# Writing your first Django app, part 1

<https://docs.djangoproject.com/en/dev/intro/tutorial01/>

We’ll assume you have [Django installed](https://docs.djangoproject.com/en/dev/intro/install/) already. You can tell Django is installed and which version by running the following command:

python -c "import django; print(django.get\_version())"

If Django is installed, you should see the version of your installation. If it isn’t, you’ll get an error telling “No module named django”.

## Creating a project

If this is your first time using Django, you’ll have to take care of some initial setup. Namely, you’ll need to auto-generate some code that establishes a Django [project](https://docs.djangoproject.com/en/dev/glossary/#term-project) – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, cd into a directory where you’d like to store your code, then run the command:

django-admin.py startproject mysite

This will create a mysite directory in your current directory. Let’s look at what [startproject](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-startproject) created:

mysite/

manage.py

mysite/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

**Doesn’t match what you see?**

The default project layout recently changed. If you’re seeing a “flat” layout (with no inner mysite/ directory), you’re probably using a version of Django that doesn’t match this tutorial version. You’ll want to either switch to the older tutorial or the newer Django version.

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.py and manage.py](https://docs.djangoproject.com/en/dev/ref/django-admin/).
* 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. (Read[more about packages](http://docs.python.org/tutorial/modules.html#packages) in the official Python docs if you’re a Python beginner.)
* mysite/settings.py: Settings/configuration for this Django project. [Django settings](https://docs.djangoproject.com/en/dev/topics/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](https://docs.djangoproject.com/en/dev/topics/http/urls/).
* mysite/wsgi.py: An entry-point for WSGI-compatible web servers to serve your project.

### The development server

Let’s verify this worked. Change into the outer mysite directory, if you haven’t already, and run the command python manage.py runserver. You’ll see the following output on the command line:

Validating models...

0 errors found

July 23, 2013 - 15:50:53

Django version 1.7, using settings 'mysite.settings'

Starting development server at <http://127.0.0.1:8000/>

Quit the server with CONTROL-C.

You’ve started the Django development server, a lightweight Web server written purely in Python. Now that the server’s running, visit <http://127.0.0.1:8000/> with your Web browser. You’ll see a “Welcome to Django” page, in pleasant, light-blue pastel. It worked!

**Changing the port**

By default, the [runserver](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-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. So to listen on all public IPs (useful if you want to show off your work on other computers), use:

python manage.py runserver 0.0.0.0:8000

### Database setup

mysite/settings.py is a normal Python module with module-level variables representing Django settings. By default, the configuration uses SQLite. To use another database, install the appropriate [database bindings](https://docs.djangoproject.com/en/dev/topics/install/#database-installation), and change the following keys in the [DATABASES](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-DATABASES) 'default' item to match your database connection settings:

* [ENGINE](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-DATABASE-ENGINE) – Either 'django.db.backends.sqlite3', 'django.db.backends.postgresql\_psycopg2', 'django.db.backends.mysql', or 'django.db.backends.oracle'. Other backends are [also available](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-DATABASE-ENGINE).
* [NAME](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-NAME) – The name of your database. If you’re using SQLite, the database will be a file on your computer; in that case, [NAME](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-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.

If you are not using SQLite as your database, additional settings like [USER](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-USER), [PASSWORD](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-PASSWORD), [HOST](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-HOST) must be added.

**Note**

If you’re using PostgreSQL or MySQL, make sure you’ve created a database by this point. Do that with “CREATE DATABASE database\_name;” within your database’s interactive prompt. If you’re using SQLite, you don’t need to create anything beforehand - the database file will be created automatically when it is needed.

While you’re editing mysite/settings.py, set [TIME\_ZONE](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-TIME_ZONE) to your time zone.

Also, note the [INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS) setting at the top of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects, and you can package and distribute them for use by others in their projects.

By default, [INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS) contains the following apps, all of which come with Django:

* [django.contrib.admin](https://docs.djangoproject.com/en/dev/ref/contrib/admin/#module-django.contrib.admin) – The admin site. You’ll use it in [part 2 of this tutorial](https://docs.djangoproject.com/en/dev/intro/tutorial02/).
* [django.contrib.auth](https://docs.djangoproject.com/en/dev/topics/auth/#module-django.contrib.auth) – An authentication system.
* [django.contrib.contenttypes](https://docs.djangoproject.com/en/dev/ref/contrib/contenttypes/#module-django.contrib.contenttypes) – A framework for content types.
* [django.contrib.sessions](https://docs.djangoproject.com/en/dev/topics/http/sessions/#module-django.contrib.sessions) – A session framework.
* [django.contrib.messages](https://docs.djangoproject.com/en/dev/ref/contrib/messages/#module-django.contrib.messages) – A messaging framework.
* [django.contrib.staticfiles](https://docs.djangoproject.com/en/dev/ref/contrib/staticfiles/#module-django.contrib.staticfiles) – A framework for managing static files.

