<https://www.djangoproject.com/start/>

Tag line: The web framework for perfectionists with deadlines.

Django is a Python web framework, thus requiring Python to be installed on your machine. At the time of writing, Python 3.12 is the latest version.

py --version

[pip](https://pypi.org/project/pip/) is a package manager for Python and is included by default with the Python installer. It helps to install and uninstall Python packages (such as Django!). For the rest of the installation, we’ll use **pip** to install Python packages from the command line

py -m venv project-name

This will create a folder called ‘project-name’ if it does not already exist and set up the virtual environment. To activate the environment, run:

**...\>** project-name\Scripts\activate.bat

The virtual environment will be activated and you’ll see “(project-name)” next to the command prompt to designate that. Each time you start a new command prompt, you’ll need to activate the environment again.

py -m pip install Django

After the installation has completed, you can verify your Django installation by executing **django-admin --version** in the command prompt.

To verify that Django can be seen by Python, type **python** from your shell. Then at the Python prompt, try to import Django:

>>> import django

>>> print(django.get\_version())

<https://docs.djangoproject.com/en/5.0/intro/tutorial01/>

# Writing your first Django app, part 1[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#writing-your-first-django-app-part-1)

Let’s learn by example.

Throughout this tutorial, we’ll walk you through the creation of a basic poll application.

It’ll consist of two parts:

* A public site that lets people view polls and vote in them.
* An admin site that lets you add, change, and delete polls.

## **Creating a project**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#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/5.0/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 following command:

/ 

**...\>** django-admin startproject mysite

This will create a **mysite** directory in your current directory. If it didn’t work, see [Problems running django-admin](https://docs.djangoproject.com/en/5.0/faq/troubleshooting/#troubleshooting-django-admin).

Let’s look at what **[startproject](https://docs.djangoproject.com/en/5.0/ref/django-admin/" \l "django-admin-startproject)** created:

mysite/

manage.py

mysite/

\_\_init\_\_.py

settings.py

urls.py

asgi.py

wsgi.py

These files are:

* The outer **mysite/** root directory is 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/5.0/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/5.0/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/5.0/topics/http/urls/).
* **mysite/asgi.py**: An entry-point for ASGI-compatible web servers to serve your project. See [How to deploy with ASGI](https://docs.djangoproject.com/en/5.0/howto/deployment/asgi/) for more details.
* **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/5.0/howto/deployment/wsgi/) for more details.

## **The development server**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#the-development-server)

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

/ 

**...\>** py manage.py runserver

You’ve started the Django development server, a lightweight web server written purely in Python. We’ve included this with Django so you can develop things rapidly, without having to deal with configuring a production server – such as Apache – until you’re ready for production.

Now’s a good time to note: **don’t** use this server in anything resembling a production environment. It’s intended only for use while developing. (We’re in the business of making web frameworks, not web servers.)

## **Creating the Polls app**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#creating-the-polls-app)

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, so you can focus on writing code rather than creating directories.

**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 blog system, a database of public records or a small 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 in the same directory as 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:

/ 

**...\>** py manage.py startapp polls

## **Write your first view**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#write-your-first-view)

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

polls/views.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#id1)

**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**. Your app directory should now look like:

polls/

\_\_init\_\_.py

admin.py

apps.py

migrations/

\_\_init\_\_.py

models.py

tests.py

urls.py

views.py

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

polls/urls.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#id2)

**from** **django.urls** **import** path

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

urlpatterns = [

path("", 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.urls.include** and insert an [**include()**](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.include) in the **urlpatterns** list, so you have:

mysite/urls.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#id3)

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

**from** **django.urls** **import** include, path

urlpatterns = [

path("polls/", include("polls.urls")),

path("admin/", admin.site.urls),

]

The [**include()**](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.include) function allows referencing other URLconfs. Whenever Django encounters [**include()**](https://docs.djangoproject.com/en/5.0/ref/urls/#django.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/5.0/ref/urls/#django.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/5.0/ref/urls/#django.urls.include)

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

You have now wired an **index** view into the URLconf. Verify it’s working with the following command:

/ 

**...\>** py 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.

**Page not found?**

If you get an error page here, check that you’re going to <http://localhost:8000/polls/> and not <http://localhost:8000/>.

The [**path()**](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.path) function is passed four arguments, two required: **route** and **view**, and two optional: **kwargs**, and **name**. At this point, it’s worth reviewing what these arguments are for.

