DJANGO. PYTHON.

INTRODUCTION

BASIC DJANGO CONCEPTS

Django implements the MVT (Model View Template) architectural pattern, which is slightly different from the familiar MVC (Model View Controller) on which Ruby on Rails and Laravel work.

Model (Model) The model in Django describes the data schema in the database. With Django ORM, you can independently describe fields and any other data types, and make migrations to simplify development.

View In the Django view, you define the basic logic and algorithms of the application, get various data from the database or manipulate them. The view is usually based on the request\response functions. Response is usually an HTTP redirect, HTTP error(404), MimeTypes, or some kind of template.

A template in Django is a simple HTML code with a special Django template language. DTL (Django Template Language) is a language with which you can dynamically change the content of the page (for example, change the user name on the page, depending on the name of the authorized user).

Settings The settings file in Django, which contains all the settings of your web application. It includes a secret key, folders with templates, middlewares (which are responsible, for example, for ensuring that your private albums are not seen by other users), a database connection, and much more.

The url of the routing configuration file is about the same as in Angular or Laravel. This binds the view to url requests.

Admin Page Since Django was originally designed for rapid prototyping and deployment of news sites, the admin panel is included by default.

CREATING A WEB_SITE USING AN EXAMPLE

STEP 1: SET UP YOUR VIRTUAL ENVIRONMENT AND DJANGO

CREATE A VIRTUAL ENVIRONMENT AND A PROJECT DIRECTORY

Whenever you develop something using Python, and especially if you'll be using external libraries, it's important to create a virtual environment. This way, you create an isolated world for your code so that the Python and library versions that you choose can't accidentally break any other applications that you've written for other versions.

Then, if you later use a different version of Python or an updated library for another project, you won't break anything from this project, because it'll continue to use its own Python and library versions.

Creating a virtual environment involves just a couple of steps.

First, create a directory as the root of your

```
C:\> mkdir projects\todo_list
C:\> cd projects\todo_list
```

With that, you've created your **project root**. Everything that you do in this tutorial will take place inside this folder. You've called it todo_list, and just to keep things tidy, you've put it inside an existing folder named projects/ in your home directory.

Make sure that you have newer installed Python, and then you can use Python's built-in module venv to create and activate your virtual environment:

```
C:\> python -m venv venv
C:\> venv\Scripts\activate.bat
```

The first line creates your virtual environment in the subdirectory venv/. It also copies pip and setuptools into it. The second line **activates** the virtual environment, and your console prompt may change to remind you of that fact.

After activating, you're using a completely independent Python interpreter and ecosystem. Any libraries that you install from now on, including Django, will be isolated to this environment.

Later, when you've finished working in the virtual environment, you can just type deactivate, after which everything will go back to how it was. Your system's default Python interpreter will be reinstated, along with your globally installed Python libraries.

INSTALL AND TEST DJANGO

Your next step is to install the Django library and its dependencies. You'll specify a particular version here, though it's also possible to leave the version unspecified, in which case pip will just install the latest one.

```
(venv) C:\> python -m pip install django=="3.2.9"
```

The pip machinery takes care of installing all the Django dependencies too, as you'll notice from the list of packages that scrolls past. After the clicking and whirring is done, you should see a success message:

```
Successfully installed asgiref-3.4.1 django-3.2.9 pytz-2021.3 sqlparse-0.4.2
```

The Django library is now installed in your virtual environment. Django, its command-line tools, and its libraries will remain available as long as this environment is active.

You can use the Python interpreter to check that Django has been installed correctly. After invoking python from the command line, you can import Django and check its version interactively:

```
>>>
```

```
>>> import django
>>> django.get_version()
'3.2.9'
>>> exit()
```

If you get a version number, as above, and not an ImportError, then you can be confident that your Django installation is good to go.

It's a good idea now to pin your dependencies. This records the versions of all the Python libraries currently installed in your virtual environment:

```
(venv) C:\> python -m pip freeze > requirements.txt
```

The text file requirements.txt now lists the exact versions of all the packages that you're using, so you or another developer can exactly reproduce your virtual environment at a later time.

STEP 2: CREATE YOUR DJANGO TO-DO APP

First, you need to use Django's tooling to perform a few project-specific steps. These include:

- Generating the parent project framework
- Creating the web app's framework
- Integrating the web app into the project

All Django projects share a similar structure, so understanding this layout will benefit your future projects too. Once you're familiar with this process, you should be able to complete it within a couple of minutes.

To download the code for this stage of the project, click the following link and navigate to the source_code_step_2/ folder:

Scaffold the Parent Project

Django distinguishes between a project and an app. A project can manage one or many apps. Start your new project now:

```
(venv) C:\> django-admin startproject todo_project .
```

Notice the final dot (.) here. It stops django-admin from creating an extra level of folders.

This command has automatically created a file named manage.py, along with a subfolder named todo_project/ containing several Python files:

```
todo_list/
│
├── todo_project/
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
```

```
│   └── wsgi.py
│
├── venv/
│
├── manage.py
└── requirements.txt
```

The contents of the todo_list/venv/ folder are too large to show here.

The file named __init__.py is empty. It exists only to tell Python that its containing folder should be treated as a package. Conceptually, a Python **module** is a single source file, and a **package** is a container for a bunch of modules. By organizing modules inside packages, you can reduce name pollution and improve code isolation.

Two of the files inside todo_list/todo_project/ are important for your app:

- settings.py holds the project-wide configuration. This includes a list of all the apps that the project knows about, as well as a setting that describes which database it'll use.
- urls.py has a list of all the URLs that the server must listen for.

You'll soon need to edit both of these files as you set things up. You'll need to add your app's name to settings.py and also provide a URL as an entry point for browsers wanting to access the application.

Get Started on Your Django To-Do List App

Create your app using the django-admin command-line tool. All you need to supply is a name. You may as well call it todo_app:

```
(venv) C:\> django-admin startapp todo_app
```

This command sets up a new Django app, with a few starter files and folders. If you check your folder, then you'll find that you now have three subfolders inside your project root:

```
todo_list/
│
├── todo_app/
│
├── todo_project/
│
└── venv/
```

It helps to remember that todo_project/ is your **project folder** and contains *project-wide* information. That means general project settings, and information that the web server will need to find the app or apps that your project contains.

On the other hand, your last command just created the todo_app/ folder and its contents. todo_app/ is your **app folder** and contains files *specific to your app*.

```
todo_app/
│
├── migrations/
│   └── __init__.py
│
├── __init__.py
├── admin.py
├── apps.py
├── models.py
├── tests.py
└── views.py
```

The django-admin tool created all these files for you, but you don't need to concern yourself with all of them now. However, there are a few that are definitely worth noticing:

- The two __init__.py files are there, as usual, just to define their containing folders as packages.
- The migrations/ subfolder will hold information about changes to your future database.
- The models.py file will define the data model for your app.
- The views.py file will handle the logic controlling the app display.

