https://docs.djangoproject.com/en/1.11/intro/tutorial01/

django-admin startproject mydjango

**Note**

You’ll need to avoid naming projects after built-in Python or Django components. In particular, this means you should avoid using names like **django** (which will conflict with Django itself) or **test** (which conflicts with a built-in Python package)

.

**Where should this code live?**

If your background is in plain old PHP (with no use of modern frameworks), you’re probably used to putting code under the Web server’s document root (in a place such as **/var/www**). With Django, you don’t do that. It’s not a good idea to put any of this Python code within your Web server’s document root, because it risks the possibility that people may be able to view your code over the Web. That’s not good for security.

Put your code in some directory **outside** of the document root, such as **/home/mycode**.

Let’s look at what [**startproject**](https://docs.djangoproject.com/en/1.11/ref/django-admin/#django-admin-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](https://docs.djangoproject.com/en/1.11/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. If you’re a Python beginner, read [more about packages](https://docs.python.org/3/tutorial/modules.html#tut-packages) in the official Python docs.
* **mysite/settings.py**: Settings/configuration for this Django project. [Django settings](https://docs.djangoproject.com/en/1.11/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/1.11/topics/http/urls/).
* **mysite/wsgi.py**: An entry-point for WSGI-compatible web servers to serve your project. See [How to deploy with WSGI](https://docs.djangoproject.com/en/1.11/howto/deployment/wsgi/)for more details.

The development server

python manage.py runserver

(thư mục mydjango ngoài cùng)

**Note**

Ignore the warning about unapplied database migrations for now; we’ll deal with the database shortly.

python manage.py runserver 8080

**$** 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**](https://docs.djangoproject.com/en/1.11/ref/django-admin/#django-admin-runserver) reference.

**Projects vs. apps**

What’s the difference between a project and an app? An app is a Web application that does something – 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 website. A project can contain multiple apps. An app can be in multiple projects.

Your apps can live anywhere on your [Python path](https://docs.python.org/3/tutorial/modules.html#tut-searchpath). 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

apps.py

migrations/

\_\_init\_\_.py

models.py

tests.py

views.py

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.")

This is the simplest view possible in Django. 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**.

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()**](https://docs.djangoproject.com/en/1.11/ref/urls/#django.conf.urls.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),

]

The [**include()**](https://docs.djangoproject.com/en/1.11/ref/urls/#django.conf.urls.include) function allows referencing other URLconfs. Note that the regular expressions for the [**include()**](https://docs.djangoproject.com/en/1.11/ref/urls/#django.conf.urls.include) function doesn’t have a **$** (end-of-string match character) but rather a trailing slash. Whenever Django encounters [**include()**](https://docs.djangoproject.com/en/1.11/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/1.11/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.

**When to use**[**include()**](https://docs.djangoproject.com/en/1.11/ref/urls/#django.conf.urls.include)

You should always use **include()** when you include other URL patterns. **admin.site.urls** is the only exception to this.

**Doesn’t match what you see?**

If you’re seeing **include(admin.site.urls)** instead of just **admin.site.urls**, 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.

python manage.py runserver

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

The [**url()**](https://docs.djangoproject.com/en/1.11/ref/urls/#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/1.11/ref/urls/#django.conf.urls.url) argument: regex[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial01/#url-argument-regex)

The term “regex” is a commonly used short form meaning “regular expression”, which is a syntax for matching patterns in strings, or in this case, url patterns. 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.

Note that these regular expressions do not search GET and POST parameters, or the domain name. For example, in a request to **https://www.example.com/myapp/**, the URLconf will look for **myapp/**. In a request to **https://www.example.com/myapp/?page=3**, the URLconf will also look for **myapp/**.

In fact, complex regexes can have poor lookup performance, so you probably shouldn’t rely on the full power of regexes.

Finally, a performance note: these regular expressions are compiled the first time the URLconf module is loaded. They’re super fast (as long as the lookups aren’t too complex as noted above).

### [url()](https://docs.djangoproject.com/en/1.11/ref/urls/#django.conf.urls.url) argument: view[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial01/#url-argument-view)

When Django finds a regular expression match, Django calls the specified view function, with an [**HttpRequest**](https://docs.djangoproject.com/en/1.11/ref/request-response/#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. We’ll give an example of this in a bit.

### [url()](https://docs.djangoproject.com/en/1.11/ref/urls/#django.conf.urls.url) argument: kwargs[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial01/#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/1.11/ref/urls/#django.conf.urls.url) argument: name[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial01/#url-argument-name)

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

<https://docs.djangoproject.com/en/1.11/intro/tutorial02/>

## Database setup[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial02/#database-setup)

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

If you wish to use another database, install the appropriate [database bindings](https://docs.djangoproject.com/en/1.11/topics/install/#database-installation) and change the following keys in the[**DATABASES**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-DATABASES) **'default'** item to match your database connection settings:

* [**ENGINE**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-DATABASE-ENGINE) – Either **'django.db.backends.sqlite3'**, **'django.db.backends.postgresql'**,**'django.db.backends.mysql'**, or **'django.db.backends.oracle'**. Other backends are [also available](https://docs.djangoproject.com/en/1.11/ref/databases/#third-party-notes).
* [**NAME**](https://docs.djangoproject.com/en/1.11/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/1.11/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 such as [**USER**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-USER), [**PASSWORD**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-PASSWORD), and [**HOST**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-HOST) must be added. For more details, see the reference documentation for [**DATABASES**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-DATABASES).

**For databases other than SQLite**

If you’re using a database besides SQLite, make sure you’ve created a database by this point. Do that with “**CREATE DATABASE database\_name;**” within your database’s interactive prompt.

Also make sure that the database user provided in **mysite/settings.py** has “create database” privileges. This allows automatic creation of a [test database](https://docs.djangoproject.com/en/1.11/topics/testing/overview/#the-test-database) which will be needed in a later tutorial.

