
2019-django2.1-tutorial-note

Documentation

Release latest

Dec 29, 2018

Contents

1	Table of Contents	3
1.1	Writing your first Django app, part 1	3
1.2	Writing your first Django app, part 2	9
1.3	Writing your first Django app, part 3	21
1.4	Writing your first Django app, part 4	28
1.5	Writing your first Django app, part 5	33
1.6	Writing your first Django app, part 6	43
1.7	Writing your first Django app, part 7	45

Read The Doc

<https://readthedocs.org/>

Django

<https://github.com/django/django>

.txt .rst

Note

Note: This is note text. Use a note for information you want the user to pay particular attention to.

If note text runs over a line, make sure the lines wrap and are indented to the same level as the note tag. If formatting is incorrect, part of the note might not render in the HTML output.

Notes can have more than one paragraph. Successive paragraphs must indent to the same level as the rest of the note.

Warning

Warning: This is warning text. Use a warning for information the user must understand to avoid negative consequences.

Warnings are formatted in the same way as notes. In the same way, lines must be broken and indented under the warning tag.

2018-12-28PDF Note Warning cs50

<https://cs50.readthedocs.io/render50/>

- To know which Python you're using, it applies to virtual venv as well
\$ which python

1.1 Writing your first Django app, part 1

Warning: <https://docs.djangoproject.com/en/2.1/intro/tutorial01/> (by Mark

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.

We'll assume you have Django installed already. You can tell Django is installed and which version by running the following command in a shell prompt (indicated by the \$ prefix):

```
$ python -m django --version
```



```
$ django-admin startproject mysite
```



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

```
.. console::
```

```
$ python -m django --version
```

Warning: \$ python -m django --version .. console:
<https://docs.djangoproject.com/en/2.1/intro/tutorial01/> ,js
2018-12-29 10:09, by Mark

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

This tutorial is written for Django latest, which supports Python 3.5 and later. If the Django version doesn't match, you can refer to the tutorial for your version of Django by using the version switcher at the bottom right corner of this page, or update Django to the newest version. If you're using an older version of Python, check [faq-python-version-support](#) to find a compatible version of Django.

See [How to install Django](#) for advice on how to remove older versions of Django and install a newer one.

Where to get help:

If you're having trouble going through this tutorial, please post a message to [#django-users](#) or drop by [#django](#) on [irc.freenode.net](#) to chat with other Django users who might be able to help.

1.1.1 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 – 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:

This will create a `mysite` directory in your current directory. If it didn't work, see [troubleshooting-django-admin](#).

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 `:djangadmin:'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 [/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 in the official Python docs.
- `mysite/settings.py`: Settings/configuration for this Django project. [/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 [/topics/http/urls](#).
- `mysite/wsgi.py`: An entry-point for WSGI-compatible web servers to serve your project. See [/howto/deployment/wsgi/index](#) for more details.

1.1.2 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:

You'll see the following output on the command line:

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

```
You have unapplied migrations; your app may not work properly until they are
↳ applied.
```

```
Run 'python manage.py migrate' to apply them.
```

```
Dec 29, 2018 - 15:50:53
```

```
Django version latest, using settings 'mysite.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

Quit the server with `CONTROL-C`.

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

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.)

Now that the server's running, visit <http://127.0.0.1:8000/> with your Web browser. You'll see a "Congratulations!" page, with a rocket taking off. It worked!

Changing the port

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

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

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

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

Automatic reloading of `:djadmin:runserver`

The development server automatically reloads Python code for each request as needed. You don't need to restart the server for code changes to take effect. However, some actions like adding files don't trigger a restart, so you'll have to restart the server in these cases.

1.1.3 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 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. 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:

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
```

This directory structure will house the poll application.

1.1.4 Write your first view

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

Listing 1: `polls/views.py`

```
from django.http import HttpResponseRedirect

def index(request):
    return HttpResponseRedirect("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:

Listing 2: `polls/urls.py`

```
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()` in the `urlpatterns` list, so you have:

Listing 3: mysite/urls.py

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

The `include()` function allows referencing other URLconfs. Whenever Django encounters `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()` 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()`

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. Lets verify it's working, run the following command:

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()` 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()` 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()` argument: `view`

When Django finds a matching pattern, it calls the specified view function with an `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 () 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.

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.

When you're comfortable with the basic request and response flow, read *part 2 of this tutorial* to start working with the database.

1.2 Writing your first Django app, part 2

Warning: <https://docs.djangoproject.com/en/2.1/intro/tutorial02/> (by Mark)

This tutorial begins where *Tutorial 1* left off. We'll setup the database, create your first model, and get a quick introduction to Django's automatically-generated admin site.

Note: (by Mark)

1.2.1 Database setup

Note: SQLite3, PostGres Oracle, Oracle (by Mark)

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

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

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

- **:setting:'ENGINE'** **<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.
- **:setting:'NAME'** – The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, **: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 **:setting:'USER'**, **:setting:'PASSWORD'**, and **:setting:'HOST'** must be added. For more details, see the reference documentation for **: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 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 **:setting:'TIME_ZONE'** to your time zone.

