Django 3…2…1… Takeoff!

Quick Guide to Learning Django 3 Web Development

by Bryam Loaiza

Django 3…2…1…Takeoff! : Quick Guide to Learning Django 3 Web Development 1st Edition.

Production Editor: Andrew Towers
Cover Illustration: Luther Willington
Cover and Interior Design: Carter Manger
Developmental Editor: Mark Conture
Technical Reviewer: Edward Peaker
Copyeditor: Annier Lukith
Proofreader: Akshay Sudhakaran
For information on distribution, translations, or bulk sales, please contact copyright owner directly:
Bryam Loaiza
New Jersey, NJ
baltlogs@gmail.com
The Library of Congress has catalogued the first edition as follows:
Loaiza, Bryam, 2020
Django 3…2…1…Takeoff! : Quick Guide to Learning Django 3 Web Development/ by Bryam Loaiza.
pages cm
Summary: " The focus of this book is to learn Django web development by example. The book has 20 chapters which cover many topics such as URLs, views, templates, AWS (Amazon Web Services), Heroku deployment, and more. By the end of the book, you will have a real web application using Django. There are plenty of books out there, but this book focuses on learning with clear and concise code plus all the chapters are accompanied by images of the web app being built. Great book for beginners who want to learn the basics of Django and start building along the way."-- Provided by author.
01177196
19089417

Table of Contents

Chapter 1 : Django Web App Setup **5**
Visual Studio and Packages Installation 6
Listings App 10
Database Setup 12
Static and Media Files Setup 14
Linter Setup (Optional) 15
Django Simple Web App 17
Chapter 2: Django Basics **20**
Django MVC Pattern 20
Django Admin Site 21
Listings Model 23
Chapter 3: Homepage **27**
Chapter 4: Listings Page **30**
Database Queries 32
Chapter 5: New Listing Page **35**
Chapter 6: Navbar **41**
Chapter 7: Detail Listing Page **45**
Chapter 8: My Listings Page **50**
Chapter 9: Edit Listing Page **53**
Chapter 10: Delete Listing Page **58**
Chapter 11: Users App **61**
Login Page 63
Chapter 12: Register Page **69**
Chapter 13: Foreign Key **73**
Chapter 14: Images **81**
Chapter 15: Django Filters **87**
Chapter 16: Styling with Bootstrap (Part I) **92**

Chapter 17: Styling with Bootstrap (Part II)..103
Django Crispy Forms..103
Chapter 18: AWS - Amazon Relational Database Service (RDS) 111
Amazon RDS Creation 113
Connecting to Amazon RDS..118
Chapter 19: Amazon S3 Buckets 123
Amazon S3 Bucket Creation..124
Chapter 20: Heroku Deployment..133

Chapter 1 : Django Web App Setup

Prerequisites:

- Any computer.
- Source-code editor (Visual Studio Code used during this book; Sublime Text is another option)
- Python 3
- Database (PostgreSQL is used during this book; however, any other database works just fine. In fact, Django already uses SQLite3 database, so no need to make changes if you want to use that database.)

Note:

If you would like to have access to the source code for this project, please send a photo of your receipt or any confirmation of having bought this book to baltslogs@gmail.com

No need for personal information in the photo, just confirmation that you have purchased the book. Once you email us, you'll receive a folder with the source code and all the necessary files that accompany this project.

The action in the note above is optional, you can totally complete this project without the source code just by reading this book. The source code is just an extra resource that comes with the book which might be helpful for some people.

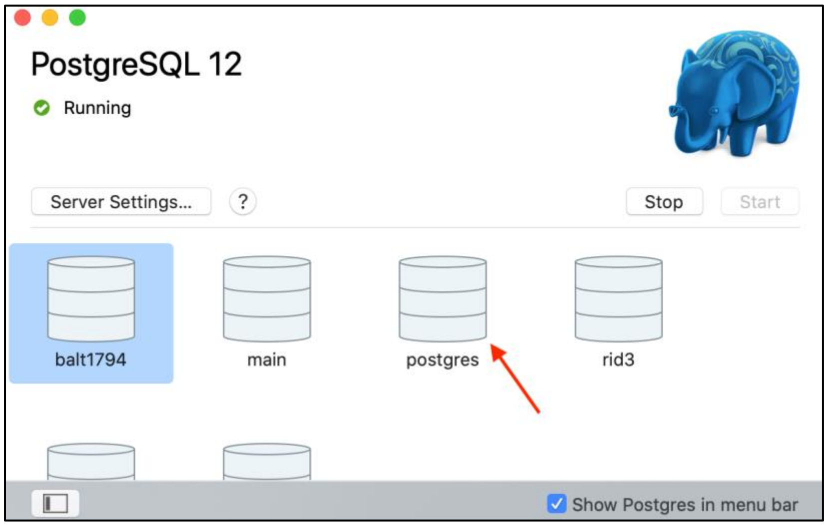
PostgreSQL Database Installation

If you plan to use PostgreSQL database, refer to the steps below.