These applications are included by default as a convenience for the common case. Some of these applications makes use of at least one database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

python manage.py syncdb

The [syncdb](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-syncdb) command looks at the [INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS) setting and creates any necessary database tables according to the database settings in your mysite/settings.py file. You’ll see a message for each database table it creates, and you’ll get a prompt asking you if you’d like to create a super user account for the authentication system. Go ahead and do that. If you’re interested, run the command-line client for your database and type \dt (PostgreSQL), SHOW TABLES; (MySQL), or.schema (SQLite) to display the tables Django created.

**For the minimalists**

The default applications are included for the common case, but not everybody needs them. If you don’t need any or all of them, feel free to comment-out or delete the appropriate line(s) from [INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS) before running [syncdb](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-syncdb). The [syncdb](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-syncdb) command will only create tables for apps in [INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS).

## Creating models

Now that your environment – a “project” – is set up, you’re set to start doing work. Each application you write in Django consists of a Python package that follows a certain convention. Django comes with a utility that automatically generates the basic directory structure of an app.

**Projects vs. apps**

An app is a Web application that does something (a set of functions) – e.g., a Weblog system, a database of public records or a simple poll app. A project is a collection of configuration and apps for a particular Web site. A project can contain multiple apps. An app can be in multiple projects.

Your apps can live anywhere on your [Python path](http://docs.python.org/tutorial/modules.html#the-module-search-path). In this tutorial, we’ll create our poll app right next to your manage.py file so that it can be imported as its own top-level module, rather than a submodule of mysite.

To create your app, make sure you’re in the same directory as manage.py and type this command:

python manage.py startapp polls

That’ll create a directory polls, which is laid out like this:

polls/

\_\_init\_\_.py

admin.py

models.py

tests.py

views.py

**Philosophy**

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. The goal is to define your data model in one place and automatically derive things from it. In our simple poll app, we’ll create two models: Poll and Choice. A Poll 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 Poll.

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

from django.db import models

**class** **Poll**(models**.**Model):

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

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

**class** **Choice**(models**.**Model):

poll **=** models**.**ForeignKey(Poll)

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

votes **=** models**.**IntegerField(default**=**0)

Each model is represented by a class that subclasses [django.db.models.Model](https://docs.djangoproject.com/en/dev/ref/models/instances/" \l "django.db.models.Model" \o "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](https://docs.djangoproject.com/en/dev/howto/custom-model-fields/#django.db.models.Field) class – e.g., [CharField](https://docs.djangoproject.com/en/dev/ref/models/fields/" \l "django.db.models.CharField" \o "django.db.models.CharField) for character fields and [DateTimeField](https://docs.djangoproject.com/en/dev/ref/models/fields/" \l "django.db.models.DateTimeField" \o "django.db.models.DateTimeField) for datetimes. This tells Django what type of data each field holds.

## Meta options[¶](https://docs.djangoproject.com/en/dev/topics/db/models/#meta-options)

Give your model metadata by using an inner class Meta, like so:

from django.db import models

**class** **Ox**(models**.**Model):

horn\_length **=** models**.**IntegerField()

**class** **Meta**:

db\_table **=** u"Ox"

Model metadata is “anything that’s not a field”, such as ordering options ([ordering](https://docs.djangoproject.com/en/dev/ref/models/options/#django.db.models.Options.ordering)), database table name ([db\_table](https://docs.djangoproject.com/en/dev/ref/models/options/" \l "django.db.models.Options.db_table" \o "django.db.models.Options.db_table)). None are required, and adding class Meta to a model is completely optional. In this example, the table named ‘Ox’ created in database. Exploit ‘*show tables;*’ command to see the presence of this table.

For more information about model, go to <https://docs.djangoproject.com/en/dev/topics/db/models/>

## Activating models

That small bit of model code gives Django a lot of information. With it, Django is able to: (1) Create a database schema (CREATE TABLE statements) for this app. (2) Create a Python database-access API for accessing Poll and Choice objects; But first we need to tell our project that the polls app is installed.

**Philosophy**

Django apps are “pluggable”: You can use an app in multiple projects, and you can distribute apps, because they don’t have to be tied to a given Django installation. Edit the mysite/settings.py file again, and change the [INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS) setting to include the string 'polls'. So it’ll look like this:

INSTALLED\_APPS **=** (

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'polls',

)

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

python manage.py sql polls

You should see something similar to the following (the CREATE TABLE SQL statements for the polls app):

**BEGIN**;

**CREATE** **TABLE** "polls\_poll" (

"id" integer **NOT** **NULL** **PRIMARY** **KEY**,

"question" varchar(200) **NOT** **NULL**,

"pub\_date" datetime **NOT** **NULL**

);

**CREATE** **TABLE** "polls\_choice" (

"id" integer **NOT** **NULL** **PRIMARY** **KEY**,

"poll\_id" integer **NOT** **NULL** **REFERENCES** "polls\_poll" ("id"),

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

"votes" integer **NOT** **NULL**

);