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/1.11/ref/settings/#std:setting-TIME_ZONE) to your time zone.

Also, note the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.11/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/1.11/ref/settings/#std:setting-INSTALLED_APPS) contains the following apps, all of which come with Django:

* [**django.contrib.admin**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#module-django.contrib.admin) – The admin site. You’ll use it shortly.
* [**django.contrib.auth**](https://docs.djangoproject.com/en/1.11/topics/auth/#module-django.contrib.auth) – An authentication system.
* [**django.contrib.contenttypes**](https://docs.djangoproject.com/en/1.11/ref/contrib/contenttypes/#module-django.contrib.contenttypes) – A framework for content types.
* [**django.contrib.sessions**](https://docs.djangoproject.com/en/1.11/topics/http/sessions/#module-django.contrib.sessions) – A session framework.
* [**django.contrib.messages**](https://docs.djangoproject.com/en/1.11/ref/contrib/messages/#module-django.contrib.messages) – A messaging framework.
* [**django.contrib.staticfiles**](https://docs.djangoproject.com/en/1.11/ref/contrib/staticfiles/#module-django.contrib.staticfiles) – A framework for managing static files.

Some of these applications make 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 migrate

The [**migrate**](https://docs.djangoproject.com/en/1.11/ref/django-admin/#django-admin-migrate) command looks at the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-INSTALLED_APPS) setting and creates any necessary database tables according to the database settings in your **mysite/settings.py** file and the database migrations shipped with the app (we’ll cover those later). You’ll see a message for each migration it applies. If you’re interested, run the command-line client for your database and type **\dt** (PostgreSQL), **SHOW TABLES;** (MySQL), **.schema** (SQLite), or **SELECT TABLE\_NAME FROMUSER\_TABLES;** (Oracle) to display the tables Django created.

**For the minimalists**

Like we said above, 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/1.11/ref/settings/#std:setting-INSTALLED_APPS) before running [**migrate**](https://docs.djangoproject.com/en/1.11/ref/django-admin/#django-admin-migrate). The [**migrate**](https://docs.djangoproject.com/en/1.11/ref/django-admin/#django-admin-migrate) command will only run migrations for apps in[**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-INSTALLED_APPS).

## Creating models

A model is the single, definitive source of truth about your data. It contains the essential fields and behaviors of the data you’re storing. Django follows the [DRY Principle](https://docs.djangoproject.com/en/1.11/misc/design-philosophies/#dry). The goal is to define your data model in one place and automatically derive things from it.

This includes the migrations - unlike in Ruby On Rails, for example, 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.

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)

Finally, note a relationship is defined, using [**ForeignKey**](https://docs.djangoproject.com/en/1.11/ref/models/fields/#django.db.models.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

**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.

To include the app in our project, we need to add a reference to its configuration class in the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-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**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-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

By running **makemigrations**, you’re telling Django that you’ve made some changes to your models (in this case, you’ve made new ones) and that you’d like the changes to be stored as a migration.

By running **makemigrations**, you’re telling Django that you’ve made some changes to your models (in this case, you’ve made new ones) and that you’d like the changes to be stored as a migration.

**$** python manage.py sqlmigrate polls 0001

If you’re interested, you can also run [**python manage.py check**](https://docs.djangoproject.com/en/1.11/ref/django-admin/#django-admin-check); this checks for any problems in your project without making migrations or touching the database.

Now, run [**migrate**](https://docs.djangoproject.com/en/1.11/ref/django-admin/#django-admin-migrate) again to create those model tables in your database:

**$** python manage.py migrate

Now, run [**migrate**](https://docs.djangoproject.com/en/1.11/ref/django-admin/#django-admin-migrate) again to create those model tables in your database:

**$** python manage.py migrate

The reason that there are separate commands to make and apply migrations is because you’ll commit migrations to your version control system and ship them with your app; they not only make your development easier, they’re also useable by other developers and in production.

Read the [django-admin documentation](https://docs.djangoproject.com/en/1.11/ref/django-admin/) for full information on what the **manage.py** utility can do.

## Playing with the API[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial02/#playing-with-the-api)

**$** 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.

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

# No questions are in the system yet.

**>>>** Question.objects.all()

<QuerySet []>

# Create a new Question.

# 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() and it will do the right thing.

**>>> from** **django.utils** **import** timezone

**>>>** q = Question(question\_text="What's new?", pub\_date=timezone.now())

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

**>>>** q.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.

**>>>** q.id

1

# Access model field values via Python attributes.

**>>>** q.question\_text

"What's new?"

**>>>** q.pub\_date

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

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

**>>>** q.question\_text = "What's up?"

**>>>** q.save()

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

**>>>** Question.objects.all()

<QuerySet [<Question: Question object>]>

Let’s fix that by editing the **Question** model (in the **polls/models.py** file) and adding a [**\_\_str\_\_()**](https://docs.djangoproject.com/en/1.11/ref/models/instances/#django.db.models.Model.__str__) method to both **Question** and**Choice**:

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

## Introducing the Django Admin[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial02/#introducing-the-django-admin)

### Creating an admin user[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial02/#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

The final step is to enter your password. You will be asked to enter your password twice, the second time as a confirmation of the first.

Password: \*\*\*\*\*\*\*\*\*\*

Password (again): \*\*\*\*\*\*\*\*\*

Superuser created successfully.

Now, open a Web browser and go to “/admin/” on your local domain – e.g., <http://127.0.0.1:8000/admin/>. You should see the admin’s login screen:

### Make the poll app modifiable in the admin[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial02/#make-the-poll-app-modifiable-in-the-admin)

But where’s our poll app? It’s not displayed on the admin index page.