### [path()](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.path) argument: route[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#path-argument-route)

**route** is a string that contains a URL pattern. When processing a request, Django starts at the first pattern in **urlpatterns** and makes its way down the list, comparing the requested URL against each pattern until it finds one that matches.

Patterns don’t 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/**.

### [path()](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.path) argument: view[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#path-argument-view)

When Django finds a matching pattern, it calls the specified view function with an **[HttpRequest](https://docs.djangoproject.com/en/5.0/ref/request-response/" \l "django.http.HttpRequest" \o "django.http.HttpRequest)** object as the first argument and any “captured” values from the route as keyword arguments. We’ll give an example of this in a bit.

### [path()](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.path) argument: kwargs[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/" \l "path-argument-kwargs" \o "Permalink to this headline)

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.

### [path()](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.path) argument: name[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial01/#path-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/5.0/intro/tutorial02/>

## **Database setup**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#database-setup)

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

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

If you wish to use another database, install the appropriate [database bindings](https://docs.djangoproject.com/en/5.0/topics/install/#database-installation) and change the following keys in the [**DATABASES**](https://docs.djangoproject.com/en/5.0/ref/settings/#std-setting-DATABASES) **'default'** item to match your database connection settings:

* [**ENGINE**](https://docs.djangoproject.com/en/5.0/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/5.0/ref/databases/#third-party-notes).
* [**NAME**](https://docs.djangoproject.com/en/5.0/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/5.0/ref/settings/#std-setting-NAME) should be the full absolute path, including filename, of that file. The default value, **BASE\_DIR / 'db.sqlite3'**, will store the file in your project directory.

While you’re editing **mysite/settings.py**, set [**TIME\_ZONE**](https://docs.djangoproject.com/en/5.0/ref/settings/#std-setting-TIME_ZONE) to your time zone.

Also, note the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/5.0/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/5.0/ref/settings/#std-setting-INSTALLED_APPS) contains the following apps, all of which come with Django:

* [**django.contrib.admin**](https://docs.djangoproject.com/en/5.0/ref/contrib/admin/#module-django.contrib.admin) – The admin site. You’ll use it shortly.
* [**django.contrib.auth**](https://docs.djangoproject.com/en/5.0/topics/auth/#module-django.contrib.auth) – An authentication system.
* [**django.contrib.contenttypes**](https://docs.djangoproject.com/en/5.0/ref/contrib/contenttypes/#module-django.contrib.contenttypes) – A framework for content types.
* [**django.contrib.sessions**](https://docs.djangoproject.com/en/5.0/topics/http/sessions/#module-django.contrib.sessions) – A session framework.
* [**django.contrib.messages**](https://docs.djangoproject.com/en/5.0/ref/contrib/messages/#module-django.contrib.messages) – A messaging framework.
* [**django.contrib.staticfiles**](https://docs.djangoproject.com/en/5.0/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 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/5.0/ref/django-admin/#django-admin-migrate) command looks at the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/5.0/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.

## **Creating models**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#creating-models)

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

**Philosophy**

A model is the single, definitive source of information 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/5.0/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 a history that Django can roll through to update your database schema to match your current models.

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

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

polls/models.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#id2)

**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)

Here, each model is represented by a class that subclasses **[django.db.models.Model](https://docs.djangoproject.com/en/5.0/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/5.0/ref/models/fields/#django.db.models.Field) class – e.g., **[CharField](https://docs.djangoproject.com/en/5.0/ref/models/fields/" \l "django.db.models.CharField" \o "django.db.models.CharField)** for character fields and **[DateTimeField](https://docs.djangoproject.com/en/5.0/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.

The name of each [**Field**](https://docs.djangoproject.com/en/5.0/ref/models/fields/#django.db.models.Field) instance (e.g. **question\_text** or **pub\_date**) is the field’s name, in machine-friendly format. You’ll use this value in your Python code, and your database will use it as the column name.

You can use an optional first positional argument to a [**Field**](https://docs.djangoproject.com/en/5.0/ref/models/fields/#django.db.models.Field) to designate a human-readable name. That’s used in a couple of introspective parts of Django, and it doubles as documentation. If this field isn’t provided, Django will use the machine-readable name. In this example, we’ve only defined a human-readable name for **Question.pub\_date**. For all other fields in this model, the field’s machine-readable name will suffice as its human-readable name.