Also, note the **: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, **:setting:'INSTALLED_APPS'** contains the following apps, all of which come with Django:

- `django.contrib.admin` – The admin site. You'll use it shortly.
- `django.contrib.auth` – An authentication system.
- `django.contrib.contenttypes` – A framework for content types.
- `django.contrib.sessions` – A session framework.
- `django.contrib.messages` – A messaging framework.
- `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:

The **:djangadmin:'migrate'** command looks at the **: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 FROM USER_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 **:setting:'INSTALLED_APPS'** before running **:djangadmin:'migrate'**. The **:djangadmin:'migrate'** command will only run migrations for apps in **:setting:'INSTALLED_APPS'**.

1.2.2 Creating models

Note: `views.py` `admin.py` copy/paste `list_display` = (`['order-code', 'code', 'useraddr1', 'useraddr2', 'useraddr3', 'note',]`) (by Mark

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

Philosophy

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. 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.

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

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

Listing 4: `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)
```

Note:

```
question = models.ForeignKey(Question, on_delete=models.CASCADE)
```

Choice

```
member = models.ForeignKey(Member, on_delete=models.CASCADE)
```

```
on_delete=models.CASCADE (by Mark
```

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

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

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

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

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

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

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

1.2.3 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 `: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 `:setting:INSTALLED_APPS` setting. It'll look like this:

Listing 5: `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',  
]
```

Warning: `'polls.apps.PollsConfig'`, `'polls'` (by Mark)

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

You should see something similar to the following:

```
Migrations for 'polls':  
  polls/migrations/0001_initial.py:  
    - Create model Choice  
    - Create model Question  
    - Add field question to choice
```

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 just 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 `:django:manage migrate`, and we'll come to it in a moment - but first, let's see what SQL that migration would run. The `:django:manage sqlmigrate` command takes migration names and returns their SQL:

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

```
BEGIN;
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
--
-- Create model Question
--
CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
--
-- Add field question to choice
--
ALTER TABLE "polls_choice" ADD COLUMN "question_id" integer NOT NULL;
ALTER TABLE "polls_choice" ALTER COLUMN "question_id" DROP DEFAULT;
CREATE INDEX "polls_choice_7aa0f6ee" ON "polls_choice" ("question_id");
ALTER TABLE "polls_choice"
    ADD CONSTRAINT "polls_choice_question_id_246c99a640fbbd72_fk_polls_question_id"
        FOREIGN KEY ("question_id")
        REFERENCES "polls_question" ("id")
        DEFERRABLE INITIALLY DEFERRED;
COMMIT;
```

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; that's just 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), `serial` (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 `:django:manage sqlmigrate` command doesn't actually run the migration on your database - it just 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 `:djangadmin:'python manage.py check <check>'`; this checks for any problems in your project without making migrations or touching the database.

Now, run `:djangadmin:'migrate'` again to create those model tables in your database:

The `:djangadmin:'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 `:djangadmin:'python manage.py makemigrations <makemigrations>'` to create migrations for those changes
- Run `:djangadmin:'python manage.py migrate <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 for full information on what the `manage.py` utility can do.

1.2.4 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:

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.

Once you're in the shell, explore the database API:

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
# 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 (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__()` method to both `Question` and `Choice`:

Listing 6: `polls/models.py`

```
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__()` 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.

Note these are normal Python methods. Let's add a custom method, just for demonstration:

Listing 7: `polls/models.py`

```
import datetime

from django.db import models
from django.utils import timezone

class Question(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Note the addition of `import datetime` and `from django.utils import timezone`, to reference Python's standard `datetime` module and Django's time-zone-related utilities in `django.utils.timezone`, respectively. If you aren't familiar with time zone handling in Python, you can learn more in the [time zone support docs](#).

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 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
```

(continues on next page)

(continued from previous page)

