub_date) is the field's name, in machine-friendly format. You'll use this value in your Python code, and your database will use it as the column name.

Some Field classes have required arguments. CharField, for example, requires that you give it a max_length. That's used not only in the database schema, but in validation.

A Field can also have various optional arguments; in this case, we've set the default value of votes to 0.

Primary key: If you'd like to specify a custom primary key, just specify primary_key=True on one of your fields.

Note a relationship is defined, using ForeignKey. That tells Django each Choice is related to a single Question. Django supports all the common database relationships: many-to-one, many-to-many, and one-to-one.

__str__(): It's important to add __str__() methods to your models, not only for your own convenience when dealing with the interactive prompt, but also because objects' representations are used throughout Django's automatically-generated admin.

1. Create the class that generates tables.

2. Activating models:tell our project that the polls app is installed - add reference in 'site/settings.py'

Add app to settings

3. $ python3 manage.py makemigrations polls

Migrations are entirely derived from your models file, and are essentially just a history that Django can roll through to update your database schema to match your current models.

By running makemigrations, you're telling Django that you've made some changes to your models and that you'd like the changes to be stored as a migration.

Migrations are how Django stores changes to your models (and thus your database schema) - they're just files on disk. You can read the migration for your new model if you likae; it's the file polls/migrations/0001_initial.py.

3.1 $ python manage.py sqlmigrate polls 0001

show SQL codes

Table names are automatically generated by combining the name of the app (polls) and the lowercase name of the model(One can override this.)

By convention, Django appends "_id" to the foreign key field name.

4. $ python3 manage.py migrate

The migrate command takes all the migrations that haven't been applied (Django tracks which ones are applied using a special table in your database called django_migrations) and runs them against your

database - essentially, synchronizing the changes you made to your models with the schema in the database.

Three-step guide to making model changes:

1. Change your models (in models.py).

2. Run "python manage.py makemigrations" to create migrations for those changes

3. Run "python manage.py migrate" to apply those changes to the database.

Django API: We can add data to the DB using the Python shell

1. $ python3 manage.py shell

2. Import Models : >>> from polls.models import Question, Choice

3. Display Questions : >>> Question.objects.all()

4. create a Question

5. >>> from django.utils import timezone

6. >>> q = Question(question_text="What's New?", pub_date=timezone.now())

7. Save to the DB : >>> q.save()

8. Get the questions id : >>> q.id

9. Get the questions text : >>> q.question_text

10. Change the question : >>> q.question_text = "What's Up?"

11. Save the change : >>> q.save()

12. database lookup API driven by keyword:

    >>> Question.objects.filter(id=1)

    >>> Question.objects.filter(question_text__startswith='What')

    >>> current_year = timezone.now().year

    >>> Question.objects.get(pub_date__year=current_year)

12. shortcut for primary-key exact lookups.(The following is identical to Question.objects.get(id=1).)

    >>> Question.objects.get(pk=1)

13. create choice:

```
   >>> q.choice_set.create(choice_text='Not much', votes=0)
   >>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
14. >>> q.choice_set.count()
15. delete one of the choice: >>> c.delete()



################################################################################
View:
# Each view is responsible for doing one of two things:
## returning an HttpResponse object containing the content for the requested page
## raising an exception such as Http404

# All Django wants is that HttpResponse. Or an exception.
# request is an HttpRequest object.

# Each view is represented by a function(method)
$ cd mysite/mysite
from django.http import HttpResponse
from django.shortcuts import get_object_or_404
# Opens a 404 page if get doesn't supply a result
# There’s also a get_list_or_404()
from django.shortcuts import render

from .models import Question, Choice
# Import Question and Choice from our models file

def index(request):
   latest_question_list = Question.objects.order_by('-pub_date')[:5]
   context = {
```

```
        'latest_question_list': latest_question_list,
    }
    return render(request, 'polls/index.html', context)
    # Renders a page when it is passed a template and any data required by the template


def results(request, question_id):


    question = get_object_or_404(Question, pk=question_id)
    # The get_object_or_404() function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the get() function of the model's manager.
    # pk : primary key


    return render(request, 'polls/results.html',
                  {'question': question})



from django.http import HttpResponseRedirect
# Used to avoid receiving data twice from a form if the user clicks the back button
from django.urls import reverse



# Use generic views: Less code is better
########################################
    from django.shortcuts import get_object_or_404, render
    from django.http import HttpResponseRedirect
    from django.urls import reverse
    from django.views import generic

    from .models import Choice, Question
```

```
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]


class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'


class ResultsView(generic.DetailView):
    model = Question
    template_name = 'polls/results.html'

def vote(request, question_id):
... # same as before
```