Some [**Field**](https://docs.djangoproject.com/en/5.0/ref/models/fields/#django.db.models.Field) classes have required arguments. **[CharField](https://docs.djangoproject.com/en/5.0/ref/models/fields/" \l "django.db.models.CharField" \o "django.db.models.CharField)**, for example, requires that you give it a **[max\_length](https://docs.djangoproject.com/en/5.0/ref/models/fields/" \l "django.db.models.CharField.max_length" \o "django.db.models.CharField.max_length)**. That’s used not only in the database schema, but in validation, as we’ll soon see.

A [**Field**](https://docs.djangoproject.com/en/5.0/ref/models/fields/#django.db.models.Field) can also have various optional arguments; in this case, we’ve set the [**default**](https://docs.djangoproject.com/en/5.0/ref/models/fields/#django.db.models.Field.default) value of **votes** to 0.

Finally, note a relationship is defined, using [**ForeignKey**](https://docs.djangoproject.com/en/5.0/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**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#activating-models)

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

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

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

**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/5.0/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/5.0/ref/settings/#std-setting-INSTALLED_APPS) setting. It’ll look like this:

mysite/settings.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#id3)

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:

/ 

**...\>** py manage.py makemigrations polls

You should see something similar to the following:

Migrations for 'polls':

polls/migrations/0001\_initial.py

- Create model Question

- Create model Choice

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.

Migrations are how Django stores changes to your models (and thus your database schema) - they’re files on disk. You can read the migration for your new model if you like; it’s the file **polls/migrations/0001\_initial.py**. Don’t worry, you’re not expected to read them every time Django makes one, but they’re designed to be human-editable in case you want to manually tweak how Django changes things.

There’s a command that will run the migrations for you and manage your database schema automatically - that’s called [**migrate**](https://docs.djangoproject.com/en/5.0/ref/django-admin/#django-admin-migrate), and we’ll come to it in a moment - but first, let’s see what SQL that migration would run. The **[sqlmigrate](https://docs.djangoproject.com/en/5.0/ref/django-admin/" \l "django-admin-sqlmigrate)** command takes migration names and returns their SQL:

/ 

**...\>** py manage.py sqlmigrate polls 0001

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

**BEGIN**;

*--*

*-- Create model Question*

*--*

**CREATE** **TABLE** "polls\_question" (

"id" bigint **NOT** **NULL** **PRIMARY** **KEY** **GENERATED** **BY** **DEFAULT** **AS** **IDENTITY**,

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

"pub\_date" **timestamp** **with** **time** **zone** **NOT** **NULL**

);

*--*

*-- Create model Choice*

*--*

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

"id" bigint **NOT** **NULL** **PRIMARY** **KEY** **GENERATED** **BY** **DEFAULT** **AS** **IDENTITY**,

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

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

"question\_id" bigint **NOT** **NULL**

);

**ALTER** **TABLE** "polls\_choice"

**ADD** **CONSTRAINT** "polls\_choice\_question\_id\_c5b4b260\_fk\_polls\_question\_id"

**FOREIGN** **KEY** ("question\_id")

**REFERENCES** "polls\_question" ("id")

**DEFERRABLE** **INITIALLY** **DEFERRED**;

**CREATE** **INDEX** "polls\_choice\_question\_id\_c5b4b260" **ON** "polls\_choice" ("question\_id");

**COMMIT**;

Note the following:

* The exact output will vary depending on the database you are using. The example above is generated for PostgreSQL.
* Table names are automatically generated by combining the name of the app (**polls**) and the lowercase name of the model – **question** and **choice**. (You can override this behavior.)
* Primary keys (IDs) are added automatically. (You can override this, too.)
* By convention, Django appends **"\_id"** to the foreign key field name. (Yes, you can override this, as well.)
* The foreign key relationship is made explicit by a **FOREIGN KEY** constraint. Don’t worry about the **DEFERRABLE** parts; it’s telling PostgreSQL to not enforce the foreign key until the end of the transaction.
* It’s tailored to the database you’re using, so database-specific field types such as **auto\_increment** (MySQL), **bigint PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY** (PostgreSQL), or **integer primary key autoincrement** (SQLite) are handled for you automatically. Same goes for the quoting of field names – e.g., using double quotes or single quotes.
* The **[sqlmigrate](https://docs.djangoproject.com/en/5.0/ref/django-admin/" \l "django-admin-sqlmigrate)** command doesn’t actually run the migration on your database - instead, it prints it to the screen so that you can see what SQL Django thinks is required. It’s useful for checking what Django is going to do or if you have database administrators who require SQL scripts for changes.