```
<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()
```

For more information on model relations, see [Accessing related objects](#). For more on how to use double underscores to perform field lookups via the API, see [Field lookups](#). For full details on the database API, see our [Database API reference](#).

1.2.5 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

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

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.

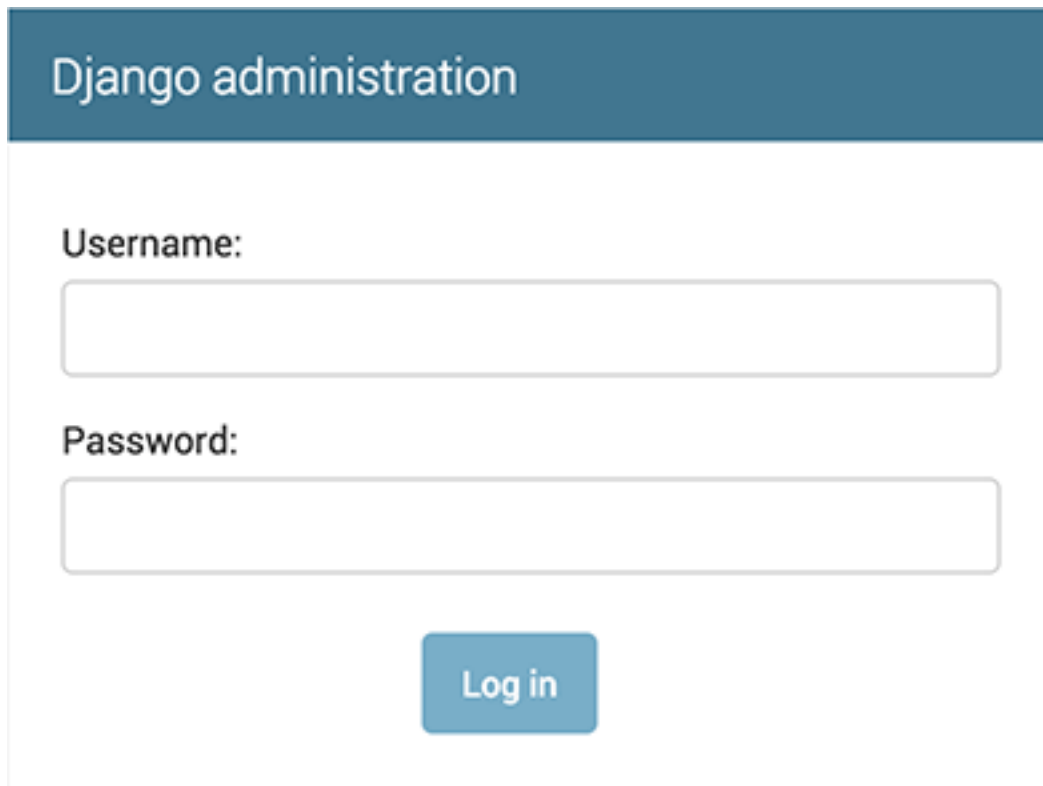
```
Password: *****
Password (again): *****
Superuser created successfully.
```

Start the development server

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:

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:



The image shows the Django administration login interface. It features a dark blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". At the bottom center, there is a blue button labeled "Log in".

Since translation is turned on by default, the login screen may be displayed in your own language, depending on your browser's settings and if Django has a translation for this language.

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:

Django administration

WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Site administration

AUTHENTICATION AND AUTHORIZATION	
Groups	+ Add Change
Users	+ Add Change

Recent Actions

My Actions

None available

You should see a few types of editable content: groups and users. They are provided by `django.contrib.auth`, the authentication framework shipped by Django.

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:

Listing 8: `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:

Site administration

AUTHENTICATION AND AUTHORIZATION	
Groups	+ Add Change
Users	+ Add Change

POLLS	
Questions	+ Add Change

Recent Actions

My Actions

None available

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:

Home > Polls > Questions

Select question to change ADD QUESTION +

Action: 0 of 1 selected

QUESTION

What's up?

1 question

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

Home > Polls > Questions > What's up?

Change question HISTORY

Question text:

Date published:

Date: Today

Time: Now

Things to note here:

- The form is automatically generated from the `Question` model.
- The different model field types (`DateTimeField`, `CharField`) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.
- Each `DateTimeField` gets free JavaScript shortcuts. Dates get a “Today” shortcut and calendar popup, and times get a “Now” shortcut and a convenient popup that lists commonly entered times.

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.

If the value of “Date published” doesn’t match the time when you created the question in *Tutorial 1*, it probably means you forgot to set the correct value for the `:setting:‘TIME_ZONE’` setting. Change it, reload the page and check that the correct value appears.

Change the “Date published” by clicking the “Today” and “Now” shortcuts. Then click “Save and continue editing.” Then click “History” in the upper right. You’ll see a page listing all changes made to this object via the Django admin, with the timestamp and username of the person who made the change:

Home › Polls › Questions › What's up? › History

Change history: What's up?

DATE/TIME	USER	ACTION
Sept. 6, 2015, 9:21 p.m.	elky	Changed pub_date.

When you're comfortable with the models API and have familiarized yourself with the admin site, read *part 3 of this tutorial* to learn about how to add more views to our polls app.

1.3 Writing your first Django app, part 3

Warning: <https://docs.djangoproject.com/en/2.1/intro/tutorial03/> (by Mark)

This tutorial begins where *Tutorial 2* left off. We're continuing the Web-poll application and will focus on creating the public interface – “views.”

1.3.1 Overview

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).

Now in your time on the web you may have come across such beauties as “ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B”. You will be pleased to know that Django allows us much more elegant *URL patterns* than that.

A URL pattern is simply the general form of a URL - for example: `/newsarchive/<year>/<month>/`.

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

This tutorial provides basic instruction in the use of URLconfs, and you can refer to </topics/http/urls> for more information.

1.3.2 Writing more views

Note: (by Mark

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

Listing 9: `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 `path()` calls:

Listing 10: `polls/urls.py`

```
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'),
]
```

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

When somebody requests a page from your website – say, `/polls/34/`, Django will load the `mysite.urls` Python module because it's pointed to by the **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)
```

The `question_id=34` part comes from `<int:question_id>`. Using angle brackets “captures” part of the URL and sends it as a keyword argument to the view function. The `:question_id` part of the string defines the name

that will be used to identify the matched pattern, and the `<int>` part is a converter that determines what patterns should match this part of the URL path.

There's no need to add URL cruft such as `.html` – unless you want to, in which case you can do something like this:

```
path('polls/latest.html', views.index),
```

But, don't do that. It's silly.

1.3.3 Write views that actually do something

Warning: (by Mark

Each view is responsible for doing one of two things: returning an `HttpResponse` object containing the content for the requested page, or raising an exception such as `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`. Or an exception.

Because it's convenient, let's use Django's own database API, which we covered in *Tutorial 2*. 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:

Listing 11: `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 **setting:** `'TEMPLATES'` setting describes how Django will load and render templates. The default settings file configures a `DjangoTemplates` backend whose **setting:** `'APP_DIRS <TEMPLATES-APP_DIRS>'` option is set to `True`. By convention `DjangoTemplates` looks for a "templates" subdirectory in each of the **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 *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by *namespacing* them. That is, by putting those templates inside *another* directory named for the application itself.