##

A common case of basic Web development: getting data from the database according to a parameter passed in the URL, loading a template and returning the rendered template. Because this is so common, Django provides a shortcut, called the "generic views" system.

ListView and DetailView. Respectively, those two views abstract the concepts of "display a list of objects" and "display a detail page for a particular type of object".

Each generic view needs to know what model it will be acting upon. This is provided using the model attribute.

The DetailView generic view expects the primary key value captured from the URL to be called "pk", so we've changed question_id to pk for the generic views.

The DetailView generic view expects the primary key value captured from the URL to be called "pk", so we've changed question_id to pk for the generic views.

By default, the DetailView generic view uses a template called <app name>/<model name>_detail.html. In our case, it would use the template "polls/question_detail.html". The template_name attribute is used to tell Django to use a specific template name instead of the autogenerated default template name.

Similarly, the ListView generic view uses a default template called <app name>/<model name>_list.html; we use template_name to tell ListView to use our existing "polls/index.html" template.

##

urls.py

polls/urls.py

##

```
from django.conf.urls import url


from . import views


app_name = 'polls'
# url namespacing


urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
```

```
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
  ]
########################################
```

?P<question_id> defines the name that will be used to identify the matched pattern(corresponding to the argument name in views)

[0-9]+ is a regular expression to match a sequence of digits (i.e., a number).

Namespacing URL names

In one django apps, there might be more than one app. Different apps might use same url name like 'detail'. How does one make it so that Django knows which app view to create for a url when using the {% url %} template tag?

The answer is to add namespaces to your URLconf. In the polls/urls.py file, go ahead and add an app_name to set the application namespace.

##

Admin

create an admin user

0. $ python3 manage.py createsuperuser

1. login at url/admin/

1.Modify polls/admin.py to register the model so that you can manage under admin page

Tell admin that our poll system has an admin interface

```
  from django.contrib import admin

  from .models import Question
```

- Import Question

admin.site.register(Question)

- Register Question(the model) for showing in admin

##

Template

0. Create new folder templates (under polls), then create a new folder "polls" under templates. Django will look for templates in there.

Within the templates directory you have just created, create another directory called polls, and within that create a file called index.html.

Your template should be at polls/templates/polls/index.html.

You can refer to this template within Django simply as polls/index.html.

Template namespacing

Now we might be able to get away with putting our templates directly in polls/templates but it would actually be a bad idea. Django will choose the first template it finds whose name matches, and if you had a template with the same name in a different application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by namespacing them. That is, by putting those templates inside another directory named for the application itself.

1. Modify views.py to render templates.

We can user the the shortcut: render(), so that we no longer need to import loader and HttpResponse.

2. Template system

2.1 The template system uses dot-lookup syntax to access variable attributes.

First Django does a dictionary lookup on the object question.

Failing that, it tries an attribute lookup。

Failing that again, it would've tried a list-index lookup.

2.2 Removing hardcoded URLs in templates

The problem with hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with a lot of templates.

However, since you defined the name argument in the url() functions in the polls.urls module, you can remove a reliance on specific URL paths defined in your url configurations by using the {% url %} template tag:

{{ question.question_text }}

The way this works is by looking up the URL definition as specified in the polls.urls module(the URL name of 'detail')

In this way, you only need to modify urls.py.

3. Django template language

(1) Variables

A variable outputs a value from the context, which is a dict-like object mapping keys to values.

Variables are surrounded by {{ and }} like this:

My first name is {{ first_name }}. My last name is {{ last_name }}.

With a context of {'first_name': 'John', 'last_name': 'Doe'}, this template renders to:

My first name is John. My last name is Doe.