If you’re interested, you can also run [**python manage.py check**](https://docs.djangoproject.com/en/5.0/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/5.0/ref/django-admin/#django-admin-migrate) again to create those model tables in your database:

/ 

**...\>** py manage.py migrate

Operations to perform:

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

Running migrations:

Rendering model states... DONE

Applying polls.0001\_initial... OK

The [**migrate**](https://docs.djangoproject.com/en/5.0/ref/django-admin/#django-admin-migrate) command takes all the migrations that haven’t been applied (Django tracks which ones are applied using a special table in your database called **django\_migrations**) and runs them against your database - essentially, synchronizing the changes you made to your models with the schema in the database.

Migrations are very powerful and let you change your models over time, as you develop your project, without the need to delete your database or tables and make new ones - it specializes in upgrading your database live, without losing data. We’ll cover them in more depth in a later part of the tutorial, but for now, remember the three-step guide to making model changes:

* Change your models (in **models.py**).
* Run [**python manage.py makemigrations**](https://docs.djangoproject.com/en/5.0/ref/django-admin/#django-admin-makemigrations) to create migrations for those changes
* Run [**python manage.py migrate**](https://docs.djangoproject.com/en/5.0/ref/django-admin/#django-admin-migrate) to apply those changes to the database.

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 usable by other developers and in production.

Read the [django-admin documentation](https://docs.djangoproject.com/en/5.0/ref/django-admin/) for full information on what the **manage.py** utility can do.

## **Playing with the API**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#playing-with-the-api)

Now, let’s hop into the interactive Python shell and play around with the free API Django gives you. To invoke the Python shell, use this command:

/ 

**...\>** py manage.py shell

We’re using this instead of simply typing “python”, because **manage.py** sets the [**DJANGO\_SETTINGS\_MODULE**](https://docs.djangoproject.com/en/5.0/topics/settings/#envvar-DJANGO_SETTINGS_MODULE) environment variable, which gives Django the Python import path to your **mysite/settings.py** file.

Once you’re in the shell, explore the [database API](https://docs.djangoproject.com/en/5.0/topics/db/queries/):

**>>> from** **polls.models** **import** Choice, Question *# 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.

**>>>** 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=datetime.timezone.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 (1)>]>

Wait a minute. **<Question: Question object (1)>** isn’t a helpful representation of this 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/5.0/ref/models/instances/#django.db.models.Model.__str__) method to both **Question** and **Choice**:

polls/models.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#id4)

**from** **django.db** **import** models

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

*# ...*

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

**return** self.question\_text

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

*# ...*

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

**return** self.choice\_text

It’s important to add [**\_\_str\_\_()**](https://docs.djangoproject.com/en/5.0/ref/models/instances/#django.db.models.Model.__str__) methods to your models, not only for your own convenience when dealing with the interactive prompt, but also because objects’ representations are used throughout Django’s automatically-generated admin.

Let’s also add a custom method to this model:

polls/models.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#id5)

**import** **datetime**

**from** **django.db** **import** models

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

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

*# ...*

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

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

Note the addition of **import datetime** and **from django.utils import timezone**, to reference Python’s standard [**datetime**](https://docs.python.org/3/library/datetime.html#module-datetime) module and Django’s time-zone-related utilities in **[django.utils.timezone](https://docs.djangoproject.com/en/5.0/ref/utils/" \l "module-django.utils.timezone" \o "django.utils.timezone: Timezone support.)**, respectively. If you aren’t familiar with time zone handling in Python, you can learn more in the [time zone support docs](https://docs.djangoproject.com/en/5.0/topics/i18n/timezones/).

Save these changes and start a new Python interactive shell by running **python manage.py shell** again:

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

# Make sure our \_\_str\_\_() addition worked.

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

<QuerySet [<Question: What's up?>]>

# Django provides a rich database lookup API that's entirely driven by

# keyword arguments.

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

<QuerySet [<Question: What's up?>]>

**>>>** Question.objects.filter(question\_text\_\_startswith="What")

<QuerySet [<Question: What's up?>]>

# Get the question that was published this year.

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

**>>>** current\_year = timezone.now().year

**>>>** Question.objects.get(pub\_date\_\_year=current\_year)

<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.

**>>>** Question.objects.get(id=2)

Traceback (most recent call last):

...

DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a

# shortcut for primary-key exact lookups.

# The following is identical to Question.objects.get(id=1).

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

<Question: What's up?>

# Make sure our custom method worked.

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

**>>>** q.was\_published\_recently()

True

# Give the Question a couple of Choices. The create call constructs a new

# Choice object, does the INSERT statement, adds the choice to the set

# of available choices and returns the new Choice object. Django creates

# a set (defined as "choice\_set") to hold the "other side" of a ForeignKey

# relation (e.g. a question's choice) which can be accessed via the API.

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

# Display any choices from the related object set -- none so far.

**>>>** q.choice\_set.all()

<QuerySet []>

# Create three choices.

**>>>** q.choice\_set.create(choice\_text="Not much", votes=0)

<Choice: Not much>

**>>>** q.choice\_set.create(choice\_text="The sky", votes=0)

<Choice: The sky>

**>>>** c = q.choice\_set.create(choice\_text="Just hacking again", votes=0)

# Choice objects have API access to their related Question objects.

**>>>** c.question

<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.

**>>>** q.choice\_set.all()

<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

**>>>** q.choice\_set.count()

3

# The API automatically follows relationships as far as you need.

# Use double underscores to separate relationships.

# This works as many levels deep as you want; there's no limit.

# Find all Choices for any question whose pub\_date is in this year

# (reusing the 'current\_year' variable we created above).

**>>>** Choice.objects.filter(question\_\_pub\_date\_\_year=current\_year)

<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

# Let's delete one of the choices. Use delete() for that.

**>>>** c = q.choice\_set.filter(choice\_text\_\_startswith="Just hacking")

**>>>** c.delete()

## **ntroducing the Django Admin**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#introducing-the-django-admin)

**Philosophy**

Generating admin sites for your staff or clients to add, change, and delete content is tedious work that doesn’t require much creativity. For that reason, Django entirely automates creation of admin interfaces for models.

Django was written in a newsroom environment, with a very clear separation between “content publishers” and the “public” site. Site managers use the system to add news stories, events, sports scores, etc., and that content is displayed on the public site. Django solves the problem of creating a unified interface for site administrators to edit content.

The admin isn’t intended to be used by site visitors. It’s for site managers.

### Creating an admin user[¶](https://docs.djangoproject.com/en/5.0/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:

/ 

**...\>** py 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.

### Start the development server[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#start-the-development-server)

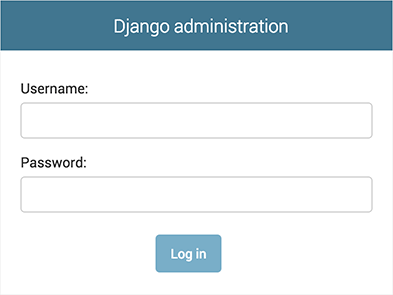
The Django admin site is activated by default. Let’s start the development server and explore it.

If the server is not running start it like so:

/ 

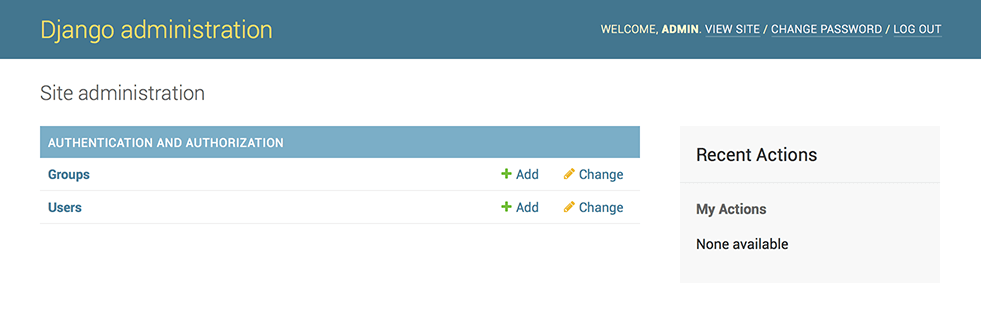
**...\>** py manage.py runserver

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:



### Enter the admin site[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#enter-the-admin-site)

Now, try logging in with the superuser account you created in the previous step. 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/5.0/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[¶](https://docs.djangoproject.com/en/5.0/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.

Only one more 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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial02/#id6)

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

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

admin.site.register(Question)

<https://docs.djangoproject.com/en/5.0/intro/tutorial03/>

# 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 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).

To get from a URL to a view, Django uses what are known as ‘URLconfs’. A URLconf maps URL patterns to views.