Put the following code in that template:

Listing 12: `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:

Listing 13: `polls/views.py`

```
from django.http import HttpResponseRedirect
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 HttpResponseRedirect(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*. The link points to the question’s detail page.

A shortcut: `render()`

Note: (by Mark

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

Listing 14: 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` and `HttpResponse` (you'll want to keep `HttpResponse` if you still have the stub methods for `detail`, `results`, and `vote`).

The `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` object of the given template rendered with the given context.

1.3.4 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:

Listing 15: polls/views.py

```

from django.http import Http404
from django.shortcuts import render

from .models import Question
# ...
def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})

```

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

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

Listing 16: polls/templates/polls/detail.html

```

{{ question }}

```

will get you started for now.

A shortcut: `get_object_or_404()`

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

Listing 17: 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()` function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the `get()` function of the model's manager. It raises `Http404` if the object doesn't exist.

Philosophy

Why do we use a helper function `get_object_or_404()` instead of automatically catching the `ObjectDoesNotExist` exceptions at a higher level, or having the model API raise `Http404` instead of `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` module.

There's also a `get_list_or_404()` function, which works just as `get_object_or_404()` – except using `filter()` instead of `get()`. It raises `Http404` if the list is empty.

1.3.5 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:

Listing 18: 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 `:ttag:'{% for %}<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 `:ttag:'{% for %}<for>'` tag.

See the template guide for more about templates.

1.3.6 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()` 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'),
...
```

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'
path('specifics/<int:question_id>/', views.detail, name='detail'),
...
```

1.3.7 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:

Listing 19: `polls/urls.py`

```
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:

Listing 20: `polls/templates/polls/index.html`

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

to point at the namespaced detail view:

Listing 21: polls/templates/polls/index.html

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

When you're comfortable with writing views, read *part 4 of this tutorial* to learn about simple form processing and generic views.

1.4 Writing your first Django app, part 4

Warning: <https://docs.djangoproject.com/en/2.1/intro/tutorial04/> (by Mark)

This tutorial begins where *Tutorial 3* left off. We're continuing the Web-poll application and will focus on simple form processing and cutting down our code.

1.4.1 Write a simple form

Warning: detail.html <form action='xxx' method='post'> Django form class (by Mark)

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

Listing 22: 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 `:ttag:'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 `:ttag:'{% csrf_token %}<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](#), we created a URLconf for the polls application that includes this line:

Listing 23: polls/urls.py

```
path('<int:question_id>/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`:

Listing 24: polls/views.py

```
from django.http import HttpResponseRedirect, HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse

from .models import Choice, Question
# ...
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

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

- `request.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` values are always strings.

Note that Django also provides `request.GET` for accessing GET data in the same way – but we're explicitly using `request.POST` in our code, to ensure that data is only altered via a POST call.

- `request.POST['choice']` will raise `KeyError` if `choice` wasn't provided in POST data. The above code checks for `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` rather than a normal `HttpResponse`. `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` 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()` function in the `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*, this `reverse()` call will return a string like

```
'/polls/3/results/'
```

where the `3` is the value of `question.id`. This redirected URL will then call the `'results'` view to display the final page.

As mentioned in *Tutorial 3*, `request` is an `HttpRequest` object. For more on `HttpRequest` objects, see the request and response documentation.

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

Listing 25: `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})
```

This is almost exactly the same as the `detail()` view from *Tutorial 3*. The only difference is the template name. We'll fix this redundancy later.

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

Listing 26: `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>
```

Now, go to `/polls/1/` in your browser and vote in the question. You should see a results page that gets updated each time you vote. If you submit the form without having chosen a choice, you should see the error message.

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](#) to learn how you can solve this issue.

1.4.2 Use generic views: Less code is better

The `detail()` (from *Tutorial 3*) 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

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

Listing 27: `polls/urls.py`

```
from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.IndexView.as_view(), name='index'),
    path('<int:pk>/', views.DetailView.as_view(), name='detail'),
    path('<int:pk>/results/', views.ResultsView.as_view(), name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Note that the name of the matched pattern in the path strings of the second and third patterns has changed from `<question_id>` to `<pk>`.

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:

Listing 28: polls/views.py

```
from django.http import HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
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` and `DetailView`. Respectively, those two views abstract the concepts of “display a list of objects” and “display a detail page for a particular type of object.”

- Each generic view needs to know what model it will be acting upon. This is provided using the `model` attribute.
- The `DetailView` generic view expects the primary key value captured from the URL to be called “pk”, so we’ve changed `question_id` to `pk` for the generic views.

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

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

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.

When you're comfortable with forms and generic views, read *part 5 of this tutorial* to learn about testing our polls app.

1.5 Writing your first Django app, part 5

Warning: <https://docs.djangoproject.com/en/2.1/intro/tutorial05/>

This tutorial begins where *Tutorial 4* left off. We've built a Web-poll application, and we'll now create some automated tests for it.

1.5.1 Introducing automated testing

What are automated tests?

Tests are simple routines that check the operation of your code.