Dictionary lookup, attribute lookup and list-index lookups are implemented with a dot notation:

{{ my_dict.key }}

{{ my_object.attribute }}

{{ my_list.0 }}

If a variable resolves to a callable, the template system will call it with no arguments and use its result instead of the callable.

(2) Tags

Tags provide arbitrary logic in the rendering process.

This definition is deliberately vague. For example, a tag can output content, serve as a control structure e.g. an “if” statement or a “for” loop, grab content from a database, or even enable access to other template tags.

Tags are surrounded by {% and %} like this:

```
{% csrf_token %}
```

Most tags accept arguments:

```
{% cycle 'odd' 'even' %}
```

Some tags require beginning and ending tags:

```
{% if user.is_authenticated %}Hello, {{ user.username }}.{% endif %}
```

(3) Filters

(4) Comments

Comments look like this:

```
{# this won't be rendered #}
```

{% comment %} tag provides multi-line comments. {% comment %}

Template example:

```
<!-- 5 If a list of questions is available create
an unordered list of the questions or print
that none are available -->
{% if latest_question_list %}
   <ul>
      {% for question in latest_question_list %}
         <li>
            <!-- 5 url defines directory to open based
             on using the namespace defined in polls/urls.py
             and the url defined in sampsite/urls.py -->
            <a href="{% url 'polls:detail' question.id %}">{{question.question_text}}</a>
         </li>
      {% endfor %}
   </ul>
{% else %}
   <p>No polls are available</p>
{% endif %}

<!-- 5 Back to polls/views.py to update index -->

#################################################################################
# form
```

https://www.w3schools.com/html/html_forms.asp

submit button: <input type="submit"> defines a button for submitting the form data to a form-handler.

The form-handler is typically a server page with a script for processing input data.

The form-handler is specified in the form's action attribute

The action attribute defines the action to be performed when the form is submitted.

The default method when submitting form data is GET.

However, when GET is used, the submitted form data will be visible in the page address field

Always use POST if the form data contains sensitive or personal information. The POST method does not display the submitted form data in the page address field.

polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

Since we’re creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don’t have to worry too hard, because Django comes with a very easy-to-use system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the {% csrf_token %} template tag.

Whenever you create a form that alters data server-side, use method="post".

##

##

static files

Create a directory called static in your polls directory. Django will look for static files there, similarly to how Django finds templates inside polls/templates/.

Within the static directory you have just created, create another directory called polls and within that create a file called style.css. In other words, your stylesheet should be at polls/static/polls/style.css. Because of how the AppDirectoriesFinder staticfile finder works, you can refer to this static file in Django simply as polls/style.css, similar to how you reference the path for templates.

Static file namespacing: Django will choose the first static file it finds whose name matches, and if you had a static file with the same name in a different application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by namespacing them. That is, by putting those static files inside another directory named for the application itself.

add the following at the top of polls/templates/polls/index.html:

{% load static %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />

The {% static %} template tag generates the absolute URL of static files

##

django remove app

python manage.py sqlclear my_app

my_site/settings.py # modify INSTALLED_APPS

##

$ python manage.py startapp blog

get block in a database and post it to page

$ cd blog

__init__.py

model.py

Model: Interface to your data

we define all the data structures we need in this file. and run a command then all the databases are created for you

views.py

View: User Interface

We write the code that actually generates the webpages.

$ vim blog/models.py

To create a database table, we write a class

class posts(models.Model):

inherit from this class

```
author = models.CharField(max_length = 50)
# This is type field of strings
title = models.CharField(max_length = 100)
body_text = models.TextField()
timestamp = models.DateTimeField()
```

mysql

$ mysql -u root -p

```
UPDATE mysql.user SET Password=PASSWORD('skynet');
FLUSH PRIVILEGES;

CREATE DATABASE myblog;
```

$ python3 manage.py runserver

$ python3 manage.py migrate

THis will update the database, can only add new ones, cannot alter existing ones.

Run this every time you change the models

##

Apache setup:

#Add the following to apache2.conf/httpd.conf

WSGIScriptAlias /django /home/ubuntu/steven_site/steven_site/wsgi.py

WSGIPythonPath /home/ubuntu/steven_site

<Directory /home/ubuntu/steven_site/steven_site>

<Files wsgi.py>

Require all granted

</Files>

</Directory>