1. Download and install PostgreSQL database: https://www.postgresql.org/download/

2. Download and install PgAdmin4(We'll use this one towards the end of the book): https://www.pgadmin.org/download/

3. Open PostgreSQL database and click on the *postgres* database (red arrow).



4. The command window will appear after you complete the previous step, issue the following commands.

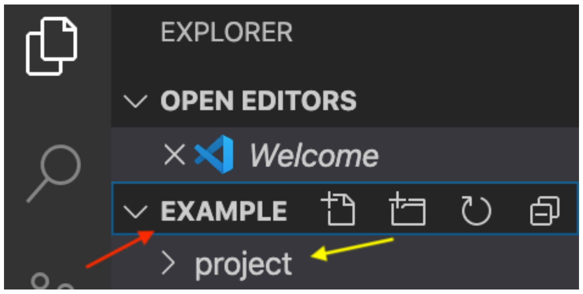
```
postgres=#
postgres=# \password
Enter new password: ******
Enter it again: ******
postgres=# CREATE DATABASE example OWNER postgres;
```

Command	Explanation
\password	- Creates database password
CREATE DATABASE *dbname* OWNER *rolename*;	- Creates a new database - For this project dbname = example and rolename = postgres

Visual Studio and Packages Installation

Start by creating a folder where you would store all your project files. I created my folder in the *Documents* folder of my computer. You can create it anywhere on your computer.

Go to Visual Studio Code and open the folder created on the previous step by clicking on *File > Open*. Inside that folder create another folder with the name of your project.



Open the terminal window by going to *Terminal* > *New Terminal*. A new terminal will appear at the bottom of VS code. Make sure you are inside the first folder you created (*'EXAMPLE'*) by looking at the path in the command line. After you have checked this, go into the folder for your project (*'project'*) by issuing the *cd* command as shown below.

```
Bryans-MacBook-Pro-2:Example balt1794$ cd project
Bryans-MacBook-Pro-2:project balt1794$
```

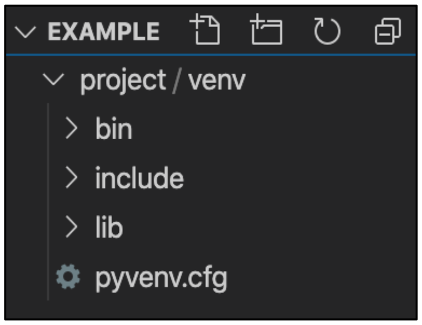
Virtual Environment

A virtual environment (venv) helps us maintain our individual projects relatively contained in order to avoid installing Python packages globally on our computer which might cause problems between conflicting files and packages.

We need to create a virtual environment and activate it. In order to do this, we issue the following commands.

```
Bryans-MacBook-Pro-2:project balt1794$ python3 -m venv venv
Bryans-MacBook-Pro-2:project balt1794$ source venv/bin/activate
(venv) Bryans-MacBook-Pro-2:project balt1794$
```

After these commands have been issued, check that your folder tree looks like the screenshot below.



Now that we are confident that the virtual environment (venv) has been created successfully, let’s start installing the packages needed for our project. There are multiple ways to install packages, we will use the pip install command in this book.

Pip

Pip is the standard package-management system for Python. Pip handles the installation of packages from the command-line interface.

Let’s issue the following commands in the terminal window.

```
(venv) Bryans-MacBook-Pro-2:project balt1794$ pip install Django
(venv) Bryans-MacBook-Pro-2:project balt1794$ pip install Pillow
(venv) Bryans-MacBook-Pro-2:project balt1794$ pip install psycopg2
(venv) Bryans-MacBook-Pro-2:project balt1794$ pip install psycopg2-binary
```

Django

Django is a Python web framework used to develop web applications in a fast and secure way. Django is free and open source. It allows for rapid prototyping while maintaining a secure and reliable structure. It’s also good for scalability.

Pillow

Pillow is a Python library that allows us to manipulate, upload, and save different image file formats. This library adds the basic image processing capabilities for any website to handle image files.

Psycopg2 & Psycopg2-binary

These libraries help us connect to the PostgreSQL database. Psycopg2-binary is ideal for development and testing, if you don’t want to move this project to Heroku by the end of the book, you can install this library only. I will move this project to Heroku, so I will also need the psycopg2 library.