Just one thing to do: we need to tell the admin that **Question** objects have an admin interface. To do this, open the **polls/admin.py** file, and edit it to look like this:

polls/admin.py

**from** **django.contrib** **import** admin

**from** **.models** **import** Question

admin.site.register(Question)

### Explore the free admin functionality

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

Click “Questions”. Now you’re at the “change list” page for questions. This page displays all the questions in the database and lets you choose one to change it. There’s the “What’s up?” question we created earlier:

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

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

* Save – Saves changes and returns to the change-list page for this type of object.
* Save and continue editing – Saves changes and reloads the admin page for this object.
* Save and add another – Saves changes and loads a new, blank form for this type of object.
* Delete – Displays a delete confirmation page.

# Writing your first Django app, part 3

A view is a “type” of Web page in your Django application that generally serves a specific function and has a specific template. For example, in a blog application, you might have the following views:

* Blog homepage – displays the latest few entries.
* Entry “detail” page – permalink page for a single entry.
* Year-based archive page – displays all months with entries in the given year.
* Month-based archive page – displays all days with entries in the given month.
* Day-based archive page – displays all entries in the given day.
* Comment action – handles posting comments to a given entry.

In our poll application, we’ll have the following four views:

* Question “index” page – displays the latest few questions.
* Question “detail” page – displays a question text, with no results but with a form to vote.
* Question “results” page – displays results for a particular question.
* Vote action – handles voting for a particular choice in a particular question.

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).

## 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()**](https://docs.djangoproject.com/en/1.11/ref/urls/#django.conf.urls.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'),

]

ake 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 website – 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/1.11/ref/settings/#std:setting-ROOT_URLCONF) setting. It finds the variable named **urlpatterns** and traverses the regular expressions in order. After finding the match at **'^polls/'**, it strips off the matching text (**"polls/"**) and sends the remaining text – **"34/"** – to the ‘polls.urls’ URLconf for further processing. There it matches **r'^(?P<question\_id>[0-9]+)/$'**, resulting in a call to the **detail()** view like so:

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

The **question\_id='34'** part comes from **(?P<question\_id>[0-9]+)**. Using parentheses around a pattern “captures” the text matched by that pattern and sends it as an argument to the view function; **?P<question\_id>** defines the name that will be used to identify the matched pattern; and **[0-9]+** 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:

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

But, don’t do that. It’s silly.

## Write views that actually do something[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial03/#write-views-that-actually-do-something)

Each view is responsible for doing one of two things: returning an [**HttpResponse**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpResponse) object containing the content for the requested page, or raising an exception such as [**Http404**](https://docs.djangoproject.com/en/1.11/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/1.11/ref/request-response/#django.http.HttpResponse). Or an exception.

Because it’s convenient, let’s use Django’s own database API, which we covered in [Tutorial 2](https://docs.djangoproject.com/en/1.11/intro/tutorial02/). Here’s one stab at a new **index()** view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

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*

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.

Your project’s [**TEMPLATES**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-TEMPLATES) setting describes how Django will load and render templates. The default settings file configures a **DjangoTemplates** backend whose [**APP\_DIRS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-TEMPLATES-APP_DIRS) option is set to **True**. By convention **DjangoTemplates** looks for a “templates” subdirectory in each of the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-INSTALLED_APPS).

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 the **app\_directories** template loader works as described above, 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** (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 differentapplication, 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:

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))

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” question from [Tutorial 2](https://docs.djangoproject.com/en/1.11/intro/tutorial02/). The link points to the question’s detail page.

### A shortcut: [render()](https://docs.djangoproject.com/en/1.11/topics/http/shortcuts/#django.shortcuts.render)[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial03/#a-shortcut-render)

It’s a very common idiom to load a template, fill a context and return an [**HttpResponse**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.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)

Note that once we’ve done this in all these views, we no longer need to import [**loader**](https://docs.djangoproject.com/en/1.11/topics/templates/#module-django.template.loader) and [**HttpResponse**](https://docs.djangoproject.com/en/1.11/ref/request-response/#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/1.11/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/1.11/ref/request-response/#django.http.HttpResponse) object of the given template rendered with the given context.

## Raising a 404 error[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial03/#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**](https://docs.djangoproject.com/en/1.11/topics/http/views/#django.http.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 }}

will get you started for now.

### A shortcut: [get\_object\_or\_404()](https://docs.djangoproject.com/en/1.11/topics/http/shortcuts/#django.shortcuts.get_object_or_404)[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial03/#a-shortcut-get-object-or-404)

It’s a very common idiom to use [**get()**](https://docs.djangoproject.com/en/1.11/ref/models/querysets/#django.db.models.query.QuerySet.get) and raise [**Http404**](https://docs.djangoproject.com/en/1.11/topics/http/views/#django.http.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})

The [**get\_object\_or\_404()**](https://docs.djangoproject.com/en/1.11/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/1.11/ref/models/querysets/#django.db.models.query.QuerySet.get) function of the model’s manager. It raises [**Http404**](https://docs.djangoproject.com/en/1.11/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/1.11/topics/http/shortcuts/#django.shortcuts.get_object_or_404) instead of automatically catching the[**ObjectDoesNotExist**](https://docs.djangoproject.com/en/1.11/ref/exceptions/#django.core.exceptions.ObjectDoesNotExist) exceptions at a higher level, or having the model API raise [**Http404**](https://docs.djangoproject.com/en/1.11/topics/http/views/#django.http.Http404) instead of[**ObjectDoesNotExist**](https://docs.djangoproject.com/en/1.11/ref/exceptions/#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/1.11/topics/http/shortcuts/#module-django.shortcuts) module.

There’s also a [**get\_list\_or\_404()**](https://docs.djangoproject.com/en/1.11/topics/http/shortcuts/#django.shortcuts.get_list_or_404) function, which works just as [**get\_object\_or\_404()**](https://docs.djangoproject.com/en/1.11/topics/http/shortcuts/#django.shortcuts.get_object_or_404) – except using [**filter()**](https://docs.djangoproject.com/en/1.11/ref/models/querysets/#django.db.models.query.QuerySet.filter)instead of [**get()**](https://docs.djangoproject.com/en/1.11/ref/models/querysets/#django.db.models.query.QuerySet.get). It raises [**Http404**](https://docs.djangoproject.com/en/1.11/topics/http/views/#django.http.Http404) if the list is empty.

