**Recommended Pre-requisites**

Programmers considering attending this Django training course should have a basic understanding of the Python language, and a basic understanding of Object Oriented programming and relational database concepts (no SQL knowledge is required).

**Course Outline/Agenda**

* Introduction
  + About Django
  + Django Components
* Installing & Configuring Django Components
  + Django Pre-Requisites
  + Downloading & Installing Django
  + Choosing a Database
  + Creating a New Project
* Generating Simple Django Views
  + About View Functions
  + Using Django's HttpResponse Class
  + Understanding HttpRequest Objects
  + Using QueryDict Objects
* Configuring URLconf's
  + About URLconf
  + Regular Expressions
  + Expression Examples
  + Simple URLConf Examples
  + Using Multiple URLConf's
  + Passing URL Arguments
* Django Templates
  + About Templates
  + Template Fundamentals
  + Creating Template Objects
    - Loading Template Files
  + Filling in Template Content (Context Objects)
  + Template Filters
  + Template Tags
  + More on For Loops
  + Template Inheritance
  + Easy Rendering of Templates
  + RequestContext Processors
  + Global Context Processors
* HTML Forms with Forms (formerly newforms)
  + The Forms Module
  + Creating the Form
    - Generating Output From the Form
  + Customizing Field Parameters
  + Processing Form Data
    - Custom Form Field Validation
  + Generating Custom Field Errors
  + Customizing Form Output
* Database Models with Django
  + About Database Models
  + Configuring Django for Database Access
  + Understanding Django Apps
  + About Django Models
  + Defining Django Models
  + Understanding Model Fields & Options
  + Table Naming Conventions
  + Creating A Django Model
  + Adding the App to Your Project
  + Validating the App
  + Generating & Reviewing the SQL
  + Adding Data to the Model
  + Primary Keys and the Model
  + Simple Data Retrieval Using a Model
  + Understanding QuerySets
  + Applying Filters
  + Specifying Field Lookups
  + Lookup Types
  + Slicing QuerySets
  + Specifying Ordering in QuerySets
  + Common QuerySet Methods
  + Deleting Records
  + Managing Related Records
  + Retrieving Related Records
  + Using Q Objects
  + Creating Forms from Models
* Using the Django Admin Interface
  + Enabling the Admin Interface
  + Creating an Admin User
* Access Control with Sessions and Users
  + Cookies & Django
  + The Django Session Framework
  + Sessions in Views
  + Session Tuning
  + Installing Django User Authentication
  + Using Authentication in Views
    - Login and Logout
  + Building your Own Login/Logout Views
  + Authentication Decorators
  + Adding & Deactivating Users
  + Asynchronous Messaging
  + Managing Permissions
* Generic Views
  + Simple Generic Views
  + Using Generic Redirects
  + Other Generic Views
  + Create/Update/Delete Generic views
* Creating Syndication Feeds
  + About Syndication
* AJAX Integration with Django
  + AJAX with Django
* Data Caching for Performance
  + Data Caching
  + Setting up Per-View Caching
  + Site Caching
* Preventing Cross Site Request Forgery
  + Cross Site Request Forgery
* Django's Email Functionality
  + Configuring Mail Settings
  + Sending Email
  + Other Email Functions
* Deploying Django Applications
  + Deploying to Apache
* Using ReportLab for PDF Generation
  + About ReportLab Toolkit
* Questions?
  + Where to Get Answers/Next Steps

Create User :

$ django-admin startproject mysite

Let’s look at what startproject created:

mysite/

manage.py

mysite/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

These files are:

* The outer mysite/ root 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 and manage.py.
* The inner mysite/ 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. mysite.urls).
* mysite/\_\_init\_\_.py: An empty file that tells Python that this directory should be considered a Python package. If you’re a Python beginner, read.
* mysite/settings.py: Settings/configuration for this Django project. Django settings will tell you all about how settings work.
* mysite/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.
* mysite/wsgi.py: An entry-point for WSGI-compatible web servers to serve your project. See How to deploy with WSGI for more details.

## he development server¶

Let’s verify your Django project works. Change into the outer mysite directory, if you haven’t already, and run the following commands:

$ python manage.py runserver

Changing the port

By default, the runserver command starts the development server on the internal IP at port 8000.

If you want to change the server’s port, pass it as a command-line argument. For instance, this command starts the server on port 8080:

$ python manage.py runserver 8080

If you want to change the server’s IP, pass it along with the port. For example, to listen on all available public IPs (which is useful if you are running Vagrant or want to show off your work on other computers on the network), use:

$ python manage.py runserver 0:8000

**0** is a shortcut for **0.0.0.0**. Full docs for the development server can be found in the runserver reference.

## Write your first view¶

Let’s write the first view. Open the file polls/views.py and put the following Python code in it:

polls/views.py

from django.http import HttpResponse

def index(request):

return HttpResponse("Hello, world. You're at the polls index.")