If you’re using the SQLite3 database, then you don’t need to install these packages.

Troubleshooting Psycopg2 Installation

If after issuing *pip install psycopg2* you get an error like the one below, please follow the next steps to troubleshoot.

```
failed with error code 1 in /private/var/folders/fz/lf7pz3kj0pj_c0dxvw9lk40h0000gn/T/pip-
install 3sa8h9hx/psycopg2/
```

Issue the *pg_config* command and copy the LDFLAGS path (copy everything after the equal sign) and then paste it between quotes on the following command as shown below.

```
(venv) Bryans-MacBook-Pro-2:project balt1794$ pg_config
LDFLAGS = -L/usr/local…
(venv) Bryans-MacBook-Pro-2:project balt1794$ env LDFLAGS="L/usr/lo…" pip install psycopg2
```

I have shortened my LDFLAGS path since it is very long and it would be different for each person, but make sure you copy the whole path. Also, don’t forget to add *pip install psycopg2* after the path. Check for extra spaces. Sometimes an extra space can mess up the entire installation.

The problem with psycopg2 arises because of the path to where PostgreSQL was installed. I experienced this problem using a Mac, I’m not sure if it is the case with Windows too.

Note:

This might be a good time to point out that even though I would like to help you tackle all your debugging problems, that is not possible because everyone will encounter different errors throughout the book.

If you got to this point without having much debugging to do, consider yourself lucky, but in the world of programming, debugging is unavoidable, so it is better if you start getting used to it. The best tool to debug is the official Django documentation. If you don’t find your solution there, start by searching your questions on sites like Stack Overflow or Google.

Keep in mind, that there is not just one way to do things, usually, there are multiple ways to solve a problem.

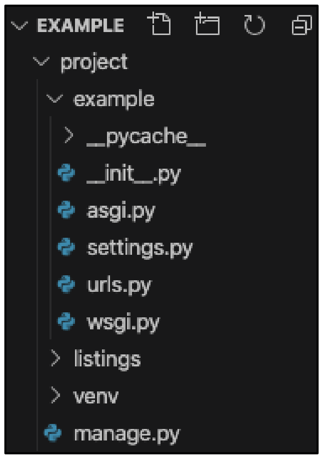
Command	Explanation
python3 -m venv *name*	- Creates a virtual environment; venv is included with Python3, but you can also install virtualenv if preferred. - name = virtual environment name (venv used in this book).
source *name*/bin/activate	- Activates the virtual environment. - name = virtual environment name (venv used in this book). - After issuing this command, you should see the name of your virtual environment on the left of your command line. *(venv)* Bryans-MacBook-Pro-2:project balt1794$.
pip install *name*	- Used to install Python libraries and packages. - name = library or package name.
pg_config	- Retrieves relevant path information about the installed version of PostgreSQL.

Listings App

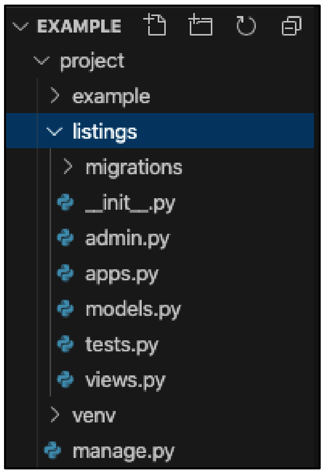
In order to create our Django web app, we need to implement a few things. Let's start by issuing the following commands.

```
(venv) Bryans-MacBook-Pro-2:project balt1794$ django-admin startproject example .
(venv) Bryans-MacBook-Pro-2:project balt1794$ python manage.py startapp listings
```

The first command creates a folder with the main Python files needed for our project. I chose the same name as the very first folder that we created (*'example'*). Don't forget the dot (.) at the end of the command. The dot (.) creates the folder inside our current working directory (*'project'*).



The second command creates an app which contains models.py and other Python files also needed. You can think of apps like small moving parts that we put together to create an entire machine, in this case, a website. In this project, we will have two apps; one for users which will register on our website and the other for listings posted by these users.



Let's start modifying some settings before we run our project. Go to *settings.py* of your project's folder and make the following changes.

```
example/settings.py
…
INSTALLED_APPS = [
   'django.contrib.admin',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
   'listings'
]
…
```

Every time that you create a new app or install some particular packages which are treated like apps, you have to register these apps in *settings.py*. Some examples are django-rest-framework, django-allauth, and others.

If you have installed an app or package and you're not sure about whether it belongs to the installed apps section, check the online documentation for that package.