## **Writing more views**[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id2)

**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 [**path()**](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.path) calls:

polls/urls.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id3)

**from** **django.urls** **import** path

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

urlpatterns = [

*# ex: /polls/*

path("", views.index, name="index"),

*# ex: /polls/5/*

path("<int:question\_id>/", views.detail, name="detail"),

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

path("<int:question\_id>/results/", views.results, name="results"),

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

path("<int:question\_id>/vote/", views.vote, name="vote"),

]

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/5.0/ref/settings/#std-setting-ROOT_URLCONF) setting. It finds the variable named **urlpatterns** and traverses the patterns 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 **'<int:question\_id>/'**, resulting in a call to the **detail()** view like so:

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

## **Write views that actually do something**[¶](https://docs.djangoproject.com/en/5.0/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/5.0/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/5.0/topics/http/views/#django.http.Http404). The rest is up to you.

Because it’s convenient, let’s use Django’s own database API, which we covered in [Tutorial 2](https://docs.djangoproject.com/en/5.0/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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id4)

**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/5.0/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/5.0/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/5.0/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 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 different application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the best 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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id5)

{% **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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id6)

**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.

### A shortcut: [render()](https://docs.djangoproject.com/en/5.0/topics/http/shortcuts/#django.shortcuts.render)[¶](https://docs.djangoproject.com/en/5.0/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/5.0/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:

polls/views.py[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id7)

**from** **django.shortcuts** **import** render

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

**def** index(request):

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

context = {"latest\_question\_list": latest\_question\_list}

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

## **Raising a 404 error**[¶](https://docs.djangoproject.com/en/5.0/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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id8)

**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/5.0/topics/http/views/#django.http.Http404) exception if a question with the requested ID doesn’t exist.

polls/templates/polls/detail.html[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id9)

{{ question }}

### A shortcut: [get\_object\_or\_404()](https://docs.djangoproject.com/en/5.0/topics/http/shortcuts/#django.shortcuts.get_object_or_404)[¶](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#a-shortcut-get-object-or-404)

It’s a very common idiom to use [**get()**](https://docs.djangoproject.com/en/5.0/ref/models/querysets/#django.db.models.query.QuerySet.get) and raise [**Http404**](https://docs.djangoproject.com/en/5.0/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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id10)

**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})

**Philosophy**

Why do we use a helper function [**get\_object\_or\_404()**](https://docs.djangoproject.com/en/5.0/topics/http/shortcuts/#django.shortcuts.get_object_or_404) instead of automatically catching the **[ObjectDoesNotExist](https://docs.djangoproject.com/en/5.0/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/5.0/topics/http/views/#django.http.Http404) instead of **[ObjectDoesNotExist](https://docs.djangoproject.com/en/5.0/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/5.0/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/5.0/topics/http/shortcuts/#django.shortcuts.get_list_or_404) function, which works just as [**get\_object\_or\_404()**](https://docs.djangoproject.com/en/5.0/topics/http/shortcuts/#django.shortcuts.get_object_or_404) – except using [**filter()**](https://docs.djangoproject.com/en/5.0/ref/models/querysets/#django.db.models.query.QuerySet.filter) instead of [**get()**](https://docs.djangoproject.com/en/5.0/ref/models/querysets/#django.db.models.query.QuerySet.get). It raises [**Http404**](https://docs.djangoproject.com/en/5.0/topics/http/views/#django.http.Http404) if the list is empty.

## **Use the template system**[¶](https://docs.djangoproject.com/en/5.0/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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id11)

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

<**ul**>

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

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

{% **endfor** %}

</**ul**>

## **Removing hardcoded URLs in templates**[¶](https://docs.djangoproject.com/en/5.0/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 [**path()**](https://docs.djangoproject.com/en/5.0/ref/urls/#django.urls.path) 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*

path("<int:question\_id>/", views.detail, name="detail"),

...

## **Namespacing URL names**[¶](https://docs.djangoproject.com/en/5.0/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[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id12)

**from** **django.urls** **import** path

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

app\_name = "polls"

urlpatterns = [

path("", views.index, name="index"),

path("<int:question\_id>/", views.detail, name="detail"),

path("<int:question\_id>/results/", views.results, name="results"),

path("<int:question\_id>/vote/", views.vote, name="vote"),

]

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

polls/templates/polls/index.html[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id13)

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

to point at the namespaced detail view:

polls/templates/polls/index.html[**¶**](https://docs.djangoproject.com/en/5.0/intro/tutorial03/#id14)

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

# Writing your first Django app, part 4