Along the way, you'll also create a few more files of your own. These will include a **data model**, new **views**, and new **templates**.

First, though, you need to configure your project so that it knows about your app. The project-level configuration lives in the project directory, in the file todo_list/todo_project/settings.py.

Configure Your Project

You'll notice an array named INSTALLED_APPS with a short list of app names, starting with django.contrib. Django provides these apps and helpfully installs them by default to address common needs. But one all-important app name is missing from the list: yours. You need to add todo_app as an item in the array INSTALLED_APPS:

```
# todo_list/todo_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "todo_app",
]
```

While you have settings.py open, notice that it contains a few other interesting variables. One example of such a variable is DATABASES:

```
# Database
# https://docs.djangoproject.com/en/3.2/ref/settings/#databases

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

DATABASES is set up by default to use the sqlite3 database. This is the easiest option to use, but you can experiment with using other databases later if you're interested.

Two variables that affect the security of your app are SECRET_KEY and DEBUG:

```
- # SECURITY WARNING: keep the secret key used in production secret

SECRET_KEY = (
    "django-insecure-!r4sgi-w($+vmxpe12rg%bvyf$7kz$co3tzw6klpu#f)yfmy#3"
)

# SECURITY WARNING: don't run with debug turned on in production

DEBUG = True
```

These two keys can be left alone during development, but you should know about them for any app that you plan to publish on the Web.

- SECRET_KEY is important if you plan to put your app on a public server. Django generates a new random SECRET_KEY for every new project.
- DEBUG is a very useful setting while you are developing the app, but you should be sure to set it to False before your app goes out on the big bad Web, as it reveals way too much about the workings of your code!

Save the file settings.py and close it.

The second file that you need to modify in the project folder is urls.py, which controls the URL lookup at the project level. Django's **URL dispatcher** uses the elements in the urlpatterns array in this file to decide how to dispatch incoming requests. You're about to add a new urlpattern element to this array, which will cause the URL dispatcher to redirect incoming URL traffic to your new to-do app:

```
1# todo_list/todo_project/urls.py
2from django.contrib import admin
3from django.urls import include, path
4
```

```
5urlpatterns = [
6    path("admin/", admin.site.urls),
7    path("", include("todo_app.urls"))
8  ]
```

The lines that you need to change or add are highlighted above. Check out what's happening in this code:

Line 6 contains the original urlpatterns element, which exists to tell Django that incoming URLs starting with "admin/" should be handled by the admin app.

Line 7 adds an element that can handle an empty string, such as a URL *without* a leading app name. Such a URL will be passed on to the todo_app URL configuration. If there were other apps in the project, then you could distinguish them by using different URL patterns, but that's not necessary here. The URL configuration for todo_app, in turn, is held in a urlpatterns array contained in a file named todo_list/todo_app/urls.py.

You'll create that app-level URL configuration file now. Open a new file in your editor and save it in the todo_list/todo_app directory under the name urls.py:

```
1# todo_list/todo_app/urls.py
2
3urlpatterns = [
4]
```

Leave the app's urlpatterns array empty for now. You'll add some real routes soon.

You now have a complete, working project and app setup, though it doesn't do much yet. The whole infrastructure is in place, but you need to add some content. You can test your work so far by starting the Django development server:

```
(venv) C:\> python manage.py runserver
```

Ignore the warning messages about migrations. The final message displayed on the console should say that your server is running.

Navigate to http://localhost:8000/ in your browser. At this stage, Django has no app pages to display, so you should see the framework's generic success page.

STEP 3: DESIGN YOUR TO-DO DATA

The heart of any application is its data structures. In this step, you'll design and code the application **data model** and the relations between application objects. Then you'll use Django's object-relational modeling tools to map that data model into database tables.

To download the code for this stage of the project, click the following link and navigate to the source_code_step_3/ folder:

Each type of user data will require its own **data model**. Your to-do list app will contain just two basic types of data:

1. **A ToDoList with a title:** You can have as many of these as you want.
2. **A ToDoItem that is linked to a particular list:** Again, there's no limit to the number of ToDoItem objects. Each ToDoItem will have its own title, a longer description, a created date, and a due date.

Your data models will form the backbone of your app. Next up, you'll define them by editing the file models.py.

Define Your Data Models

Open the file models.py in your editor. It's pretty minimal at the moment:

```
# todo_list/todo_app/models.py
from django.db import models


# Create your models here.
```

This is just placeholder text to help you remember where to define data models. Replace this text with the code for your data models:

```
# todo_list/todo_app/models.py
from django.utils import timezone

from django.db import models
from django.urls import reverse

def one_week_hence():
    return timezone.now() + timezone.timedelta(days=7)

class ToDoList(models.Model):
    title = models.CharField(max_length=100, unique=True)

    def get_absolute_url(self):
        return reverse("list", args=[self.id])

    def __str__(self):
        return self.title

class ToDoItem(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField(null=True, blank=True)
    created_date = models.DateTimeField(auto_now_add=True)
    due_date = models.DateTimeField(default=one_week_hence)
    todo_list = models.ForeignKey(ToDoList, on_delete=models.CASCADE)
```

```
25
26  def get_absolute_url(self):
27      return reverse(
28          "item-update", args=[str(self.todo_list.id), str(self.id)]
29      )
30
31  def __str__(self):
32      return f"{self.title}: due {self.due_date}"
33
34  class Meta:
35      ordering = ["due_date"]
```

The file models.py defines your entire data model. In it, you've defined one function and two data model classes:

- **Lines 7 to 8** define a stand-alone utility function, one_week_hence(), that'll be useful for setting ToDoItem default due dates.
- **Lines 10 to 35** define two classes that extend Django's django.db.models.Model superclass. That class does most of the heavy lifting for you. All you need to do in the subclasses is define the data fields in each model, as described below.

The Model superclass also defines an id field, which is automatically unique for each object and serves as its identifier.

The django.db.models submodule also has handy classes for all the field types that you might want to define. These allow you to set up useful default behavior:

- **Lines 11 and 20** declare title fields that are each limited to one hundred characters. In addition, ToDoList.title must be unique. You can't have two ToDoList objects with the same title.
- **Line 21** declares a ToDoItem.description field that may be empty.
- **Lines 22 and 23** each provide useful defaults for their date fields. Django will automatically set .created_date to the current date the first time a ToDoItem object is saved, while .due_date uses one_week_hence() to set a default due date one week in the future. Of course, the application will allow the user to change the due date if this default value doesn't suit them.
- **Line 24** declares what is perhaps the most interesting field, ToDoItem.todo_list. This field is declared as a foreign key. It links the ToDoItem back to its ToDoList, so that each ToDoItem must have exactly one ToDoList to which it belongs. In database lingo, this is a one-to-many relationship. The on_delete keyword in the same line ensures that if a to-do list is deleted, then all the associated to-do items will be deleted too.
- **Lines 16 to 17 and 31 to 32** declare .__str__() methods for each model class. This is the standard Python way of creating a readable representation of an object. It's not strictly necessary to write this function, but it can help with debugging.
- **Lines 13 to 14 and 26 to 29** implement the .get_absolute_url() method, a Django convention for data models. This function returns the URL for the particular data item. This allows you to reference the URL conveniently and robustly in your code. The return statement of both implementations of .get_absolute_url() uses reverse() to avoid hard-coding the URL and its parameters.
- **Line 34** defines the nested Meta class, which allows you to set some useful options. Here, you're using it to set a default ordering for ToDoItem records.

Now save the models.py file with its two model classes. With this file, you've just declared the data model for the entire app.

Create the Database

You've defined the two model classes in your Python code. Use the command line to create and activate the migrations:

```
(venv) C:\> python manage.py makemigrations todo_app
(venv) C:\> python manage.py migrate
```

These two subcommands, make migrations and migrate, are provided by manage.py and help to automate the process of keeping your physical database structure in line with the data model in your code.

With make migrations, you're telling Django that you've changed the application's data model, and you'd like to record those changes. In this particular case, you've defined two brand-new tables, one for each of your data models. In addition, Django has automatically created its own data models for the admin interface and for its own internal use.

Each time you make a change to the data model and call makemigrations, Django adds a file to the folder todo_app/migrations/. The files in this folder store a history of the changes that you've made to the database structure. This allows you to revert and then reapply those changes later if need be.

With the migrate command, you put those changes into effect by running commands against the database. Just like that, you've created your new tables, along with some useful admin tables.

Your data model is now mirrored in the database, and you've also created an audit trail in preparation for any structural changes that you may apply later.

STEP 4: ADD YOUR SAMPLE TO-DO DATA

So now you have a data model and a database, but you don't yet have any actual *data* in the form of to-do lists or items. You can create a little test data in the easiest way, by using Django's ready-made admin interface. This tool enables you not only to manage model data, but also to authenticate users, display and handle forms, and validate input.

To download the code for this stage of the project, click the following link and navigate to the source_code_step_4/ folder:

Meet the Django Admin Interface

To use the admin interface, you should have a **superuser**. This will be someone with extraordinary powers who can be trusted with the keys to the whole Django server. Create your new **superuser** now:

```
(venv) C:\> python manage.py createsuperuser
```

Just follow the prompts to register yourself as a **superuser**. Your superpowers are installed!

Although you now have access to the admin interface, there's one more step to complete before it can make use of your new data models. You need to *register* the models with the admin app, which you can do by editing the file admin.py:

```
# todo_list/todo_app/admin.py


from django.contrib import admin
from todo_app.models import ToDoItem, ToDoList
```

```
admin.site.register(ToDoItem)
admin.site.register(ToDoList)
```

And now you're ready to use the Django administration app. Launch the development server and start exploring. First, make sure the development server is running:

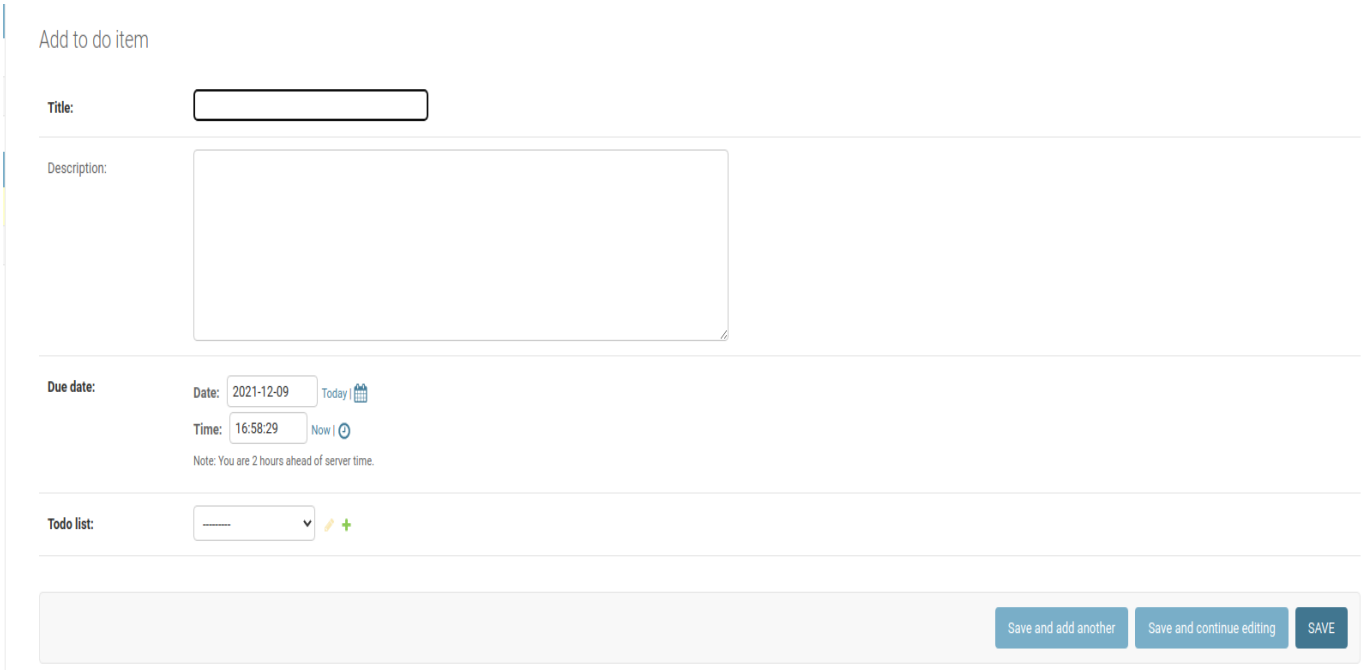
```
(venv) C:\> python manage.py runserver
```

Now open a web browser and go to the address http://127.0.0.1:8000/admin/. You should see the admin app's login screen. Enter your newly minted credentials, and the admin landing page appears:

Start a To-Do List

At the left of the main Django administration page, click on *To do lists*. On the next screen, click on the button at the top right that says *ADD TO DO LIST*.

You're ready to create your first to-do list. Give it a title, for example "Things to do today," and click the *SAVE* button on the extreme right of the screen. The new list now appears on a page headed *Select to do list to change*. You can ignore this and instead click on the + *Add* button that appears next to *To do items* on the left of the screen. A new form appears:



Fill the item with some sample data. Give your item the title "Start my to-do list" and the description "First things first." Leave the due date as it is, exactly one week from today. For *Todo list*, select your newly created to-do list title from the dropdown menu. Then hit *SAVE*.

STEP 5: CREATE THE DJANGO VIEWS

How to create the public interface for your app? In Django, this involves using views and templates. A **view** is the code that orchestrates web pages, the *presentation logic* of your web app. The **template** is the component that more closely resembles an HTML page, as you'll see.

To download the code for this stage of the project, click the following link and navigate to the source_code_step_5/ folder:

Code Your First View

A **view** is Python code that tells Django how to navigate between pages and which data to send along to be displayed. A web application works by receiving an HTTP request from the browser, deciding what to do with it, and then sending back a response. The application then sits back and waits for the next request.

In Django, this request-response cycle is controlled by the view, which, in its most basic form, is a Python function living inside the file views.py. The view function's main input data is an HttpRequest Python object, and its job is to return an HttpResponse Python object.

There are two basic approaches to coding a view. You can just create a function, as described above, or you can use a Python class, whose *methods* will handle the requests. In either case, you need to inform Django that your view is intended for handling a particular type of request.

There are some definite advantages to using a class:

- **Consistency:** Each HTTP request is associated with a command to the server, known as its method or verb. This may be GET, POST, HEAD, or another, according to the response required. Each **verb** has its own matching method name in your class. For example, the method for handling an HTTP **GET** request is named .get().
- **Inheritance:** You can use the power of inheritance by extending existing classes that already do most of what you need the views to do.

You'll be using the **class-based** approach and taking advantage of Django's pre-built generic views for maximum code reuse. Here's where you can benefit from the wisdom of Django's designers. There are many, many different databases and data models out there, but fundamentally they're all based on units of information called **records** and four basic operations that you can do on them:

1. **Create** records
2. **Read** records
3. **Update** records
4. **Delete** records

These activities are commonly known as CRUD, and they're more or less standard across many applications.

For example, a user expects to be able to select a record, edit its fields, and save it, regardless of whether the record represents a to-do item, an inventory item, or anything else. So Django provides developers with **class-based views**, which are pre-built views implemented as classes that already contain most of the code to do these things.

In your editor, open the file todo_app/views.py. Currently, it contains no useful code. Clear the file and create your first view:

```
# todo_list/todo_app/views.py
from django.views.generic import ListView
from .models import ToDoList



class ListListView(ListView):
    model = ToDoList
    template_name = "todo_app/index.html"
```

The ListListView class will display a list of the to-do list titles. As you can see, not much code is required here. You're making use of the generic class django.views.generic.ListView. It already knows how to retrieve a list of objects from the database, so you only need to tell it two things:

1. The data-model **class** that you'd like to fetch
2. The name of the **template** that'll format the list into a displayable form

In this case, the data-model class is ToDoList, which you created in Step 3. Now it's time to learn about **templates**.

Understand Templates

A **template** is just a file containing HTML markup, with a few additional placeholders to accommodate dynamic data. Because you want to be able to reuse your code, you're going to start by creating a **base template** that contains all the boilerplate HTML code that you want to appear on every page. The actual app pages will then **inherit** all this boilerplate, like how your views will inherit much of their functionality from base classes.

Create a Base Template

Create a new folder inside the todo_app/ directory named templates/. Now add a new file to this folder. You can call it base.html. Create your base template:

```
<!-- todo_list/todo_app/templates/base.html -->
<!-- Base template -->
<!doctype html>
<html lang="en">

<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
   <!--Simple.css-->
   <link rel="stylesheet" href="https://cdn.simplecss.org/simple.min.css">
   <title>Django To-do Lists</title>
</head>

<body>
  <div>
    <h1 onclick="location.href='{% url "index" %}'">
       Django To-do Lists
    </h1>
  </div>
  <div>
    {% block content %}
    This content will be replaced by different html code for each page.
    {% endblock %}
  </div>
</body>
```

```
27
28</html>
```

For the most part, this is just a skeleton HTML page with the standard structure. **Lines 22 to 24**, however, declare special {% block content %} … {% endblock %} tags. These placeholders reserve a space that'll receive more HTML markup from the pages that **inherit** from this one.

Line 11 imports the open-source Simple.css library. Raw HTML without CSS renders in a way most people find, well, ugly. Just by importing Simple.css like this, you automatically make your site look much better, with no further effort. Now you can develop and test your site logic without hurting your eyes. Of course, you'll still have the option of adding your own creative touches later!

Line 17 introduces the template syntax {% url "index" %} as the target of an onclick event handler.

The Django template engine will find the urlpatterns entry named "index" in the **URLConf** file todo_app/urls.py and replace this template with the correct path. The effect of this is that a click on the *Django To-do Lists* heading will cause the browser to redirect itself to the URL named "index", which will be your home page.

So base.html is a complete web page, but not very exciting as it stands. To make it more interesting, you need some markup to go between the {% block content %} and {% endblock %} tags. The inheriting templates will each supply their own markup to fill this block.

Add a Home Page Template

Django's convention for the templates belonging to an app is that they live in a folder named templates/<appname> inside the app folder. So although the base template base.html went in the todo_app/templates/ folder, the others will all be placed inside a folder named todo_app/templates/todo_app/:

```
(venv) C:\> mkdir todo_app\templates\todo_app
```

Your first objective will be to code the template for the home page of your website, traditionally named index.html. Create a new file in your editor and save it in the folder that you've just created:

```
1<!-- todo_list/todo_app/templates/todo_app/index.html -->
2
3{% extends "base.html" %}
4{% block content %}
5<!--index.html-->
6{% if object_list %}
7<h3>All my lists</h3>
8{% endif %}
9<ul>
10   {% for todolist in object_list %}
11   <li>
12      <div
13         role="button"
14         onclick="location.href='{% url "list" todolist.id %}'">
```