## Use the template system[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial03/#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**>

The template system uses dot-lookup syntax to access variable attributes. In the example of **{{question.question\_text }}**, first Django does a dictionary lookup on the object **question**. 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/1.11/ref/templates/builtins/#std:templatetag-for) loop: **question.choice\_set.all** is interpreted as the Python code**question.choice\_set.all()**, which returns an iterable of **Choice** objects and is suitable for use in the [**{% for %}**](https://docs.djangoproject.com/en/1.11/ref/templates/builtins/#std:templatetag-for)tag.

See the [template guide](https://docs.djangoproject.com/en/1.11/topics/templates/) for more about templates.

## Removing hardcoded URLs in templates[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial03/#removing-hardcoded-urls-in-templates)

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

<**li**><**a** href="/polls/{{ question.id }}/">{{ question.question\_text }}</**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/1.11/ref/urls/#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' question.id %}">{{ question.question\_text }}</**a**></**li**>

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<question\_id>[0-9]+)/$', 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<question\_id>[0-9]+)/$', views.detail, name='detail'),

...

## Namespacing URL names[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial03/#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 URLconf. In the **polls/urls.py** file, go ahead and add an **app\_name** to set the application namespace:

polls/urls.py

**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'),

]

Now change your **polls/index.html** template from:

polls/templates/polls/index.html

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

to point at the namespaced detail view:

polls/templates/polls/index.html

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

# Writing your first Django app, part 4[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial04/#writing-your-first-django-app-part-4)

This tutorial begins where [Tutorial 3](https://docs.djangoproject.com/en/1.11/intro/tutorial03/) left off. We’re continuing the Web-poll application and will focus on simple form processing and cutting down our code.

## Write a simple form

Let’s update our poll detail template (“polls/detail.html”) from the last tutorial, so that the template contains an HTML **<form>** element:

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**>

A quick rundown:

* The above template displays a radio button for each question choice. The **value** of each radio button is the associated question choice’s ID. The **name** of each radio button is **"choice"**. That means, when somebody selects one of the radio buttons and submits the form, it’ll send the POST data **choice=#** where # is the ID of the selected choice. This is the basic concept of HTML forms.
* We set the form’s **action** to **{% url 'polls:vote' question.id %}**, and we set **method="post"**. Using **method="post"** (as opposed to **method="get"**) is very important, because the act of submitting this form will alter data server-side. Whenever you create a form that alters data server-side, use **method="post"**. This tip isn’t specific to Django; it’s just good Web development practice.
* **forloop.counter** indicates how many times the [**for**](https://docs.djangoproject.com/en/1.11/ref/templates/builtins/#std:templatetag-for) tag has gone through its loop
* 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 %}**](https://docs.djangoproject.com/en/1.11/ref/templates/builtins/#std:templatetag-csrf_token)template tag.

Now, let’s create a Django view that handles the submitted data and does something with it. Remember, in [Tutorial 3](https://docs.djangoproject.com/en/1.11/intro/tutorial03/), we created a URLconf for the polls application that includes this line:

polls/urls.py

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

We also created a dummy implementation of the **vote()** function. Let’s create a real version. Add the following to **polls/views.py**:

This code includes a few things we haven’t covered yet in this tutorial:

* [**request.POST**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpRequest.POST) is a dictionary-like object that lets you access submitted data by key name. In this case,**request.POST['choice']** returns the ID of the selected choice, as a string. [**request.POST**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpRequest.POST) values are always strings.

Note that Django also provides [**request.GET**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpRequest.GET) for accessing GET data in the same way – but we’re explicitly using [**request.POST**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpRequest.POST) in our code, to ensure that data is only altered via a POST call.

* **request.POST['choice']** will raise [**KeyError**](https://docs.python.org/3/library/exceptions.html#KeyError) if **choice** wasn’t provided in POST data. The above code checks for[**KeyError**](https://docs.python.org/3/library/exceptions.html#KeyError) and redisplays the question form with an error message if **choice** isn’t given.
* After incrementing the choice count, the code returns an [**HttpResponseRedirect**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpResponseRedirect) rather than a normal[**HttpResponse**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpResponse). [**HttpResponseRedirect**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpResponseRedirect) takes a single argument: the URL to which the user will be redirected (see the following point for how we construct the URL in this case).

As the Python comment above points out, you should always return an [**HttpResponseRedirect**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpResponseRedirect) after successfully dealing with POST data. This tip isn’t specific to Django; it’s just good Web development practice.

* We are using the [**reverse()**](https://docs.djangoproject.com/en/1.11/ref/urlresolvers/#django.urls.reverse) function in the [**HttpResponseRedirect**](https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpResponseRedirect) constructor in this example. This function helps avoid having to hardcode a URL in the view function. It is given the name of the view that we want to pass control to and the variable portion of the URL pattern that points to that view. In this case, using the URLconf we set up in [Tutorial 3](https://docs.djangoproject.com/en/1.11/intro/tutorial03/), this [**reverse()**](https://docs.djangoproject.com/en/1.11/ref/urlresolvers/#django.urls.reverse) call will return a string like
* '/polls/3/results/'

After somebody votes in a question, the **vote()** view redirects to the results page for the question. Let’s write that view:

polls/views.py

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

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

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

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

Now, create a **polls/results.html** template:

polls/templates/polls/results.html

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

<**ul**>

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

<**li**>{{ choice.choice\_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</**li**>

{% **endfor** %}

</**ul**>

<**a** href="{% **url** 'polls:detail' question.id %}">Vote again?</**a**>

**Note**

The code for our **vote()** view does have a small problem. It first gets the **selected\_choice** object from the database, then computes the new value of **votes**, and then saves it back to the database. If two users of your website try to vote at exactly the same time, this might go wrong: The same value, let’s say 42, will be retrieved for **votes**. Then, for both users the new value of 43 is computed and saved, but 44 would be the expected value.