Testing operates at different levels. Some tests might apply to a tiny detail (*does a particular model method return values as expected?*) while others examine the overall operation of the software (*does a sequence of user inputs on the site produce the desired result?*). That's no different from the kind of testing you did earlier in *Tutorial 2*, using the `:djadmin:shell` to examine the behavior of a method, or running the application and entering data to check how it behaves.

What's different in *automated* tests is that the testing work is done for you by the system. You create a set of tests once, and then as you make changes to your app, you can check that your code still works as you originally intended, without having to perform time consuming manual testing.

Why you need to create tests

So why create tests, and why now?

You may feel that you have quite enough on your plate just learning Python/Django, and having yet another thing to learn and do may seem overwhelming and perhaps unnecessary. After all, our polls application is working quite happily now; going through the trouble of creating automated tests is not going to make it work any better. If creating the polls application is the last bit of Django programming you will ever do, then true, you don't need to know how to create automated tests. But, if that's not the case, now is an excellent time to learn.

Tests will save you time

Up to a certain point, 'checking that it seems to work' will be a satisfactory test. In a more sophisticated application, you might have dozens of complex interactions between components.

A change in any of those components could have unexpected consequences on the application's behavior. Checking that it still 'seems to work' could mean running through your code's functionality with twenty different variations of your test data just to make sure you haven't broken something - not a good use of your time.

That's especially true when automated tests could do this for you in seconds. If something's gone wrong, tests will also assist in identifying the code that's causing the unexpected behavior.

Sometimes it may seem a chore to tear yourself away from your productive, creative programming work to face the unglamorous and unexciting business of writing tests, particularly when you know your code is working properly.

However, the task of writing tests is a lot more fulfilling than spending hours testing your application manually or trying to identify the cause of a newly-introduced problem.

Tests don't just identify problems, they prevent them

It's a mistake to think of tests merely as a negative aspect of development.

Without tests, the purpose or intended behavior of an application might be rather opaque. Even when it's your own code, you will sometimes find yourself poking around in it trying to find out what exactly it's doing.

Tests change that; they light up your code from the inside, and when something goes wrong, they focus light on the part that has gone wrong - *even if you hadn't even realized it had gone wrong*.

Tests make your code more attractive

You might have created a brilliant piece of software, but you will find that many other developers will simply refuse to look at it because it lacks tests; without tests, they won't trust it. Jacob Kaplan-Moss, one of Django's original developers, says "Code without tests is broken by design."

That other developers want to see tests in your software before they take it seriously is yet another reason for you to start writing tests.

Tests help teams work together

The previous points are written from the point of view of a single developer maintaining an application. Complex applications will be maintained by teams. Tests guarantee that colleagues don't inadvertently break your code (and that you don't break theirs without knowing). If you want to make a living as a Django programmer, you must be good at writing tests!

1.5.2 Basic testing strategies

There are many ways to approach writing tests.

Some programmers follow a discipline called "[test-driven development](#)"; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it's similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development simply formalizes the problem in a Python test case.

More often, a newcomer to testing will create some code and later decide that it should have some tests. Perhaps it would have been better to write some tests earlier, but it's never too late to get started.

Sometimes it's difficult to figure out where to get started with writing tests. If you have written several thousand lines of Python, choosing something to test might not be easy. In such a case, it's fruitful to write your first test the next time you make a change, either when you add a new feature or fix a bug.

So let's do that right away.

1.5.3 Writing our first test

We identify a bug

Fortunately, there's a little bug in the `polls` application for us to fix right away: the `Question.was_published_recently()` method returns `True` if the `Question` was published within the last day (which is correct) but also if the `Question`'s `pub_date` field is in the future (which certainly isn't).

Confirm the bug by using the `:djadmin:shell` to check the method on a question whose date lies in the future:

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

Since things in the future are not 'recent', this is clearly wrong.

Create a test to expose the bug

What we've just done in the `:djadmin:shell` to test for the problem is exactly what we can do in an automated test, so let's turn that into an automated test.

A conventional place for an application's tests is in the application's `tests.py` file; the testing system will automatically find tests in any file whose name begins with `test`.

Put the following in the `tests.py` file in the `polls` application:

Listing 29: `polls/tests.py`

```
import datetime

from django.test import TestCase
from django.utils import timezone

from .models import Question

class QuestionModelTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() returns False for questions whose pub_date
        is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(), False)
```

Here we have created a `django.test.TestCase` subclass with a method that creates a `Question` instance with a `pub_date` in the future. We then check the output of `was_published_recently()` - which *ought* to be `False`.

Running tests

In the terminal, we can run our test:

and you'll see something like:

```

Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
=====
FAIL: test_was_published_recently_with_future_question (polls.tests.
↳QuestionModelTests)
-----
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in test_was_published_recently_with_
↳future_question
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False
-----

Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...

```

What happened is this:

- `manage.py test polls` looked for tests in the `polls` application
- it found a subclass of the `django.test.TestCase` class
- it created a special database for the purpose of testing
- it looked for test methods - ones whose names begin with `test`
- in `test_was_published_recently_with_future_question` it created a `Question` instance whose `pub_date` field is 30 days in the future
- ... and using the `assertIs()` method, it discovered that its `was_published_recently()` returns `True`, though we wanted it to return `False`

The test informs us which test failed and even the line on which the failure occurred.