In the polls/urls.py file include the following code:

polls/urls.py

from django.conf.urls import url

from . import views

urlpatterns = [

url(r'^$', views.index, name='index'),

]

The next step is to point the root URLconf at the polls.urls module. In mysite/urls.py, add an import for django.conf.urls.include and insert an include() in the urlpatterns list, so you have:

mysite/urls.py

from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

url(r'^polls/', include('polls.urls')),

url(r'^admin/', admin.site.urls),

]

**Database setup¶**

Now, open up mysite/settings.py. It’s a normal Python module with module-level variables representing Django settings.

By default, the configuration uses SQLite. If you’re new to databases, or you’re just interested in trying Django, this is the easiest choice. SQLite is included in Python, so you won’t need to install anything else to support your database. When starting your first real project, however, you may want to use a more scalable database like PostgreSQL, to avoid database-switching headaches down the road.

If you wish to use another database, install the appropriate database bindings and change the following keys in the DATABASES 'default' item to match your database connection settings:

* ENGINE – Either 'django.db.backends.sqlite3', 'django.db.backends.postgresql', 'django.db.backends.mysql', or 'django.db.backends.oracle'. Other backends are also available.
* NAME – The name of your database. If you’re using SQLite, the database will be a file on your computer; in that case, NAME should be the full absolute path, including filename, of that file. The default value, os.path.join(BASE\_DIR, 'db.sqlite3'), will store the file in your project directory.

**Update database**

$ python manage.py migrate

## Creating models¶

Now we’ll define your models – essentially, your database layout, with additional metadata.

In our simple poll app, we’ll create two models: Question and Choice. A Question has a question and a publication date. A Choice has two fields: the text of the choice and a vote tally. Each Choice is associated with a Question.

These concepts are represented by simple Python classes. Edit the polls/models.py file so it looks like this:

polls/models.py

from django.db import models

class Question(models.Model):

question\_text = models.CharField(max\_length=200)

pub\_date = models.DateTimeField('date published')

class Choice(models.Model):

question = models.ForeignKey(Question, on\_delete=models.CASCADE)

choice\_text = models.CharField(max\_length=200)

votes = models.IntegerField(default=0)

The code is straightforward. 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.

You can use an optional first positional argument to a Field to designate a human-readable name. That’s used in a couple of introspective parts of Django, and it doubles as documentation. If this field isn’t provided, Django will use the machine-readable name. In this example, we’ve only defined a human-readable name for Question.pub\_date. For all other fields in this model, the field’s machine-readable name will suffice as its human-readable 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, as we’ll soon see.

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

Finally, 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.

**Activating models¶**

That small bit of model code gives Django a lot of information. With it, Django is able to:

* Create a database schema (CREATE TABLE statements) for this app.
* Create a Python database-access API for accessing Question and Choice objects.

But first we need to tell our project that the polls app is installed.

To include the app in our project, we need to add a reference to its configuration class in the INSTALLED\_APPS setting. The PollsConfig class is in the polls/apps.py file, so its dotted path is 'polls.apps.PollsConfig'. Edit the mysite/settings.py file and add that dotted path to the INSTALLED\_APPS setting. It’ll look like this:

mysite/settings.py

INSTALLED\_APPS = [

'polls.apps.PollsConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Now Django knows to include the polls app. Let’s run another command:

$ python manage.py makemigrations polls

You should see something similar to the following:

Migrations for 'polls':

polls/migrations/0001\_initial.py:

- Create model Choice

- Create model Question

- Add field question to choice

$ python manage.py sqlmigrate polls 0001

You should see something similar to the following (we’ve reformatted it for readability):

BEGIN;

--

-- Create model Choice

--

CREATE TABLE "polls\_choice" (

"id" serial NOT NULL PRIMARY KEY,

"choice\_text" varchar(200) NOT NULL,

"votes" integer NOT NULL

);

--

-- Create model Question

--

CREATE TABLE "polls\_question" (

"id" serial NOT NULL PRIMARY KEY,

"question\_text" varchar(200) NOT NULL,

"pub\_date" timestamp with time zone NOT NULL

);

--

-- Add field question to choice

--

ALTER TABLE "polls\_choice" ADD COLUMN "question\_id" integer NOT NULL;

ALTER TABLE "polls\_choice" ALTER COLUMN "question\_id" DROP DEFAULT;

CREATE INDEX "polls\_choice\_7aa0f6ee" ON "polls\_choice" ("question\_id");

ALTER TABLE "polls\_choice"