**COMMIT**;

Note: (1) The exact output will vary depending on the database you are using. The example above is generated for SQLite. (2) Table names are automatically generated by combining the name of the app (polls) and the lowercase name of the model – poll and choice. (You can override this behavior.) (3) Primary keys (IDs) are added automatically. (You can override this, too.) (4) By convention, Django appends "\_id" to the foreign key field name. (Yes, you can override this, as well.) (5) The foreign key relationship is made explicit by a REFERENCES statement. (6) It’s tailored to the database you’re using, so database-specific field types such as auto\_increment (MySQL), serial(PostgreSQL), or integer primary key (SQLite) are handled for you automatically. Same goes for quoting of field names – e.g., using double quotes or single quotes. (7) The [sql](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-sql) command doesn’t actually run the SQL in your database - it just prints it to the screen so that you can see what SQL Django thinks is required. If you wanted to, you could copy and paste this SQL into your database prompt. However, as we will see shortly, Django provides an easier way of committing the SQL to the database.

If you’re interested, also run the following commands:

* [python manage.py validate](https://docs.djangoproject.com/en/dev/ref/django-admin/#django-admin-validate) – Checks for any errors in the construction of your models.
* [python manage.py sqlcustom polls](https://docs.djangoproject.com/en/dev/ref/django-admin/#django-admin-sqlcustom) – Outputs any [custom SQL statements](https://docs.djangoproject.com/en/dev/howto/initial-data/#initial-sql) (such as table modifications or constraints) that are defined for the application.
* [python manage.py sqlclear polls](https://docs.djangoproject.com/en/dev/ref/django-admin/#django-admin-sqlclear) – Outputs the necessary DROP TABLE statements for this app, according to which tables already exist in your database (if any).
* [python manage.py sqlindexes polls](https://docs.djangoproject.com/en/dev/ref/django-admin/#django-admin-sqlindexes) – Outputs the CREATE INDEX statements for this app.
* [python manage.py sqlall polls](https://docs.djangoproject.com/en/dev/ref/django-admin/#django-admin-sqlall) – A combination of all the SQL from the [sql](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-sql), [sqlcustom](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-sqlcustom), and [sqlindexes](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-sqlindexes) commands.

Now, run [syncdb](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-syncdb) again to create those model tables in your database:

python manage.py syncdb

The [syncdb](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-syncdb) command runs the SQL from [sqlall](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-sqlall) on your database for all apps in [INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS) that don’t already exist in your database. This creates all the tables, initial data and indexes for any apps you’ve added to your project since the last time you ran syncdb. [syncdb](https://docs.djangoproject.com/en/dev/ref/django-admin/" \l "django-admin-syncdb) can be called as often as you like, and it will only ever create the tables that don’t exist.

## Playing with the API

To invoke the Python shell, use this command:

python manage.py shell

We’re using this instead of simply typing “python”, because manage.py sets the DJANGO\_SETTINGS\_MODULE environment variable, which gives Django the Python import path to your mysite/settings.py file. For more information on all of this, see the [django-admin.py documentation](https://docs.djangoproject.com/en/dev/ref/django-admin/).

Once you’re in the shell, explore the [database API](https://docs.djangoproject.com/en/dev/topics/db/queries/):

>>> from polls.models import Poll, Choice *# Import the model classes we just wrote.*

# No polls are in the system yet.

>>> Poll**.**objects**.**all()

[]

# Create a new Poll. Support for time zones is enabled in the default settings file, so Django expects a datetime

# with tzinfo for pub\_date. Use timezone.now() instead of datetime.datetime.now().

>>> from django.utils import timezone

>>> p **=** Poll(question**=**"What's new?", pub\_date**=**timezone**.**now())

# Save the object into the database. You have to call save() explicitly.

>>> p**.**save()

# Now it has an ID. Note that this might say "1L" instead of "1", depending

# on which database you're using. That's no biggie; it just means your

# database backend prefers to return integers as Python long integer objects.

>>> p**.**id

1

# Access database columns via Python attributes.

>>> p**.**question

"What's new?"

>>> p**.**pub\_date

datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().

>>> p**.**question **=** "What's up?"

>>> p**.**save()

# objects.all() displays all the polls in the database.

>>> Poll**.**objects**.**all()

[<Poll: Poll object>]

Let’s fix that by editing the polls model (in the polls/models.py file) and adding a [\_\_unicode\_\_()](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model.__unicode__) method to both Poll and Choice. On Python 3, simply replace\_\_unicode\_\_ by \_\_str\_\_ in the following example:

from django.db import models

**class** **Poll**(models**.**Model):

*# ...*

**def** **\_\_unicode\_\_**(self): *# Python 3: def \_\_str\_\_(self):*

**return** self**.**question

**class** **Choice**(models**.**Model):

*# ...*

**def** **\_\_unicode\_\_**(self): *# Python 3: def \_\_str\_\_(self):*

**return** self**.**choice\_text

It’s important to add [\_\_unicode\_\_()](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model.__unicode__) methods (or [\_\_str\_\_()](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model.__str__) on Python 3) to your models, not only for your own sanity when dealing with the interactive prompt, but also because objects’ representations are used throughout Django’s automatically-generated admin.