Fixing the bug

We already know what the problem is: `Question.was_published_recently()` should return `False` if its `pub_date` is in the future. Amend the method in `models.py`, so that it will only return `True` if the date is also in the past:

Listing 30: `polls/models.py`

```

def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now

```

and run the test again:

```

Creating test database for alias 'default'...
System check identified no issues (0 silenced).

```

(continues on next page)

(continued from previous page)

```

.
-----
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

After identifying a bug, we wrote a test that exposes it and corrected the bug in the code so our test passes.

Many other things might go wrong with our application in the future, but we can be sure that we won't inadvertently reintroduce this bug, because simply running the test will warn us immediately. We can consider this little portion of the application pinned down safely forever.

More comprehensive tests

While we're here, we can further pin down the `was_published_recently()` method; in fact, it would be positively embarrassing if in fixing one bug we had introduced another.

Add two more test methods to the same class, to test the behavior of the method more comprehensively:

Listing 31: `polls/tests.py`

```

def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() returns False for questions whose pub_date
    is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=1, seconds=1)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() returns True for questions whose pub_date
    is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=23, minutes=59, seconds=59)
    recent_question = Question(pub_date=time)
    self.assertIs(recent_question.was_published_recently(), True)
```

And now we have three tests that confirm that `Question.was_published_recently()` returns sensible values for past, recent, and future questions.

Again, `polls` is a simple application, but however complex it grows in the future and whatever other code it interacts with, we now have some guarantee that the method we have written tests for will behave in expected ways.

1.5.4 Test a view

The `polls` application is fairly indiscriminating: it will publish any question, including ones whose `pub_date` field lies in the future. We should improve this. Setting a `pub_date` in the future should mean that the `Question` is published at that moment, but invisible until then.

A test for a view

When we fixed the bug above, we wrote the test first and then the code to fix it. In fact that was a simple example of test-driven development, but it doesn't really matter in which order we do the work.

In our first test, we focused closely on the internal behavior of the code. For this test, we want to check its behavior as it would be experienced by a user through a web browser.

Before we try to fix anything, let's have a look at the tools at our disposal.

The Django test client

Django provides a test `Client` to simulate a user interacting with the code at the view level. We can use it in `tests.py` or even in the `:djangadmin:'shell'`.

We will start again with the `:djangadmin:'shell'`, where we need to do a couple of things that won't be necessary in `tests.py`. The first is to set up the test environment in the `:djangadmin:'shell'`:

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

`setup_test_environment()` installs a template renderer which will allow us to examine some additional attributes on responses such as `response.context` that otherwise wouldn't be available. Note that this method *does not* setup a test database, so the following will be run against the existing database and the output may differ slightly depending on what questions you already created. You might get unexpected results if your `TIME_ZONE` in `settings.py` isn't correct. If you don't remember setting it earlier, check it before continuing.

Next we need to import the test client class (later in `tests.py` we will use the `django.test.TestCase` class, which comes with its own client, so this won't be required):

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

With that ready, we can ask the client to do some work for us:

```
>>> # get a response from '/'
>>> response = client.get('/')
Not Found: /
>>> # we should expect a 404 from that address; if you instead see an
>>> # "Invalid HTTP_HOST header" error and a 400 response, you probably
>>> # omitted the setup_test_environment() call described earlier.
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n    <ul>\n    \n        <li><a href="/polls/1/">What's up?</a></li>\n    \n    \n    </ul>\n\n'
>>> response.context['latest_question_list']
<QuerySet [<Question: What's up?>>>
```

Improving our view

The list of polls shows polls that aren't published yet (i.e. those that have a `pub_date` in the future). Let's fix that.

In *Tutorial 4* we introduced a class-based view, based on `ListView`:

Listing 32: `polls/views.py`

```
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]
```

We need to amend the `get_queryset()` method and change it so that it also checks the date by comparing it with `timezone.now()`. First we need to add an import:

Listing 33: `polls/views.py`

```
from django.utils import timezone
```

and then we must amend the `get_queryset` method like so:

Listing 34: `polls/views.py`

```
def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

`Question.objects.filter(pub_date__lte=timezone.now())` returns a `queryset` containing Questions whose `pub_date` is less than or equal to - that is, earlier than or equal to - `timezone.now`.

Testing our new view

Now you can satisfy yourself that this behaves as expected by firing up `runserver`, loading the site in your browser, creating `Questions` with dates in the past and future, and checking that only those that have been published are listed. You don't want to have to do that *every single time you make any change that might affect this* - so let's also create a test, based on our `:djadmin:shell` session above.

Add the following to `polls/tests.py`:

Listing 35: `polls/tests.py`

```
from django.urls import reverse
```

and we'll create a shortcut function to create questions as well as a new test class:

Listing 36: polls/tests.py

```
def create_question(question_text, days):
    """
    Create a question with the given `question_text` and published the
    given number of `days` offset to now (negative for questions published
    in the past, positive for questions that have yet to be published).
    """
    time = timezone.now() + datetime.timedelta(days=days)
    return Question.objects.create(question_text=question_text, pub_date=time)

class QuestionIndexViewTests(TestCase):
    def test_no_questions(self):
        """
        If no questions exist, an appropriate message is displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_past_question(self):
        """
        Questions with a pub_date in the past are displayed on the
        index page.
        """
        create_question(question_text="Past question.", days=-30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_future_question(self):
        """
        Questions with a pub_date in the future aren't displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        are displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )
)
```

(continues on next page)

(continued from previous page)

```

def test_two_past_questions(self):
    """
    The questions index page may display multiple questions.
    """
    create_question(question_text="Past question 1.", days=-30)
    create_question(question_text="Past question 2.", days=-5)
    response = self.client.get(reverse('polls:index'))
    self.assertEqual(
        response.context['latest_question_list'],
        ['<Question: Past question 2.>', '<Question: Past question 1.>']
    )

```

Let's look at some of these more closely.