```
15          {{ todolist.title }}
16      </div>
17  </li>
18  {% empty %}
19  <h4>You have no lists!</h4>
20  {% endfor %}
21</ul>
22
23{% endblock %}
```

Line 3 sets the scene: you're **extending** the base template. That means that everything from the base template will appear in the rendered version of index.html, *except* everything between and including the {% block content %} and {% endblock %} tags. Instead, that section of base.html will be replaced by the code inside the corresponding pair of {% block content %} and {% endblock %} tags in index.html.

There's more template magic to be found inside index.html, where more template tags allow some quite Python-like logic:

- **Lines 6 to 8** define a code block delimited by {% if object_list %} and {% endif %} tags. These ensure that the heading All my lists won't appear if object_list is null or empty. ListView automatically provides the context variable object_list to the template, and object_list contains the list of to-do lists.
- **Lines 10 to 20** define a loop bracketed by the {% for todolist in object_list %} and {% endfor %} tags. This construct renders the enclosed HTML once for each object in the list. As a nice bonus, the {% empty %} tag in **line 18** lets you define what should be rendered instead, if the list is empty.
- **Line 15** demonstrates the mustache syntax. The double curly brackets ({{ … }}) cause the template engine to emit HTML that displays the value of the enclosed variable. In this case, you're rendering the title property of the loop variable todolist.

You've now created the **home page** of your application, which will display a list of all your to-do lists if you have any, or an informative message if not. Your user will be directed to other pages, or back to this one, by the **URL dispatcher**.

Build a Request Handler

The life of a web app is mostly spent waiting for HTTP Requests to arrive from the browser. The two most common HTTP request verbs are **GET** and **POST**. The action performed by a GET request is mostly defined by its URL, which not only routes the request back to the correct server, but also contains **parameters** that tell the server exactly what information the browser is requesting. A typical HTTP URL might look like http://example.com/list/3/item/4.

This hypothetical URL, sent with a GET request, might be interpreted by the server at example.com as a request to use the default app to show item four from list three.

A POST request may also have URL parameters, but it behaves a little differently. POST sends some further information, besides the URL parameters, to the server. This information, which isn't displayed in the browser's search bar, might be used to update a record from a web form, for example.

Django's URL dispatcher has the responsibilty of parsing URLs and forwarding the requests to the appropriate **views**. The URLs may be received from the browser or sometimes internally, from the server itself.

The **URL dispatcher** does its job by consulting what's known as the URLconf, a set of URL patterns mapped to views. These mappings are conventionally stored in a file named urls.py. Once the URL dispatcher finds a match, then it calls the matching view with the URL parameters.

Back in Step 2, you created a urlpatterns item in the *project-level* URLconf file, todo_project/urls.py. That urlpattern sees to it that any HTTP request starting with http://example.com[:port]/... will be passed to your app. The *app-level* urls.py takes it from there.

You've already created the app-level URL file. Now it's time to add the first **route** to that file. Edit the file todo_app/urls.py to add the route:

```
1# todo_list/todo_app/urls.py
2from django.urls import path
3from . import views
4
5urlpatterns = [
6    path("", views.ListListView.as_view(), name="index"),
7]
```

Line 6 tells Django that if the rest of the URL is empty, your ListListView class should be called to handle the request. Notice that the name="index" parameter matches the target of the {% url "index" %} macro that you saw in **line 18** of the base.html template.

So now you have all the ingredients to produce your first home-baked view. The request-response cycle proceeds as follows:

1. When the server receives a **GET** request with this URL from the browser, it creates an HTTPRequest object and sends it to the ListListView that you previously defined in views.py.
2. This particular view is a ListView based on the ToDoList model, so it fetches all the ToDoList records from the database, turns them into ToDoList Python objects, and appends them to a list, named object_list by default.
3. The view then passes the list to the template engine for display, using the specified template, index.html.
4. The template engine builds the HTML code from index.html, automatically combining it with base.html and using the passed-in data plus the template's embedded logic to populate the HTML elements.
5. The view constructs an HttpResponse object containing the fully built HTML and returns this to Django.
6. Django turns the HttpResponse into an HTTP message and sends that back to the browser.
7. The browser, seeing a fully formed HTTP page, displays it to the user.

You've just created first end-to-end Django request handler. You may have noticed that your index.html file references a URL name that doesn't exist yet: "list". That means that if you try to run your app now, it won't work. You'll need to define that URL and create the corresponding view to make the app work properly.

This new view, like ListListView and all the other views will be **class-based**.

Reuse Class-Based Generic Views

A class that's intended to work as a view should extend the class django.views.View and override that class's methods, such as .get() and .post(), that handle the corresponding HttpRequest types. Each of these methods accepts an HttpRequest and returns an HttpResponse.

Class-based generic views take reusability to the next level. Most of the expected functionality is already encoded in the base class. A view class based on the generic view class ListView, for example, needs to know just two things:

1. What data type it's listing
2. What template it'll use to render the HTML

With that information, it can render a list of objects. Of course, ListView, like other generic views, doesn't limit you to this very basic pattern. You can tweak and subclass the base class to your heart's content. But the basic functionality is already there, and it costs you nothing.

Subclass ListView to Display a List of To-Do Items

Now you can do something very similar to display a list of to-do *items*. You'll start by creating another view class, this time called ItemListView. Like the class ListListView, ItemListView will extend the generic Django class ListView. Open views.py and add your new class:

```
# todo_list/todo_app/views.py
from django.views.generic import ListView
from .models import ToDoList, ToDoItem

class ListListView(ListView):
    model = ToDoList
    template_name = "todo_app/index.html"

class ItemListView(ListView):
    model = ToDoItem
    template_name = "todo_app/todo_list.html"

    def get_queryset(self):
        return ToDoItem.objects.filter(todo_list_id=self.kwargs["list_id"])

    def get_context_data(self):
        context = super().get_context_data()
        context["todo_list"] = ToDoList.objects.get(id=self.kwargs["list_id"])
        return context
```

In your ItemListView implementation, you're specializing ListView a little bit. When you show a list of ToDoItem objects, you don't want to show *every* ToDoItem in the database, just those that belong to the

current list. To do this, **lines 13 to 14** override the ListView.get_queryset() method by using the model's objects.filter() method to restrict the data returned.

Every descendant of the View class also has a .get_context_data() method. The return value from this is the template's context, a Python dictionary that determines what data is available for rendering. The result of .get_queryset() is automatically included in context under the key object_list, but you'd like the template to be able to access the todo_list object itself, and not just the items within it that were returned by the query.

Lines 16 to 19 override .get_context_data() to add this reference to the context dictionary. It's important that **line 17** calls the superclass's .get_context_data() first, so that the new data can be merged with the existing context instead of clobbering it.

Notice that both of the overridden methods make use of self.kwargs["list_id"]. This implies that there must be a keyword argument named list_id passed to the class when it's constructed. You'll soon learn where this argument comes from.

Show the Items in a To-Do List

Your next task is to create a template for displaying the TodoItems in a given list. Once again, the {% for <item> in <list> %} … {% endfor %} construct will be indispensable. Create the new template, todo_list.html:

```
1<!-- todo_list/todo_app/templates/todo_app/todo_list.html -->
2{% extends "base.html" %}
3
4{% block content %}
5<div>
6  <div>
7    <div>
8      <h3>Edit list:</h3>
9      <h5>{{ todo_list.title | upper }}</h5>
10    </div>
11    <ul>
12      {% for todo in object_list %}
13      <li>
14        <div>
15          <div
16            role="button"
17            onclick="location.href='#'">
18            {{ todo.title }}
19            (Due {{ todo.due_date | date:"l, F j" }})
20          </div>
21        </div>
22      </li>
23      {% empty %}
```