**\_\_unicode\_\_ or \_\_str\_\_?**

If you’re familiar with Python 2, you might be in the habit of adding [\_\_str\_\_()](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model.__str__) methods to your classes, not[\_\_unicode\_\_()](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model.__unicode__) methods. We use [\_\_unicode\_\_()](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model.__unicode__) here because Django models deal with Unicode by default. All data stored in your database is converted to Unicode when it’s returned.

Django models have a default [\_\_str\_\_()](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model.__str__) method that calls [\_\_unicode\_\_()](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model.__unicode__) and converts the result to a UTF-8 bytestring. This means that unicode(p) will return a Unicode string, and str(p) will return a normal string, with characters encoded as UTF-8. Note these are normal Python methods. Let’s add a custom method, just for demonstration:

# Writing your first Django app, part 2

## <https://docs.djangoproject.com/en/dev/intro/tutorial02/>

## Start the development server

The Django admin site is activated by default. Let’s start the development server and explore it.

Recall from Tutorial 1 that you start the development server like so:

python manage.py runserver

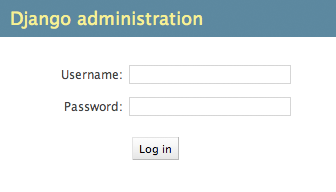
## Exploit the Admin account on Web page

## Uncomment *django.contrib.admin* in *settings.py*.

## Uncomment *from django.contrib import admin* and *admin.autodiscover()* in *urls.py*.

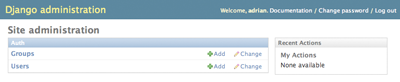
## Uncomment *url(r‘^admin/’, include(admin.site.urls))*, in *urls.py*.

1. Open a Web browser and type <http://127.0.0.1:8000/admin/>. You should see the admin’s login screen:



## Enter the admin site

Now, try logging in. You created a super user account in the first part of this tutorial, remember? If you didn’t create one or forgot the password you can [create another one](https://docs.djangoproject.com/en/dev/topics/auth/default/#topics-auth-creating-superusers). You should see the Django admin index page:



You should see a few types of editable content: groups and users. They are provided by [django.contrib.auth](https://docs.djangoproject.com/en/dev/topics/auth/" \l "module-django.contrib.auth" \o "django.contrib.auth: Django's authentication framework.), the authentication framework shipped by Django.

## Make the poll app modifiable in the admin

We need to tell the admin that Poll objects have an admin interface. To do this, open thepolls/admin.py file, and edit it to look like this:

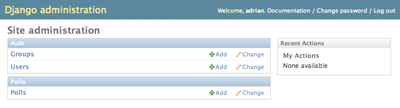
from django.contrib import admin

from polls.models import Poll

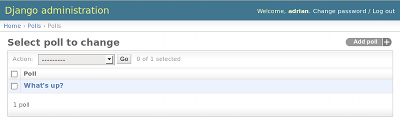
admin**.**site**.**register(Poll)

## Explore the free admin functionality

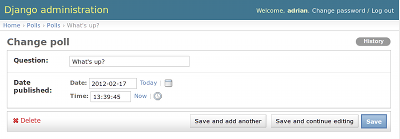
Now that we’ve registered Poll, Django knows that it should be displayed on the admin index page:



Click “Polls.” Now you’re at the “change list” page for polls. This page displays all the polls in the database and lets you choose one to change it. There’s the “What’s up?” poll we created in the first tutorial:



Click the “What’s up?” poll to edit it:



Things to note here:

* The form is automatically generated from the Poll model.
* The different model field types ([DateTimeField](https://docs.djangoproject.com/en/dev/ref/models/fields/" \l "django.db.models.DateTimeField" \o "django.db.models.DateTimeField), [CharField](https://docs.djangoproject.com/en/dev/ref/models/fields/" \l "django.db.models.CharField" \o "django.db.models.CharField)) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.

The bottom part of the page gives you a couple of options:

## Customize the admin form

By registering the Poll model withadmin.site.register(Poll), Django was able to construct a default form representation. Often, you’ll want to customize how the admin form looks and works. You’ll do this by telling Django the options you want when you register the object. Replace the admin.site.register(Poll) line with:

from django.contrib import admin

from polls.models import Poll

**class** **PollAdmin**(admin**.**ModelAdmin):

fields **=** ['pub\_date', 'question']

admin**.**site**.**register(Poll, PollAdmin)

Create a model admin object, then pass it as the second argument to admin.site.register() – any time you need to change the admin options for an object.

And speaking of forms with dozens of fields, you might want to split the form up into fieldsets:

from django.contrib import admin

from polls.models import Poll

**class** **PollAdmin**(admin**.**ModelAdmin):

fieldsets **=** [

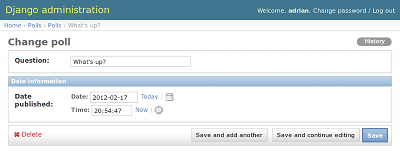
(None, {'fields': ['question']}),

('Date information', {'fields': ['pub\_date']}),

]

admin**.**site**.**register(Poll, PollAdmin)

The first element of each tuple in fieldsets is the title of the fieldset. Here’s what our form looks like now:



# Writing your first Django app, part 3

<https://docs.djangoproject.com/en/dev/intro/tutorial03/>

## Philosophy

In Django, web pages and other content are delivered by views. Each view is represented by a simple Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that’s requested (to be precise, the part of the URL after the domain name).