First is a question shortcut function, `create_question`, to take some repetition out of the process of creating questions.

`test_no_questions` doesn't create any questions, but checks the message: "No polls are available." and verifies the `latest_question_list` is empty. Note that the `django.test.TestCase` class provides some additional assertion methods. In these examples, we use `assertContains()` and `assertQuerysetEqual()`.

In `test_past_question`, we create a question and verify that it appears in the list.

In `test_future_question`, we create a question with a `pub_date` in the future. The database is reset for each test method, so the first question is no longer there, and so again the index shouldn't have any questions in it.

And so on. In effect, we are using the tests to tell a story of admin input and user experience on the site, and checking that at every state and for every new change in the state of the system, the expected results are published.

Testing the DetailView

What we have works well; however, even though future questions don't appear in the *index*, users can still reach them if they know or guess the right URL. So we need to add a similar constraint to `DetailView`:

Listing 37: `polls/views.py`

```

class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())

```

And of course, we will add some tests, to check that a `Question` whose `pub_date` is in the past can be displayed, and that one with a `pub_date` in the future is not:

Listing 38: `polls/tests.py`

```

class QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        The detail view of a question with a pub_date in the future
        returns a 404 not found.
        """
        future_question = create_question(question_text='Future question.', days=5)
        url = reverse('polls:detail', args=(future_question.id,))

```

(continues on next page)

(continued from previous page)

```
response = self.client.get(url)
self.assertEqual(response.status_code, 404)

def test_past_question(self):
    """
    The detail view of a question with a pub_date in the past
    displays the question's text.
    """
    past_question = create_question(question_text='Past Question.', days=-5)
    url = reverse('polls:detail', args=(past_question.id,))
    response = self.client.get(url)
    self.assertContains(response, past_question.question_text)
```

Ideas for more tests

We ought to add a similar `get_queryset` method to `ResultsView` and create a new test class for that view. It'll be very similar to what we have just created; in fact there will be a lot of repetition.

We could also improve our application in other ways, adding tests along the way. For example, it's silly that `Questions` can be published on the site that have no `Choices`. So, our views could check for this, and exclude such `Questions`. Our tests would create a `Question` without `Choices` and then test that it's not published, as well as create a similar `Question` *with* `Choices`, and test that it *is* published.

Perhaps logged-in admin users should be allowed to see unpublished `Questions`, but not ordinary visitors. Again: whatever needs to be added to the software to accomplish this should be accompanied by a test, whether you write the test first and then make the code pass the test, or work out the logic in your code first and then write a test to prove it.

At a certain point you are bound to look at your tests and wonder whether your code is suffering from test bloat, which brings us to:

1.5.5 When testing, more is better

It might seem that our tests are growing out of control. At this rate there will soon be more code in our tests than in our application, and the repetition is unaesthetic, compared to the elegant conciseness of the rest of our code.

It doesn't matter. Let them grow. For the most part, you can write a test once and then forget about it. It will continue performing its useful function as you continue to develop your program.

Sometimes tests will need to be updated. Suppose that we amend our views so that only `Questions` with `Choices` are published. In that case, many of our existing tests will fail - *telling us exactly which tests need to be amended to bring them up to date*, so to that extent tests help look after themselves.

At worst, as you continue developing, you might find that you have some tests that are now redundant. Even that's not a problem; in testing redundancy is a *good* thing.

As long as your tests are sensibly arranged, they won't become unmanageable. Good rules-of-thumb include having:

- a separate `TestClass` for each model or view
- a separate test method for each set of conditions you want to test
- test method names that describe their function

1.5.6 Further testing

This tutorial only introduces some of the basics of testing. There's a great deal more you can do, and a number of very useful tools at your disposal to achieve some very clever things.

For example, while our tests here have covered some of the internal logic of a model and the way our views publish information, you can use an “in-browser” framework such as [Selenium](#) to test the way your HTML actually renders in a browser. These tools allow you to check not just the behavior of your Django code, but also, for example, of your JavaScript. It's quite something to see the tests launch a browser, and start interacting with your site, as if a human being were driving it! Django includes `LiveServerTestCase` to facilitate integration with tools like Selenium.

If you have a complex application, you may want to run tests automatically with every commit for the purposes of [continuous integration](#), so that quality control is itself - at least partially - automated.

A good way to spot untested parts of your application is to check code coverage. This also helps identify fragile or even dead code. If you can't test a piece of code, it usually means that code should be refactored or removed. Coverage will help to identify dead code. See [topics-testing-code-coverage](#) for details.

Testing in Django has comprehensive information about testing.

1.5.7 What's next?

For full details on testing, see [Testing in Django](#).

When you're comfortable with testing Django views, read [part 6 of this tutorial](#) to learn about static files management.

1.6 Writing your first Django app, part 6

Warning: <https://docs.djangoproject.com/en/2.1/intro/tutorial06/>

This tutorial begins where [Tutorial 5](#) 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.