```
24          <p>There are no to-do items in this list.</p>
25          {% endfor %}
26      </ul>
27      <p>
28          <input
29              value="Add a new item"
30              type="button"
31              onclick="location.href='#'" />
32      </p>
33  </div>
34</div>
35{% endblock %}
```

This template for displaying a single list with its to-do items is similar to index.html, with a couple of extra wrinkles:

- **Lines 9 and 19** exhibit a curious syntax. These expressions with a pipe symbol (|) are called template filters, and they provide a convenient way of formatting the title and the due date, respectively, using the pattern to the right of the pipe.
- **Lines 15 to 17 and 28 to 31** define a couple of button-like elements. Right now, their onclick event handlers do nothing useful, but you'll soon be fixing that.

So you've coded an ItemListView class, but so far, there's no way for your user to invoke it. You need to add a new route into urls.py so that ItemListView can be used:

```
1# todo_list/todo_app/urls.py
2from todo_app import views
3
4urlpatterns = [
5   path("",
6       views.ListListView.as_view(), name="index"),
7   path("list/<int:list_id>/",
8       views.ItemListView.as_view(), name="list"),
9]
```

Line 7 declares a placeholder as the new route's first parameter. This placeholder will match a positional parameter in the URL path that the browser returns. The syntax list/<int:list_id>/ means that this entry will match a URL like list/3/ and pass the named parameter list_id = 3 to the ItemListView instance. If you revisit the ItemListView code in views.py, you'll notice that it references this parameter in the form self.kwargs["list_id"].

Now you can view all of your lists, thanks to the route, view, and template that you've created. You've also created a route, view, and template for listing individual to-do items.

It's time to test what you've done so far. Try running your development server now, in the usual way:

```
(venv) C:\> python manage.py runserver
```

Depending on what you added from the admin interface, you should see one or more list names. You can click on each one to show the items that particular list contains. You can click on the main *Django To-do Lists* heading to navigate back to the app's main page.

Your app can now display lists and items. You've implemented just the **Read** part of the **CRUD** operations. Run your development server now. You should be able to navigate back and forth between the list of to-do lists and the items in a single list, but you cannot yet add or delete lists, or add, edit, and remove items.

STEP 6: CREATE AND UPDATE MODEL OBJECTS IN DJANGO

In this step, you'll enhance your app by enabling **Creation** and **Update** of lists and items. You'll do this by extending some more of Django's generic view classes. Through this process, you'll notice how the logic already baked into these classes accounts for many of the typical **CRUD** use cases. But remember that you're not limited to the pre-baked logic. You can override almost any part of the request-response cycle.

To download the code for this stage of the project, click the following link and navigate to the source_code_step_6/ folder:

The first item on the agenda will be to add the new views that support the Create and Update actions. Next, you'll add URLs referencing those views, and finally, you'll update the todo_items.html template to provide links allowing the user to navigate to the new URLs.

Add your new imports and view classes to views.py:

```
1# todo_list/todo_app/views.py
2from django.urls import reverse
3
4from django.views.generic import (
5    ListView,
6    CreateView,
7    UpdateView,
8)
9from .models import ToDoItem, ToDoList
10
11class ListListView(ListView):
12    model = ToDoList
13    template_name = "todo_app/index.html"
14
15class ItemListView(ListView):
16    model = ToDoItem
17    template_name = "todo_app/todo_list.html"
18
19    def get_queryset(self):
20        return ToDoItem.objects.filter(todo_list_id=self.kwargs["list_id"])
21
```

```
22    def get_context_data(self):
23        context = super().get_context_data()
24        context["todo_list"] = ToDoList.objects.get(id=self.kwargs["list_id"])
25        return context
26
27class ListCreate(CreateView):
28    model = ToDoList
29    fields = ["title"]
30
31    def get_context_data(self):
32        context = super(ListCreate, self).get_context_data()
33        context["title"] = "Add a new list"
34        return context
35
36class ItemCreate(CreateView):
37    model = ToDoItem
38    fields = [
39        "todo_list",
40        "title",
41        "description",
42        "due_date",
43    ]
44
45    def get_initial(self):
46        initial_data = super(ItemCreate, self).get_initial()
47        todo_list = ToDoList.objects.get(id=self.kwargs["list_id"])
48        initial_data["todo_list"] = todo_list
49        return initial_data
50
51    def get_context_data(self):
52        context = super(ItemCreate, self).get_context_data()
53        todo_list = ToDoList.objects.get(id=self.kwargs["list_id"])
54        context["todo_list"] = todo_list
55        context["title"] = "Create a new item"
56        return context
57
58    def get_success_url(self):
59        return reverse("list", args=[self.object.todo_list_id])
60
61class ItemUpdate(UpdateView):
62    model = ToDoItem
63    fields = [
```

```
64      "todo_list",
65      "title",
66      "description",
67      "due_date",
68  ]
69
70  def get_context_data(self):
71      context = super(ItemUpdate, self).get_context_data()
72      context["todo_list"] = self.object.todo_list
73      context["title"] = "Edit item"
74      return context
75
76  def get_success_url(self):
77      return reverse("list", args=[self.object.todo_list_id])
```

There are three new view classes here, all derived from Django's generic view classes.

Two of the new classes extend django.view.generic.CreateView, while the third extends django.view.generic.UpdateView:

- **Lines 27 to 34** define ListCreate. This class defines a form containing the sole public ToDoList attribute, its title. The form itself also has a title, which is passed in the context data.
- **Lines 36 to 59** define the ItemCreate class. This generates a form with four fields. The .get_initial() and .get_context_data() methods are overridden to provide useful information to the template. The .get_success_url() method provides the view with a page to display after the new item has been created. In this case, it calls the list view after a successful form submit to display the full to-do list containing the new item.
- **Lines 61 to 77** define ItemUpdate, which is very similar to ItemCreate but supplies a more appropriate title.

You've now defined three new view classes for creating and updating to-do lists and their items. Your code and templates will instantiate these classes on demand, complete with the relevant list or item data.

Lists and Items

ListCreate and ItemCreate both extend the class CreateView. This is a generic view that can be used with any Model subclass. The Django documentation describes CreateView as follows:

So CreateView can be a base class for any view designed to create objects.

ItemUpdate will extend the generic view class UpdateView. This is quite similar to CreateView, and you can use the same template for both. The main difference is that the ItemUpdate view will pre-populate the template form with the data from an existing ToDoItem.

The generic views know how to handle the **POST** request generated by the form on a successful submit action.

As always, the child classes need to be told which Model they're based on. These models will be ToDoList and ToDoItem, respectively. The views also have a fields property that you can use to restrict which of the Model data fields are displayed to the user. For example, the ToDoItem.created_date field is

completed automatically in the data model, and you probably don't want the user to change it, so you can omit it from the fields array.

Now you need to define routes, so that the user can reach each of the new views with the appropriate data values set. Add the routes as new items in the urlpatterns array, naming them "list-add", "item-add", and "item-update":