A URL pattern is simply the general form of a URL - for example: /newsarchive/<year>/<month>/. To get from a URL to a view, Django uses what are known as ‘URLconfs’. A URLconf maps URL patterns (described as regular expressions) to views.

## Write your first view

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

from django.http import HttpResponse

**def** **index**(request):

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

To call the view, we need to map it to a URL - and for this we need a URLconf. To create a URLconf in the polls directory, create a file called urls.py. Your app directory should now look like:

polls/

\_\_init\_\_.py

admin.py

models.py

tests.py

urls.py

views.py

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

from django.conf.urls import patterns, url

from polls import views

urlpatterns **=** patterns('',

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

)

The next step is to point the root URLconf at the polls.urls module. In mysite/urls.py insert an [include()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.include), leaving you with:

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

from django.contrib import admin

admin**.**autodiscover()

urlpatterns **=** patterns('',

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

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

)

You have now wired an index view into the URLconf. Go to <http://localhost:8000/polls/> in your browser, and you should see the text “Hello, world. You’re at the poll index.”, which you defined in the index view.

The [url()](https://docs.djangoproject.com/en/dev/ref/urls/" \l "django.conf.urls.url" \o "django.conf.urls.url) function is passed four arguments, two required: regex and view, and two optional: kwargs, and name. At this point, it’s worth reviewing what these arguments are for.

### [url()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.url) argument: regex

Django starts at the first regular expression and makes its way down the list, comparing the requested URL against each regular expression until it finds one that matches. These regular expressions do not search GET and POST parameters, or the domain name. For example, in a request to http://www.example.com/myapp/, the URLconf will look for myapp/. In a request to http://www.example.com/myapp/?page=3, the URLconf will also look for myapp/.

### [url()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.url) argument: view

When Django finds a regular expression match, Django calls the specified view function, with an [HttpRequest](https://docs.djangoproject.com/en/dev/ref/request-response/" \l "django.http.HttpRequest" \o "django.http.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.

### [url()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.url) argument: kwargs

Arbitrary keyword arguments can be passed in a dictionary to the target view. We aren’t going to use this feature of Django in the tutorial.

### [url()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.url) argument: name

Naming your URL lets you refer to it unambiguously from elsewhere in Django especially templates. This powerful feature allows you to make global changes to the url patterns of your project while only touching a single file.

## Writing more views

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

**def** **detail**(request, poll\_id):

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

**def** **results**(request, poll\_id):

**return** HttpResponse("You're looking at the results of poll %s." **%** poll\_id)

**def** **vote**(request, poll\_id):

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

Wire these new views into the polls.urls module by adding the following [url()](https://docs.djangoproject.com/en/dev/ref/urls/" \l "django.conf.urls.url" \o "django.conf.urls.url) calls:

from django.conf.urls import patterns, url

from polls import views

urlpatterns **=** patterns('',

*# ex: /polls/*

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

*# ex: /polls/5/*

url(r'^(?P<poll\_id>\d+)/$', views**.**detail, name**=**'detail'),

*# ex: /polls/5/results/*

url(r'^(?P<poll\_id>\d+)/results/$', views**.**results, name**=**'results'),

*# ex: /polls/5/vote/*

url(r'^(?P<poll\_id>\d+)/vote/$', views**.**vote, name**=**'vote'),

)

Take a look in your browser, at “/polls/34/”. It’ll run the detail() method and display whatever ID you provide in the URL. Try “/polls/34/results/” and “/polls/34/vote/” too – these will display the placeholder results and voting pages.

When somebody requests a page from your Web site – say, “/polls/34/”, Django will load the mysite.urls Python module because it’s pointed to by the [ROOT\_URLCONF](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-ROOT_URLCONF) setting. It finds the variable named urlpatterns and traverses the regular expressions in order. The [include()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.include) functions we are using simply reference other URLconfs. Note that the regular expressions for the [include()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.include) functions don’t have a $ (end-of-string match character) but rather a trailing slash. Whenever Django encounters [include()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.include), it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

The idea behind [include()](https://docs.djangoproject.com/en/dev/ref/urls/#django.conf.urls.include) is to make it easy to plug-and-play URLs. Since polls are in their own URLconf (polls/urls.py), they can be placed under “/polls/”, or under “/fun\_polls/”, or under “/content/polls/”, or any other path root, and the app will still work.