1.6.1 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 **: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 **: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 *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by *namespacing* them. That is, by putting those static files inside *another* directory named for the application itself.

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

Listing 39: `polls/static/polls/style.css`

```
li a {
    color: green;
}
```

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

Listing 40: `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.

Start the server (or restart it if it's already running):

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.

1.6.2 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`):

Listing 41: `polls/static/polls/style.css`

```
body {
    background: white url("images/background.gif") no-repeat;
}
```

Reload `http://localhost:8000/polls/` and you should see the background loaded in the top left 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 **:setting:'STATIC_URL'** (used by the **:tag:'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 and the staticfiles reference. Deploying static files discusses how to use static files on a real server.

When you're comfortable with the static files, read *part 7 of this tutorial* to learn how to customize Django's automatically-generated admin site.

1.7 Writing your first Django app, part 7

Warning: <https://docs.djangoproject.com/en/2.1/intro/tutorial07/>

This tutorial begins where *Tutorial 6* 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*.

1.7.1 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:

Listing 42: polls/admin.py

```
from django.contrib import admin

from .models import Question

class QuestionAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question_text']

admin.site.register(Question, QuestionAdmin)
```



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:

Home > Polls > Questions > What's up?

Change question

Date published:

Date: Today | 
Time: Now | 

Question text:

This isn't impressive with only two fields, but for admin forms with dozens of fields, choosing an intuitive order is an important usability detail.

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

Listing 43: 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 `fieldset`s is the title of the fieldset. Here's what our form looks like now:

Home > Polls > Questions > What's up?


Change question


Question text:

What's up?

Date information

Date published:

Date: 2015-09-06 Today | 

Time: 21:16:20 Now | 

1.7.2 Adding related objects

OK, we have our `Question` admin page, but a `Question` has multiple `Choices`, 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:

Listing 44: polls/admin.py

```

from django.contrib import admin

from .models import Choice, Question
# ...
admin.site.register(Choice)




```

Now “Choices” is an available option in the Django admin. The “Add choice” form looks like this:

Home > Polls > Choices > Add choice

Add choice

Question:

Choice text:

Votes:

In that form, the “Question” field is a select box containing every question in the database. Django knows that a `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:

Listing 45: polls/admin.py

```

from django.contrib import admin

from .models import Choice, Question

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

```

(continues on next page)

(continued from previous page)

```

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:

Home › Polls › Questions › Add question

Add question

Question text:

Date information (Hide)

Date published: Date: Today 
 Time: Now 

CHOICES

Choice: #1

Choice text:

Votes:

Choice: #2

Choice text:

Votes:

Choice: #3

Choice text:

Votes:

[+ Add another Choice](#)

It works like this: There are three slots for related Choices – as specified by `extra` – and each time you come back

to the “Change” page for an already-created object, you get another three extra slots.

At the end of the three current slots you will find an “Add another Choice” link. If you click on it, a new slot will be added. If you want to remove the added slot, you can click on the X to the top right of the added slot. Note that you can’t remove the original three slots. This image shows an added slot:

CHOICES	
Choice: #1	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #2	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #3	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #4 ✕	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
+ Add another Choice	

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:

Listing 46: 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:

CHOICES		
CHOICE TEXT	VOTES	DELETE?
<input type="text"/>	<input type="text" value="0"/>	
<input type="text"/>	<input type="text" value="0"/>	
<input type="text"/>	<input type="text" value="0"/>	

[+ Add another Choice](#)

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.

1.7.3 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.

Here’s what it looks like at this point:

Home > Polls > Questions

Select question to change ADD QUESTION +

Action: 0 of 1 selected

QUESTION

What's up?

1 question

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` admin option, which is a tuple of field names to display, as columns, on the change list page for the object:

Listing 47: 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*:

Listing 48: polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date', 'was_published_recently')
```

Now the question change list page looks like this:

Home > Polls > Questions

Select question to change

Action: 0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	WAS PUBLISHED RECENTLY
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	False

1 question

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:

Listing 49: 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`.

Edit your `polls/admin.py` file again and add an improvement to the Question change list page: filters using the `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:

Home > Polls > Questions

Select question to change ADD QUESTION +

Action: 0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	PUBLISHED RECENTLY?
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	+

1 question

FILTER

By date published

- Any date
- Today
- Past 7 days
- This month
- This year

The type of filter displayed depends on the type of field you're filtering on. Because `pub_date` is a `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, search boxes, filters, date-hierarchies, and column-header-ordering all work together like you think they should.

1.7.4 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

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 `:setting:'DIRS <TEMPLATES-DIRS>'` option in the `:setting:'TEMPLATES'` setting:

Listing 50: `mysite/settings.py`

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
```

(continues on next page)

(continued from previous page)

```

    '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',
        ],
    },
],
]

```

:setting:‘DIRS <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 *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:

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` attribute to more easily make this particular customization.

This 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](#).

Note that any of Django’s default admin templates can be overridden. To override a template, just do the same thing you did with `base_site.html` – copy it from the default directory into your custom directory, and make changes.

Customizing your *application’s* templates

Astute readers will ask: But if **:setting:‘DIRS <TEMPLATES-DIRS>’** was empty by default, how was Django finding the default admin templates? The answer is that, since **:setting:‘APP_DIRS <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 for more information about how Django finds its templates.

1.7.5 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 `: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.

1.7.6 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.

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](#).