This is called a race condition. If you are interested, you can read [Avoiding race conditions using F()](https://docs.djangoproject.com/en/1.11/ref/models/expressions/#avoiding-race-conditions-using-f) to learn how you can solve this issue.

## Use generic views: Less code is better[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial04/#use-generic-views-less-code-is-better)

The **detail()** (from [Tutorial 3](https://docs.djangoproject.com/en/1.11/intro/tutorial03/)) and **results()** views are very simple – and, as mentioned above, redundant. The **index()** view, which displays a list of polls, is similar.

These views represent 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.

Generic views abstract common patterns to the point where you don’t even need to write Python code to write an app.

Let’s convert our poll app to use the generic views system, so we can delete a bunch of our own code. We’ll just have to take a few steps to make the conversion. We will:

1. Convert the URLconf.
2. Delete some of the old, unneeded views.
3. Introduce new views based on Django’s generic views.

Read on for details.

**Why the code-shuffle?**

Generally, when writing a Django app, you’ll evaluate whether generic views are a good fit for your problem, and you’ll use them from the beginning, rather than refactoring your code halfway through. But this tutorial intentionally has focused on writing the views “the hard way” until now, to focus on core concepts.

You should know basic math before you start using a calculator.

### Amend URLconf[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial04/#amend-urlconf)

First, open the **polls/urls.py** URLconf and change it like so:

polls/urls.py

**from** **django.conf.urls** **import** url

**from** **.** **import** views

app\_name = 'polls'

urlpatterns = [

url(r'^$', views.IndexView.as\_view(), name='index'),

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

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

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

]

Note that the name of the matched pattern in the regexes of the second and third patterns has changed from **<question\_id>** to **<pk>**.

### Amend views[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial04/#amend-views)

Next, we’re going to remove our old **index**, **detail**, and **results** views and use Django’s generic views instead. To do so, open the **polls/views.py** file and change it like so:

polls/views.py

**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 above, no changes needed.*

We’re using two generic views here: [**ListView**](https://docs.djangoproject.com/en/1.11/ref/class-based-views/generic-display/#django.views.generic.list.ListView) and [**DetailView**](https://docs.djangoproject.com/en/1.11/ref/class-based-views/generic-display/#django.views.generic.detail.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**](https://docs.djangoproject.com/en/1.11/ref/class-based-views/generic-display/#django.views.generic.detail.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**](https://docs.djangoproject.com/en/1.11/ref/class-based-views/generic-display/#django.views.generic.detail.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. We also specify the **template\_name**for the **results** list view – this ensures that the results view and the detail view have a different appearance when rendered, even though they’re both a [**DetailView**](https://docs.djangoproject.com/en/1.11/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView) behind the scenes.

Similarly, the [**ListView**](https://docs.djangoproject.com/en/1.11/ref/class-based-views/generic-display/#django.views.generic.list.ListView) generic view uses a default template called **<app name>/<model name>\_list.html**; we use **template\_name** to tell [**ListView**](https://docs.djangoproject.com/en/1.11/ref/class-based-views/generic-display/#django.views.generic.list.ListView) to use our existing **"polls/index.html"** template.

In previous parts of the tutorial, the templates have been provided with a context that contains the **question** and **latest\_question\_list** context variables. For **DetailView** the **question** variable is provided automatically – since we’re using a Django model (**Question**), Django is able to determine an appropriate name for the context variable. However, for ListView, the automatically generated context variable is **question\_list**. To override this we provide the **context\_object\_name** attribute, specifying that we want to use **latest\_question\_list** instead. As an alternative approach, you could change your templates to match the new default context variables – but it’s a lot easier to just tell Django to use the variable you want.

Run the server, and use your new polling app based on generic views.

For full details on generic views, see the [generic views documentation](https://docs.djangoproject.com/en/1.11/topics/class-based-views/).

# Writing your first Django app, part 5

This tutorial begins where [Tutorial 4](https://docs.djangoproject.com/en/1.11/intro/tutorial04/) left off. We’ve built a Web-poll application, and we’ll now create some automated tests for it.

# Writing your first Django app, part 6

This tutorial begins where [Tutorial 5](https://docs.djangoproject.com/en/1.11/intro/tutorial05/) left off. We’ve built a tested Web-poll application, and we’ll now add a stylesheet and an image.

Aside from the HTML generated by the server, web applications generally need to serve additional files — such as images, JavaScript, or CSS — necessary to render the complete web page. In Django, we refer to these files as “static files”.

For small projects, this isn’t a big deal, because you can just keep the static files somewhere your web server can find it. However, in bigger projects – especially those comprised of multiple apps – dealing with the multiple sets of static files provided by each application starts to get tricky.

That’s what **django.contrib.staticfiles** is for: it collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in production.

## Customize your app’s look and feel

First, 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/**.

Django’s [**STATICFILES\_FINDERS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-STATICFILES_FINDERS) setting contains a list of finders that know how to discover static files from various sources. One of the defaults is **AppDirectoriesFinder** which looks for a “static” subdirectory in each of the[**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-INSTALLED_APPS), like the one in **polls** we just created. The admin site uses the same directory structure for its static files.

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**

Just like templates, we might be able to get away with putting our static files directly in **polls/static**(rather than creating another **polls** subdirectory), but it would actually be a bad idea. 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 differentapplication, 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.

Put the following code in that stylesheet (**polls/static/polls/style.css**):

polls/static/polls/style.css

**li** **a** {

**color**: **green**;

}

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

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.

That’s all you need to do for development. Reload **http://localhost:8000/polls/** and you should see that the question links are green (Django style!) which means that your stylesheet was properly loaded.

## Adding a background-image

Next, we’ll create a subdirectory for images. Create an **images** subdirectory in the **polls/static/polls/** directory. Inside this directory, put an image called **background.gif**. In other words, put your image in**polls/static/polls/images/background.gif**.