Here’s what happens if a user goes to “/polls/34/” in this system:

* Django will find the match at '^polls/'
* Then, Django will strip off the matching text ("polls/") and send the remaining text – "34/" – to the ‘polls.urls’ URLconf for further processing which matches r'^(?P<poll\_id>\d+)/$' resulting in a call to the detail() view like so:

detail(request=<HttpRequest object>, poll\_id='34')

The poll\_id='34' part comes from (?P<poll\_id>\d+). Using parentheses around a pattern “captures” the text matched by that pattern and sends it as an argument to the view function; ?P<poll\_id> defines the name that will be used to identify the matched pattern; and \d+ is a regular expression to match a sequence of digits (i.e., a number).

Because the URL patterns are regular expressions, there really is no limit on what you can do with them. And there’s no need to add URL cruft such as .html – unless you want to, in which case you can do something like this:

(r'^polls/latest\.html$', 'polls.views.index'),

But, don’t do that. It’s silly.

## Write views that actually do something

Each view is responsible for doing one of two things: returning an [HttpResponse](https://docs.djangoproject.com/en/dev/ref/request-response/" \l "django.http.HttpResponse" \o "django.http.HttpResponse) object containing the content for the requested page, or raising an exception such as [Http404](https://docs.djangoproject.com/en/dev/topics/http/views/#django.http.Http404). The rest is up to you.

Your view can read records from a database, or not. It can use a template system such as Django’s – or a third-party Python template system – or not. It can generate a PDF file, output XML, create a ZIP file on the fly, anything you want, using whatever Python libraries you want.

All Django wants is that [HttpResponse](https://docs.djangoproject.com/en/dev/ref/request-response/" \l "django.http.HttpResponse" \o "django.http.HttpResponse). Or an exception.

Because it’s convenient, let’s use Django’s own database API, which we covered in [Tutorial 1](https://docs.djangoproject.com/en/dev/intro/tutorial01/). Here’s one stab at the index()view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

from django.http import HttpResponse

from polls.models import Poll

**def** **index**(request):

latest\_poll\_list **=** Poll**.**objects**.**order\_by('-pub\_date')[:5]

output **=** ', '**.**join([p**.**question **for** p **in** latest\_poll\_list])

**return** HttpResponse(output)

There’s a problem here, though: the page’s design is hard-coded in the view. If you want to change the way the page looks, you’ll have to edit this Python code. So let’s use Django’s template system to separate the design from Python by creating a template that the view can use.

First, create a directory called templates in your polls directory. Django will look for templates in there.

Django’s [TEMPLATE\_LOADERS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-TEMPLATE_LOADERS) setting contains a list of callables that know how to import templates from various sources. One of the defaults is [django.template.loaders.app\_directories.Loader](https://docs.djangoproject.com/en/dev/ref/templates/api/" \l "django.template.loaders.app_directories.Loader" \o "django.template.loaders.app_directories.Loader) which looks for a “templates” subdirectory in each of the[INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS) - this is how Django knows to find the polls templates even though we didn’t modify [TEMPLATE\_DIRS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-TEMPLATE_DIRS), as we did in [Tutorial 2](https://docs.djangoproject.com/en/dev/intro/tutorial02/#ref-customizing-your-projects-templates).

**Organizing templates**

We could have all our templates together, in one big templates directory, and it would work perfectly well. However, this template belongs to the polls application, so unlike the admin template we created in the previous tutorial, we’ll put this one in the application’s template directory (polls/templates) rather than the project’s (templates). We’ll discuss in more detail in the [reusable apps tutorial](https://docs.djangoproject.com/en/dev/intro/reusable-apps/) why we do this.

Within the templates directory you have just created, create another directory called polls, and within that create a file called index.html. In other words, your template should be at polls/templates/polls/index.html. Because of how theapp\_directories template loader works as described above, you can refer to this template within Django simply aspolls/index.html.

**Template namespacing**

Now we might be able to get away with putting our templates directly in polls/templates (rather than creating another polls subdirectory), 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.

Put the following code in that template:

**{%** **if** latest\_poll\_list **%}**

<ul>

**{%** **for** poll **in** latest\_poll\_list **%}**

<li><a href="/polls/**{{** poll.id **}}**/">**{{** poll.question **}}**</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:

from django.http import HttpResponse

from django.template import RequestContext, loader

from polls.models import Poll

**def** **index**(request):

latest\_poll\_list **=** Poll**.**objects**.**order\_by('-pub\_date')[:5]

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

context **=** RequestContext(request, {

'latest\_poll\_list': latest\_poll\_list,

})

**return** HttpResponse(template**.**render(context))

That code loads the template called polls/index.html and passes it a context. The context is a dictionary mapping template variable names to Python objects.

Load the page by pointing your browser at “/polls/”, and you should see a bulleted-list containing the “What’s up” poll from Tutorial 1. The link points to the poll’s detail page.