ADD CONSTRAINT "polls\_choice\_question\_id\_246c99a640fbbd72\_fk\_polls\_question\_id"

FOREIGN KEY ("question\_id")

REFERENCES "polls\_question" ("id")

DEFERRABLE INITIALLY DEFERRED;

COMMIT;

$ python manage.py migrate

Operations to perform:

Apply all migrations: admin, auth, contenttypes, polls, sessions

Running migrations:

Rendering model states... DONE

Applying polls.0001\_initial... OK

polls/models.py

from django.db import models

from django.utils.encoding import python\_2\_unicode\_compatible

@python\_2\_unicode\_compatible # only if you need to support Python 2

class Question(models.Model):

# ...

def \_\_str\_\_(self):

return self.question\_text

@python\_2\_unicode\_compatible # only if you need to support Python 2

class Choice(models.Model):

# ...

def \_\_str\_\_(self):

return self.choice\_text

polls/models.py

import datetime

from django.db import models

from django.utils import timezone

class Question(models.Model):

# ...

def was\_published\_recently(self):

return self.pub\_date >= timezone.now() - datetime.timedelta(days=1)

### Creating an admin user¶

First we’ll need to create a user who can login to the admin site. Run the following command:

$ python manage.py createsuperuser

Enter your desired username and press enter.

Username: admin

You will then be prompted for your desired email address:

Email address: admin@example.com

## Writing more views¶

Now let’s add a few more views to polls/views.py. These views are slightly different, because they take an argument:

polls/views.py

def detail(request, question\_id):

return HttpResponse("You're looking at question %s." % question\_id)

def results(request, question\_id):

response = "You're looking at the results of question %s."

return HttpResponse(response % question\_id)

def vote(request, question\_id):

return HttpResponse("You're voting on question %s." % question\_id)

Wire these new views into the polls.urls module by adding the following url() calls:

polls/urls.py

from django.conf.urls import url

from . import views

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'),

]

polls/views.py

from django.http import HttpResponse

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

output = ', '.join([q.question\_text for q in latest\_question\_list])

return HttpResponse(output)

# Leave the rest of the views (detail, results, vote) unchanged

Put the following code in that template:

polls/templates/polls/index.html

{% if latest\_question\_list %}

<ul>

{% for question in latest\_question\_list %}

<li><a href="/polls/{{ question.id }}/">{{ question.question\_text }}</a></li>

{% endfor %}

</ul>

{% else %}

<p>No polls are available.</p>

{% endif %}

Now let’s update our index view in polls/views.py to use the template:

polls/views.py

from django.http import HttpResponse

from django.template import loader

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

template = loader.get\_template('polls/index.html')

context = {

'latest\_question\_list': latest\_question\_list,

}

return HttpResponse(template.render(context, request))

### A shortcut: render()¶

It’s a very common idiom to load a template, fill a context and return an HttpResponse object with the result of the rendered template. Django provides a shortcut. Here’s the full index() view, rewritten:

polls/views.py

from django.shortcuts import render

from .models import Question

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)

## Raising a 404 error¶

Now, let’s tackle the question detail view – the page that displays the question text for a given poll. Here’s the view:

polls/views.py

from django.http import Http404

from django.shortcuts import render

from .models import Question

# ...

def detail(request, question\_id):

try:

question = Question.objects.get(pk=question\_id)

except Question.DoesNotExist:

raise Http404("Question does not exist")

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

The new concept here: The view raises the Http404 exception if a question with the requested ID doesn’t exist.

We’ll discuss what you could put in that polls/detail.html template a bit later, but if you’d like to quickly get the above example working, a file containing just:

polls/templates/polls/detail.html

{{ question }}

### A shortcut: get\_object\_or\_404()¶

It’s a very common idiom to use get() and raise Http404 if the object doesn’t exist. Django provides a shortcut. Here’s the detail() view, rewritten:

polls/views.py

from django.shortcuts import get\_object\_or\_404, render

from .models import Question

# ...

def detail(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

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

## Use the template system¶

Back to the detail() view for our poll application. Given the context variable question, here’s what the polls/detail.html template might look like:

polls/templates/polls/detail.html

<h1>{{ question.question\_text }}</h1>

<ul>

{% for choice in question.choice\_set.all %}

<li>{{ choice.choice\_text }}</li>

{% endfor %}

</ul>

url

from django.conf.urls import url

from . import views

app\_name = 'polls'

urlpatterns = [

url(r'^$', views.index, name='index'),

url(r'^(?P<question\_id>[0-9]+)/$', views.detail, name='detail'),

url(r'^(?P<question\_id>[0-9]+)/results/$', views.results, name='results'),

url(r'^(?P<question\_id>[0-9]+)/vote/$', views.vote, name='vote'),

]