Then, add to your stylesheet (**polls/static/polls/style.css**):

polls/static/polls/style.css

**body** {

**background**: **white** url("images/background.gif") **no-repeat** **right** **bottom**;

}

Reload **http://localhost:8000/polls/** and you should see the background loaded in the bottom right of the screen.

**Warning**

Of course the **{% static %}** template tag is not available for use in static files like your stylesheet which aren’t generated by Django. You should always use **relative paths** to link your static files between each other, because then you can change [**STATIC\_URL**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-STATIC_URL) (used by the [**static**](https://docs.djangoproject.com/en/1.11/ref/templates/builtins/#std:templatetag-static) template tag to generate its URLs) without having to modify a bunch of paths in your static files as well.

These are the **basics**. For more details on settings and other bits included with the framework see [the static files howto](https://docs.djangoproject.com/en/1.11/howto/static-files/) and[the staticfiles reference](https://docs.djangoproject.com/en/1.11/ref/contrib/staticfiles/). [Deploying static files](https://docs.djangoproject.com/en/1.11/howto/static-files/deployment/) discusses how to use static files on a real server.

# Writing your first Django app, part 7

This tutorial begins where [Tutorial 6](https://docs.djangoproject.com/en/1.11/intro/tutorial06/) left off. We’re continuing the Web-poll application and will focus on customizing Django’s automatically-generated admin site that we first explored in [Tutorial 2](https://docs.djangoproject.com/en/1.11/intro/tutorial02/).

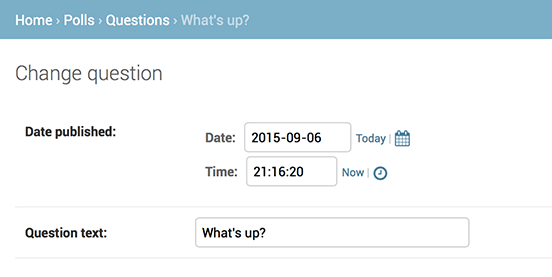
## Customize the admin form

By registering the **Question** model with **admin.site.register(Question)**, 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.

Let’s see how this works by reordering the fields on the edit form. Replace the **admin.site.register(Question)** line with:

You’ll follow this pattern – create a model admin class, then pass it as the second argument to **admin.site.register()**– any time you need to change the admin options for a model.

This particular change above makes the “Publication date” come before the “Question” field:



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

polls/admin.py

**from** **django.contrib** **import** admin

**from** **.models** **import** Question

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

fieldsets = [

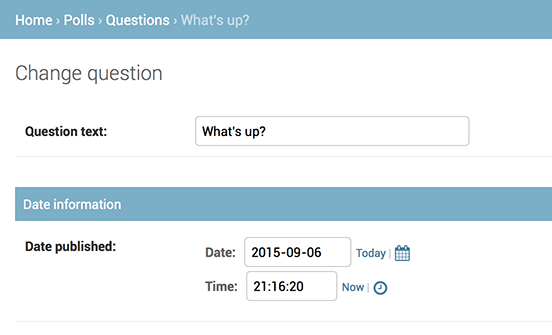
(**None**, {'fields': ['question\_text']}),

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

]

admin.site.register(Question, QuestionAdmin)

The first element of each tuple in [**fieldsets**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.fieldsets) is the title of the fieldset. Here’s what our form looks like now:



## Adding related objects

OK, we have our Question admin page, but a **Question** has multiple **Choice**s, and the admin page doesn’t display choices.

Yet.

There are two ways to solve this problem. The first is to register **Choice** with the admin just as we did with **Question**. That’s easy:

polls/admin.py

**from** **django.contrib** **import** admin

**from** **.models** **import** Choice, Question

*# ...*

admin.site.register(Choice)

In that form, the “Question” field is a select box containing every question in the database. Django knows that a [**ForeignKey**](https://docs.djangoproject.com/en/1.11/ref/models/fields/#django.db.models.ForeignKey) should be represented in the admin as a **<select>** box. In our case, only one question exists at this point.

Also note the “Add Another” link next to “Question.” Every object with a **ForeignKey** relationship to another gets this for free. When you click “Add Another”, you’ll get a popup window with the “Add question” form. If you add a question in that window and click “Save”, Django will save the question to the database and dynamically add it as the selected choice on the “Add choice” form you’re looking at.

But, really, this is an inefficient way of adding **Choice** objects to the system. It’d be better if you could add a bunch of Choices directly when you create the **Question** object. Let’s make that happen.

Remove the **register()** call for the **Choice** model. Then, edit the **Question** registration code to read:

polls/admin.py

**from** **django.contrib** **import** admin

**from** **.models** **import** Choice, Question

**class** **ChoiceInline**(admin.StackedInline):

model = Choice

extra = 3

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

fieldsets = [

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

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

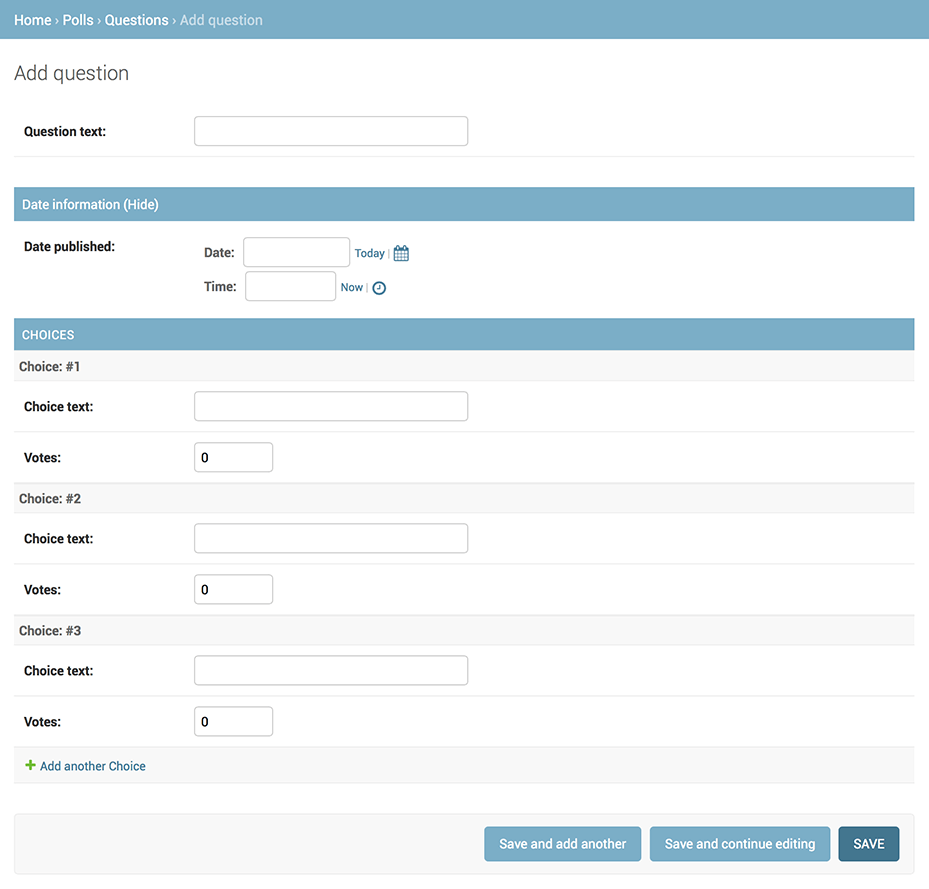
]

inlines = [ChoiceInline]

admin.site.register(Question, QuestionAdmin)

This tells Django: “**Choice** objects are edited on the **Question** admin page. By default, provide enough fields for 3 choices.”

Load the “Add question” page to see how that looks:



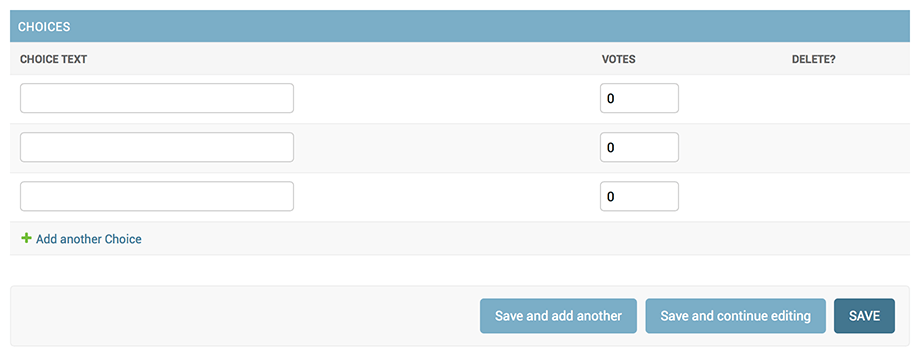
One small problem, though. It takes a lot of screen space to display all the fields for entering related **Choice** objects. For that reason, Django offers a tabular way of displaying inline related objects; you just need to change the **ChoiceInline**declaration to read:

polls/admin.py

**class** **ChoiceInline**(admin.TabularInline):

*#...*

With that **TabularInline** (instead of **StackedInline**), the related objects are displayed in a more compact, table-based format:



Note that there is an extra “Delete?” column that allows removing rows added using the “Add Another Choice” button and rows that have already been saved.

## Customize the admin change list

Now that the Question admin page is looking good, let’s make some tweaks to the “change list” page – the one that displays all the questions in the system.

## By default, Django displays the str() of each object. But sometimes it’d be more helpful if we could display individual fields. To do that, use the [list\_display](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display) admin option, which is a tuple of field names to display, as columns, on the change list page for the object:

polls/admin.py

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

*# ...*

list\_display = ('question\_text', 'pub\_date')

Just for good measure, let’s also include the **was\_published\_recently()** method from [Tutorial 2](https://docs.djangoproject.com/en/1.11/intro/tutorial02/):

polls/admin.py

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

*# ...*

list\_display = ('question\_text', 'pub\_date', 'was\_published\_recently')

You can click on the column headers to sort by those values – except in the case of the **was\_published\_recently**header, because sorting by the output of an arbitrary method is not supported. Also note that the column header for**was\_published\_recently** is, by default, the name of the method (with underscores replaced with spaces), and that each line contains the string representation of the output.

You can improve that by giving that method (in **polls/models.py**) a few attributes, as follows:

polls/models.py

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

*# ...*

**def** was\_published\_recently(self):

now = timezone.now()

**return** now - datetime.timedelta(days=1) <= self.pub\_date <= now

was\_published\_recently.admin\_order\_field = 'pub\_date'

was\_published\_recently.boolean = **True**

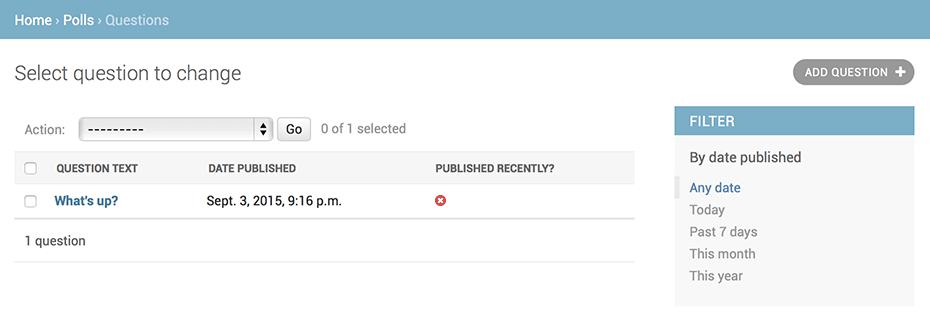
was\_published\_recently.short\_description = 'Published recently?'

For more information on these method properties, see [**list\_display**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display).