### A shortcut: [render()](https://docs.djangoproject.com/en/dev/topics/http/shortcuts/#django.shortcuts.render)

It’s a very common idiom to load a template, fill a context and return an [HttpResponse](https://docs.djangoproject.com/en/dev/ref/request-response/" \l "django.http.HttpResponse" \o "django.http.HttpResponse) object with the result of the rendered template. Django provides a shortcut. Here’s the full index() view, rewritten:

from django.shortcuts import render

from polls.models import Poll

**def** **index**(request):

latest\_poll\_list **=** Poll**.**objects**.**all()**.**order\_by('-pub\_date')[:5]

context **=** {'latest\_poll\_list': latest\_poll\_list}

**return** render(request, 'polls/index.html', context)

Note that once we’ve done this in all these views, we no longer need to import [loader](https://docs.djangoproject.com/en/dev/ref/templates/api/#module-django.template.loader), [RequestContext](https://docs.djangoproject.com/en/dev/ref/templates/api/" \l "django.template.RequestContext" \o "django.template.RequestContext) and [HttpResponse](https://docs.djangoproject.com/en/dev/ref/request-response/" \l "django.http.HttpResponse" \o "django.http.HttpResponse)(you’ll want to keep HttpResponse if you still have the stub methods for detail, results, and vote).

The [render()](https://docs.djangoproject.com/en/dev/topics/http/shortcuts/#django.shortcuts.render) function takes the request object as its first argument, a template name as its second argument and a dictionary as its optional third argument. It returns an [HttpResponse](https://docs.djangoproject.com/en/dev/ref/request-response/" \l "django.http.HttpResponse" \o "django.http.HttpResponse) object of the given template rendered with the given context.

## Raising a 404 error

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

from django.http import Http404

from django.shortcuts import render

from polls.models import Poll

*# ...*

**def** **detail**(request, poll\_id):

**try**:

poll **=** Poll**.**objects**.**get(pk**=**poll\_id)

**except** Poll**.**DoesNotExist:

**raise** Http404

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

The new concept here: The view raises the [Http404](https://docs.djangoproject.com/en/dev/topics/http/views/#django.http.Http404) exception if a poll 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:

{{ poll }}

will get you started for now.

### A shortcut: [get\_object\_or\_404()](https://docs.djangoproject.com/en/dev/topics/http/shortcuts/#django.shortcuts.get_object_or_404)

It’s a very common idiom to use [get()](https://docs.djangoproject.com/en/dev/ref/models/querysets/#django.db.models.query.QuerySet.get) and raise [Http404](https://docs.djangoproject.com/en/dev/topics/http/views/#django.http.Http404) if the object doesn’t exist. Django provides a shortcut. Here’s thedetail() view, rewritten:

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

from polls.models import Poll

*# ...*

**def** **detail**(request, poll\_id):

poll **=** get\_object\_or\_404(Poll, pk**=**poll\_id)

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

The [get\_object\_or\_404()](https://docs.djangoproject.com/en/dev/topics/http/shortcuts/#django.shortcuts.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()](https://docs.djangoproject.com/en/dev/ref/models/querysets/#django.db.models.query.QuerySet.get) function of the model’s manager. It raises [Http404](https://docs.djangoproject.com/en/dev/topics/http/views/#django.http.Http404) if the object doesn’t exist.

**Philosophy**

Why do we use a helper function [get\_object\_or\_404()](https://docs.djangoproject.com/en/dev/topics/http/shortcuts/#django.shortcuts.get_object_or_404) instead of automatically catching the [ObjectDoesNotExist](https://docs.djangoproject.com/en/dev/ref/exceptions/" \l "django.core.exceptions.ObjectDoesNotExist" \o "django.core.exceptions.ObjectDoesNotExist)exceptions at a higher level, or having the model API raise [Http404](https://docs.djangoproject.com/en/dev/topics/http/views/#django.http.Http404) instead of [ObjectDoesNotExist](https://docs.djangoproject.com/en/dev/ref/exceptions/" \l "django.core.exceptions.ObjectDoesNotExist" \o "django.core.exceptions.ObjectDoesNotExist)?

Because that would couple the model layer to the view layer. One of the foremost design goals of Django is to maintain loose coupling. Some controlled coupling is introduced in the [django.shortcuts](https://docs.djangoproject.com/en/dev/topics/http/shortcuts/" \l "module-django.shortcuts" \o "django.shortcuts: Convenience shortcuts that span multiple levels of Django's MVC stack.) module.

There’s also a [get\_list\_or\_404()](https://docs.djangoproject.com/en/dev/topics/http/shortcuts/#django.shortcuts.get_list_or_404) function, which works just as [get\_object\_or\_404()](https://docs.djangoproject.com/en/dev/topics/http/shortcuts/#django.shortcuts.get_object_or_404) – except using [filter()](https://docs.djangoproject.com/en/dev/ref/models/querysets/#django.db.models.query.QuerySet.filter) instead of[get()](https://docs.djangoproject.com/en/dev/ref/models/querysets/#django.db.models.query.QuerySet.get). It raises [Http404](https://docs.djangoproject.com/en/dev/topics/http/views/#django.http.Http404) if the list is empty.

## Write a 404 (page not found) view

When you raise [Http404](https://docs.djangoproject.com/en/dev/topics/http/views/#django.http.Http404) from within a view, Django will load a special view devoted to handling 404 errors. It finds it by looking for the variable handler404 in your root URLconf (and only in your root URLconf; setting handler404 anywhere else will have no effect), which is a string in Python dotted syntax – the same format the normal URLconf callbacks use. A 404 view itself has nothing special: It’s just a normal view.