```
# todo_list/todo_app/urls.py
from django.urls import path
from todo_app import views

urlpatterns = [
    path("", views.ListListView.as_view(), name="index"),
    path("list/<int:list_id>/", views.ItemListView.as_view(), name="list"),
    # CRUD patterns for ToDoLists
    path("list/add/", views.ListCreate.as_view(), name="list-add"),
    # CRUD patterns for ToDoItems
    path(
        "list/<int:list_id>/item/add/",
        views.ItemCreate.as_view(),
        name="item-add",
    ),
    path(
        "list/<int:list_id>/item/<int:pk>/",
        views.ItemUpdate.as_view(),
        name="item-update",
    ),
]
```

Now you've associated names, URL patterns, and views with the three new routes, each of which corresponds to an action on the data.

Notice that the "item-add" and "item-update" URL patterns contain parameters, just like the "list" path. To create a new item, your view code needs to know the list_id of its parent list. To update an item, both its list_id and the item's own ID, which is here called pk, must be known to the view.

New Views

Next, you'll need to provide some links in your templates to activate the new views. Just before the {% endblock %} tag in index.html, add a button:

```
<p>
  <input
    value="Add a new list"
    type="button"
    onclick="location.href='{% url "list-add" %}'"/>
</p>
```

A click on this button will now generate a request with the "list-add" pattern. If you look back at the corresponding urlpattern item in todo_app/urls.py, then you'll see that the associated URL looks like "list/add/", and it causes the **URL dispatcher** to instantiate a ListCreate view.

Now you'll update the two dummy onclick events in todo_list.html:

```
<!-- todo_list/todo_app/templates/todo_app/todo_list.html -->
{% extends "base.html" %}

{% block content %}
<div>
  <div>
    <div>
      <h3>Edit list:</h3>
      <h5>{{ todo_list.title | upper }}</h5>
    </div>
    <ul>
      {% for todo in object_list %}
      <li>
        <div>
          <div
            role="button"
            onclick="location.href=
            '{% url "item-update" todo_list.id todo.id %}'">
            {{ todo.title }}
            (Due {{ todo.due_date | date:"l, F j"}})
          </div>
        </div>
      </li>
      {% empty %}
      <p>There are no to-do items in this list.</p>
      {% endfor %}
    </ul>
    <p>
      <input
        value="Add a new item"
        type="button"
        onclick="location.href='{% url "item-add" todo_list.id %}'"
      />
    </p>
  </div>
</div>
{% endblock %}
```

The onclick event handlers now invoke the new URLs named "item-update" and "item-add". Notice again the syntax {% url "key" [param1 [, param2 [,...]]]%} in **lines 18 and 32**, where the urlpattern name is combined with data from context to construct hyperlinks.

For example, in **lines 15 to 21**, you're setting up a button-like div element with an onclick event handler.

Notice that the "item-update" URL requires IDs for both the list and the item to be updated, whereas "item-add" required only todo_list.id.

You'll need templates to render your new ListCreate, ItemCreate, and ItemUpdate views. The first one that you'll tackle is the form for creating a new list. Create a new template file named todolist_form.html:

```
1<!-- todo_list/todo_app/templates/todo_app/todolist_form.html -->
2{% extends "base.html" %}
3
4{% block content %}
5
6<h3>{{ title }}</h3>
7<div>
8   <div>
9      <form method="post">
10         {% csrf_token %}
11         {{ form.as_p }}
12         <input
13            value="Save"
14            type="submit">
15         <input
16            value="Cancel"
17            type="button"
18            onclick="location.href='{% url "index" %}';">
19      </form>
20   </div>
21</div>
22
23{% endblock %}
```

This page contains a <form> element in **lines 9 to 19** that'll generate a **POST** request when the user submits it, with the user-updated form contents as part of its payload. In this case, the form contains only the list title.

- **Line 10** uses the {% csrf_token %} macro, which generates a Cross-Site Request Forgery token, a necessary precaution for modern web forms.
- **Line 11** uses the {{ form.as_p }} tag to invoke the view class's .as_p() method. This auto-generates the form contents from the fields attribute and the model structure. The form will be rendered as HTML inside a <p> tag.

Next, you'll create another form that'll allow the user to create a new ToDoItem, or edit the details of an existing one. Add the new template todoitem_form.html:

```
1<!-- todo_list/todo_app/templates/todo_app/todoitem_form.html -->
2{% extends "base.html" %}
3
4{% block content %}
5
6<h3>{{ title }}</h3>
7<form method="post">
8   {% csrf_token %}
9   <table>
10      {{ form.as_table }}
11   </table>
12   <input
13      value="Submit"
14      type="submit">
15   <input
16      value="Cancel"
17      type="button"
18      onclick="location.href='{% url "list" todo_list.id %}'">
19</form>
20
21{% endblock %}
```

This time, you're rendering the form as a table (**lines 9 to 11**), because there are several fields per item. Both CreateView and UpdateView contain a .form member with convenient methods like form.as_p() and form.as_table() to perform an automatic layout. The *Submit* button will generate a **POST** request using the form's contents. The *Cancel* button will redirect the user to the "list" URL, passing along the current list id as a parameter.

Run your development server again to verify that you can now create new lists and add items to those lists.

STEP 7: DELETE TO-DO LISTS AND ITEMS

You've written code to create and update both to-do lists and to-do items. But no **CRUD** application is complete without the **Delete** functionality. In this step, you'll add links to the forms to allow the user to delete one item at a time, or even an entire list. Django provides generic views that handle these cases too.

To download the code for this stage of the project, click the following link and navigate to the source_code_step_7/ folder.

Make DeleteView Subclasses

You'll start by adding view classes that extend django.views.generic.DeleteView. Open views.py and make sure that you have all the necessary imports:

```
# todo_list/todo_app/views.py
from django.urls import reverse, reverse_lazy
```

```
from django.views.generic import (
    ListView,
    CreateView,
    UpdateView,
    DeleteView,
)
```

Also, add the two new view classes that support deleting objects. You'll need one for lists and one for items:

```
# todo_list/todo_app/views.py
class ListDelete(DeleteView):
    model = ToDoList
    # You have to use reverse_lazy() instead of reverse(),
    # as the urls are not loaded when the file is imported.
    success_url = reverse_lazy("index")

class ItemDelete(DeleteView):
    model = ToDoItem

    def get_success_url(self):
        return reverse_lazy("list", args=[self.kwargs["list_id"]])

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["todo_list"] = self.object.todo_list
        return context
```

Both of your new classes extend django.views.generic.edit.DeleteView. The official Django documentation describes DeleteView like this:

Define Deletion Confirmations and URLS

Because you'll be offering the user the *Delete* option from the editing pages, you only need to create new templates for the corresponding confirmation pages. There's even a default name for these confirmation templates: <modelname>_confirm_delete.html. If such a template exists, your classes derived from DeleteView will automatically render it when the associated form is submitted.

Create a new template in the file todolist_confirm_delete.html:

```
<!-- todo_list/todo_app/templates/todo_app/todolist_confirm_delete.html -->
{% extends "base.html" %}