Edit your **polls/admin.py** file again and add an improvement to the **Question** change list page: filters using the[**list\_filter**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_filter). Add the following line to **QuestionAdmin**:

list\_filter = ['pub\_date']

That adds a “Filter” sidebar that lets people filter the change list by the **pub\_date** field:



The type of filter displayed depends on the type of field you’re filtering on. Because **pub\_date** is a [**DateTimeField**](https://docs.djangoproject.com/en/1.11/ref/models/fields/#django.db.models.DateTimeField), Django knows to give appropriate filter options: “Any date”, “Today”, “Past 7 days”, “This month”, “This year”.

This is shaping up well. Let’s add some search capability:

search\_fields = ['question\_text']

That adds a search box at the top of the change list. When somebody enters search terms, Django will search the **question\_text** field. You can use as many fields as you’d like – although because it uses a **LIKE** query behind the scenes, limiting the number of search fields to a reasonable number will make it easier for your database to do the search.

Now’s also a good time to note that change lists give you free pagination. The default is to display 100 items per page. [**Change list pagination**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_per_page), [**search boxes**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.search_fields), [**filters**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_filter), [**date-hierarchies**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.date_hierarchy), and [**column-header-ordering**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display) all work together like you think they should.

## Customize the admin look and feel[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial07/#customize-the-admin-look-and-feel)

Clearly, having “Django administration” at the top of each admin page is ridiculous. It’s just placeholder text.

That’s easy to change, though, using Django’s template system. The Django admin is powered by Django itself, and its interfaces use Django’s own template system.

### Customizing your project’s templates[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial07/#customizing-your-project-s-templates)

Create a **templates** directory in your project directory (the one that contains **manage.py**). Templates can live anywhere on your filesystem that Django can access. (Django runs as whatever user your server runs.) However, keeping your templates within the project is a good convention to follow.

Open your settings file (**mysite/settings.py**, remember) and add a [**DIRS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-TEMPLATES-DIRS) option in the [**TEMPLATES**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-TEMPLATES) setting:

mysite/settings.py

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [os.path.join(BASE\_DIR, 'templates')],

'APP\_DIRS': **True**,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

[**DIRS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-TEMPLATES-DIRS) is a list of filesystem directories to check when loading Django templates; it’s a search path.

**Organizing templates**

Just like the static files, we could have all our templates together, in one big templates directory, and it would work perfectly well. However, templates that belong to a particular application should be placed in that application’s template directory (e.g. **polls/templates**) rather than the project’s (**templates**). We’ll discuss in more detail in the [reusable apps tutorial](https://docs.djangoproject.com/en/1.11/intro/reusable-apps/) why we do this.

Now create a directory called **admin** inside **templates**, and copy the template **admin/base\_site.html** from within the default Django admin template directory in the source code of Django itself (**django/contrib/admin/templates**) into that directory.

**Where are the Django source files?**

If you have difficulty finding where the Django source files are located on your system, run the following command:

**$** python -c "import django; print(django.\_\_path\_\_)"

Then, just edit the file and replace **{{ site\_header|default:\_('Django administration') }}** (including the curly braces) with your own site’s name as you see fit. You should end up with a section of code like:

{% **block** branding %}

<**h1** id="site-name"><**a** href="{% **url** 'admin:index' %}">Polls Administration</**a**></**h1**>

{% **endblock** %}

We use this approach to teach you how to override templates. In an actual project, you would probably use the [**django.contrib.admin.AdminSite.site\_header**](https://docs.djangoproject.com/en/1.11/ref/contrib/admin/#django.contrib.admin.AdminSite.site_header) attribute to more easily make this particular customization.

his template file contains lots of text like **{% block branding %}** and **{{ title }}**. The **{%** and **{{** tags are part of Django’s template language. When Django renders **admin/base\_site.html**, this template language will be evaluated to produce the final HTML page, just like we saw in [Tutorial 3](https://docs.djangoproject.com/en/1.11/intro/tutorial03/).

### Customizing your application’s templates[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial07/#customizing-your-application-s-templates)

Astute readers will ask: But if [**DIRS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-TEMPLATES-DIRS) was empty by default, how was Django finding the default admin templates? The answer is that, since [**APP\_DIRS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-TEMPLATES-APP_DIRS) is set to **True**, Django automatically looks for a **templates/** subdirectory within each application package, for use as a fallback (don’t forget that **django.contrib.admin** is an application).

Our poll application is not very complex and doesn’t need custom admin templates. But if it grew more sophisticated and required modification of Django’s standard admin templates for some of its functionality, it would be more sensible to modify the application’s templates, rather than those in the project. That way, you could include the polls application in any new project and be assured that it would find the custom templates it needed.

See the [template loading documentation](https://docs.djangoproject.com/en/1.11/topics/templates/#template-loading) for more information about how Django finds its templates.

## Customize the admin index page

On a similar note, you might want to customize the look and feel of the Django admin index page.

By default, it displays all the apps in [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-INSTALLED_APPS) that have been registered with the admin application, in alphabetical order. You may want to make significant changes to the layout. After all, the index is probably the most important page of the admin, and it should be easy to use.

The template to customize is **admin/index.html**. (Do the same as with **admin/base\_site.html** in the previous section – copy it from the default directory to your custom template directory). Edit the file, and you’ll see it uses a template variable called **app\_list**. That variable contains every installed Django app. Instead of using that, you can hard-code links to object-specific admin pages in whatever way you think is best.

## What’s next?[¶](https://docs.djangoproject.com/en/1.11/intro/tutorial07/#what-s-next)

The beginner tutorial ends here. In the meantime, you might want to check out some pointers on [where to go from here](https://docs.djangoproject.com/en/1.11/intro/whatsnext/).

If you are familiar with Python packaging and interested in learning how to turn polls into a “reusable app”, check out [Advanced tutorial: How to write reusable apps](https://docs.djangoproject.com/en/1.11/intro/reusable-apps/).