You normally won’t have to bother with writing 404 views. If you don’t set handler404, the built-in view[django.views.defaults.page\_not\_found()](https://docs.djangoproject.com/en/dev/topics/http/views/#django.views.defaults.page_not_found) is used by default. Optionally, you can create a 404.html template in the root of your template directory. The default 404 view will then use that template for all 404 errors when [DEBUG](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-DEBUG) is set to False (in your settings module). If you do create the template, add at least some dummy content like “Page not found”.

**Warning**

If [DEBUG](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-DEBUG) is set to False, all responses will be “Bad Request (400)” unless you specify the proper [ALLOWED\_HOSTS](https://docs.djangoproject.com/en/dev/ref/settings/" \l "std:setting-ALLOWED_HOSTS)as well (something like ['localhost', '127.0.0.1'] for local development).

A couple more things to note about 404 views:

* If [DEBUG](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-DEBUG) is set to True (in your settings module) then your 404 view will never be used (and thus the 404.html template will never be rendered) because the traceback will be displayed instead.
* The 404 view is also called if Django doesn’t find a match after checking every regular expression in the URLconf.

## Write a 500 (server error) view

Similarly, your root URLconf may define a handler500, which points to a view to call in case of server errors. Server errors happen when you have runtime errors in view code.

Likewise, you should create a 500.html template at the root of your template directory and add some content like “Something went wrong”.

## Use the template system

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

<h1>**{{** poll.question **}}**</h1>

<ul>

**{%** **for** choice **in** poll.choice\_set.all **%}**

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

**{%** **endfor** **%}**

</ul>

The template system uses dot-lookup syntax to access variable attributes. In the example of {{ poll.question }}, first Django does a dictionary lookup on the object poll. Failing that, it tries an attribute lookup – which works, in this case. If attribute lookup had failed, it would’ve tried a list-index lookup.

Method-calling happens in the [{% for %}](https://docs.djangoproject.com/en/dev/ref/templates/builtins/#std:templatetag-for) loop: poll.choice\_set.all is interpreted as the Python codepoll.choice\_set.all(), which returns an iterable of Choice objects and is suitable for use in the [{% for %}](https://docs.djangoproject.com/en/dev/ref/templates/builtins/#std:templatetag-for) tag.

See the [template guide](https://docs.djangoproject.com/en/dev/topics/templates/) for more about templates.

## Removing hardcoded URLs in templates

Remember, when we wrote the link to a poll in the polls/index.html template, the link was partially hardcoded like this:

<li><a href="/polls/**{{** poll.id **}}**/">**{{** poll.question **}}**</a></li>

The problem with this 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()](https://docs.djangoproject.com/en/dev/ref/urls/" \l "django.conf.urls.url" \o "django.conf.urls.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:

<li><a href="**{%** **url** 'detail' poll.id **%}**">**{{** poll.question **}}**</a></li>

**Note**

If {% url 'detail' poll.id %} (with quotes) doesn’t work, but {% url detail poll.id %} (without quotes) does, that means you’re using a version of Django < 1.5. In this case, add the following declaration at the top of your template:

**{%** **load** url from future **%}**

The way this works is by looking up the URL definition as specified in the polls.urls module. You can see exactly where the URL name of ‘detail’ is defined below:

**...**

*# the 'name' value as called by the {% url %} template tag*

url(r'^(?P<poll\_id>\d+)/$', views**.**detail, name**=**'detail'),

**...**

If you want to change the URL of the polls detail view to something else, perhaps to something like polls/specifics/12/instead of doing it in the template (or templates) you would change it in polls/urls.py:

**...**

*# added the word 'specifics'*

url(r'^specifics/(?P<poll\_id>\d+)/$', views**.**detail, name**=**'detail'),

**...**

## Namespacing URL names

The tutorial project has just one app, polls. In real Django projects, there might be five, ten, twenty apps or more. How does Django differentiate the URL names between them? For example, the polls app has a detail view, and so might an app on the same project that is for a blog. 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 root URLconf. In the mysite/urls.py file (the project’s urls.py, not the application’s), go ahead and change it to include namespacing:

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

from django.contrib import admin

admin**.**autodiscover()

urlpatterns **=** patterns('',

url(r'^polls/', include('polls.urls', namespace**=**"polls")),

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

)

Now change your polls/index.html template from:

<li><a href="**{%** **url** 'detail' poll.id **%}**">**{{** poll.question **}}**</a></li>

to point at the namespaced detail view:

<li><a href="**{%** **url** 'polls:detail' poll.id **%}**">**{{** poll.question **}}**</a></li>

When you’re comfortable with writing views, read [part 4 of this tutorial](https://docs.djangoproject.com/en/dev/intro/tutorial04/) to learn about simple form processing and generic views

MVC:

M -> Model, database.

V -> View

C -> Controller