{% block content %}
<!--todolist_confirm_delete.html-->

<h3>Delete List</h3>
```

```
<p>Are you sure you want to delete the list <i>{{ object.title }}</i>?</p>

<form method="POST">
    {% csrf_token %}
    <input
        value="Yes, delete."
        type="submit">
</form>
<input
    value="Cancel"
    type="button"
    onclick="location.href='{% url "index" %}';">

{% endblock %}
```

The DeleteView subclass will still be in control when this template is rendered. By clicking the *Yes, delete.* button, you submit the form, and the class goes ahead and deletes the list from the database. If you click *Cancel*, it does nothing. In either case, Django then redirects you to the home page.

That's it for deleting ToDoList objects. Now you can do the same for the deletion of ToDoItem objects. Create another new template, named todoitem_confirm_delete.html:

```
<!-- todo_list/todo_app/templates/todo_app/todoitem_confirm_delete.html -->
{% extends "base.html" %}

{% block content %}

<h3>Delete To-do Item</h3>

<p>Are you sure you want to delete the item: <b>{{ object.title }}</b>
  from the list <i>{{ todo_list.title }}</i>?</p>

<form method="POST">
    {% csrf_token %}
    <input
        value="Yes, delete."
        type="submit">
    <input
        value="Cancel"
        type="button"
        onclick="location.href='{% url "list" todo_list.id %}';">
</form>
```

```
{% endblock %}
```

Exactly the same logic applies, though this time if the *Cancel* button is pressed, the user will be redirected to the "list" URL to display the parent list, rather than to the app's index page.

Now you need to define routes for the deletion URLs. You can do that by adding the highlighted lines to the application's urls.py:

```
# todo_list/todo_app/urls.py
from django.urls import path
from todo_app import views

urlpatterns = [
    path("", views.ListListView.as_view(), name="index"),
    path("list/<int:list_id>/", views.ItemListView.as_view(), name="list"),
    # CRUD patterns for ToDoLists
    path("list/add/", views.ListCreate.as_view(), name="list-add"),
    path(
        "list/<int:pk>/delete/", views.ListDelete.as_view(), name="list-delete"
    ),
    # CRUD patterns for ToDoItems
    path(
        "list/<int:list_id>/item/add/",
        views.ItemCreate.as_view(),
        name="item-add",
    ),
    path(
        "list/<int:list_id>/item/<int:pk>/",
        views.ItemUpdate.as_view(),
        name="item-update",
    ),
    path(
        "list/<int:list_id>/item/<int:pk>/delete/",
        views.ItemDelete.as_view(),
        name="item-delete",
    ),
]
```

These new URLs will load the DeleteView subclasses as views. There's no need to define special URLs for delete confirmation because Django handles that requirement by default, rendering the confirmation page with the <modelname>_confirm_delete templates that you've just added.

Enable Deletions

So far, you've created views and URLS to delete things, but there's no mechanism for your user to invoke this functionality. To fix that, you'll start by adding a button to todoitem_form.html to allow the user to delete the current item. Open todoitem_form.html and add the highlighted lines:

```
<!-- todo_list/todo_app/templates/todo_app/todoitem_form.html -->
{% extends "base.html" %}

{% block content %}

<h3>{{ title }}</h3>
<form method="post">
  {% csrf_token %}
  <table>
    {{ form.as_table }}
  </table>
  <input
    value="Submit"
    type="submit">
  <input
    value="Cancel"
    type="button"
    onclick="location.href='{% url "list" todo_list.id %}'">
  {% if object %}
    <input
      value="Delete this item"
      type="button"
      onclick="location.href=
      '{% url "item-delete" todo_list.id object.id %}'">
  {% endif %}
</form>

{% endblock %}
```

Recall that this view is used for both creating items and updating them. In the case of creation, there will be no item instance loaded in the form. So to avoid confusing the user, you need to wrap the new input element in the conditional template block {% if object %} … {% endif %} so that the *Delete this item* option only appears if the item already exists.

Now you need to add the user interface element for deleting an entire list. Add the highlighted lines to todolist.html:

```
<!-- todo_list/todo_app/templates/todo_app/todo_list.html -->
{% extends "base.html" %}
```

```
{% block content %}
<div>
  <div>
    <div>
      <h3>Edit list:</h3>
      <h5>{{ todo_list.title | upper }}</h5>
    </div>
    <ul>
      {% for todo in object_list %}
      <li>
        <div>
          <div
            role="button"
            onclick="location.href=
            '{% url "item-update" todo_list.id todo.id %}'">
            {{ todo.title }}
            (Due {{ todo.due_date | date:"l, F j" }})
          </div>
        </div>
      </li>
      {% empty %}
      <p>There are no to-do items in this list.</p>
      {% endfor %}
    </ul>
    <p>
      <input
        value="Add a new item"
        type="button"
        onclick="location.href=
        '{% url "item-add" todo_list.id %}'" />
      <input
        value="Delete this list"
        type="button"
        onclick="location.href=
        '{% url "list-delete" todo_list.id %}'" />
    </p>
  </div>
</div>
{% endblock %}
```

In this case, there's always a ToDoList instance associated with the template, so it always makes sense to offer the *Delete this list* option.

STEP 8: USE YOUR DJANGO TO-DO LIST APP

Your project code is now complete. You can download the complete code for this project by clicking the following link and navigating to the source_code_step_final folder:

You've built the entire to-do list application. That means you're ready to put the whole app through its paces!

One more time, fire up your development server. If the console displays errors, then you'll have to resolve them before continuing. Otherwise, use your browser to navigate to http://localhost:8000/

The app heading *Django To-do Lists* will appear on every page. It serves as a link back to the home page, allowing the user to return there from anywhere in the application.

There may be some data already in the app, depending on your previous testing. Now you can start exercising the app logic.

- Click on *Add a new list*. A new screen appears, offering a blank text box for the new list's title.
- Give your new list a name and press *Save*. You're taken to the *Edit List* page, with the message *There are no to-do items in this list.*
- Click on *Add a new item*. The *Create a new item* form appears. Fill in a title and a description, and notice that the default due date is exactly one week ahead.

INDIVIDUAL TASK (choose one option) :

1. CREATE YOUR OWN BLOG.

Idea: A blog site is a great project for your portfolio. Think over the design of the site so that it corresponds to a specific topic (you choose). You can also add admin functionality and the ability to leave comments.

2. PINTEREST CLONE ON DJANGO

Idea: develop a beautiful application in which users will be able to demonstrate their talents by posting photos, paintings, etc. Users should be able to like posts and subscribe to authors in order to see their latest work in the feed.

3. APPLICATION FOR NOTES

Idea: Here you will need to develop an application with a friendly interface for creating notes. You need to think over the functionality for adding, editing and deleting notes. You can also provide for the insertion of images.

4. EXPLANATORY DICTIONARY

Idea: while doing such a project, you will use various APIs from the Internet. The application should provide an interpretation of the words, as well as their antonyms and synonyms.