Database Setup

SQLite 3 Database Setup

The current default database used by Django is the SQLite3 database. If you are planning to use this database, don't edit the code below.

```
example/settings.py
…

DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.sqlite3',
      'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
   }
}
…
```

PostgreSQL Database Setup

If planning to use PostgreSQL, make the following changes.

```
example/settings.py
…

DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.postgresql',
      'NAME': 'example',
      'USER': 'postgres',
      'PASSWORD': 'password123',
      'HOST': 'localhost',
      'PORT': '5432',
   }
}
…
```

ENGINE

Refers to the database component used to perform CRUD (create, read, update, delete) operations on the database data.

NAME

Name of the database. The name *'example'* refers to the name given to the database when it was created back in the *PostgreSQL Database Installation* section. If you gave the database a different name, replace it here.

USER

Username of the database. I haven't changed the username for this project. As you can see the username is *'postgres'*. If you changed the username when creating the database, include it here.

PASSWORD

This is the database password. The password that I created was *'password123'*. Include your own password if different. Refer to the *PostgreSQL Database Installation* section if you are having problems.

HOST

Refers to the host Django will use when connecting to the database. Since we are using our local machine, we put *'localhost'*. If that doesn't work, try *'127.0.0.1'*. If the HOST field is left empty, Django will connect using the localhost by default.

PORT

Refers to which port to use when connecting to the database. In my case, the port is *'5432'*. If the PORT field is left empty, Django will connect using the default port.

Static and Media Files Setup

Static Files

These files could be images, logos, JavaScript code, CSS files, and more.

Scroll all the way down to the end of settings.py. You will see that the *STATIC_URL* is already there. We need to add the following code as well.

example/settings.py

```
…

STATIC_URL = '/static/'

STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]

MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')

FILE_UPLOAD_PERMISSIONS = 0O640
```

STATIC_URL

Django will use this path and append it to the base URL of your website. For example, (http://websitename/static/style.css). It's a reference to the static files during development.

STATICFILES_DIRS

Refers to the location of static files in our project. You can have different paths listed if you have any additional directories that contain static files. It tells Django to look into the paths specified for extra static files that might exist in the project.

STATIC_ROOT

Refers to the absolute path which Django will collect static files from during production. It's not recommended to serve static files from your own local project, that is why we need to include this path when in production.

Media Files

These are files uploaded by users.

MEDIA_URL

Similar to *STATIC_URL* but refers to media files instead.

MEDIA_ROOT

Refers to the absolute path that points to the directory that will hold user-uploaded files.

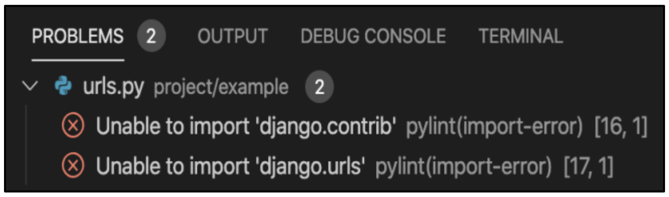
FILE_UPLOAD_PERMISSIONS

Sometimes Python throws an error regarding what users can do when an image is uploaded. In order to avoid inconsistencies with file permisions, please set *FILE_UPLOAD_PERMISSIONS = 0O640*.

Linter Setup (Optional)

A code linter analyses your code for potential errors such as syntax, structural, and best practice errors. If you're not bothered by the red line under your code, you can skip this setup. On the other hand, code linters are useful tools when it comes to learning programming languages.

Try opening a different file such as *urls.py* from the project directory. Check for problems in the lower part of the screen. If one of the problems shows pylint(import-error), follow the next steps to solve it.

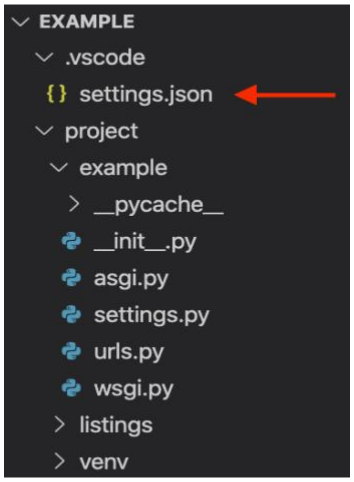


```
Unable to import 'django.contrib' pylint(import-error) [16, 1]
Unable to import 'django.urls' pylint(import-error) [17, 1]
```

1. Go to View > Command Palette > Python:SelectLinter > pylint. (Make sure the option Enable Linting is on, by default VS code enables pylint for Python code).

2. After you select pylint, a *.vscode* folder will appear on the left side of VS code.

3. Open the *settings.json* file that is inside the *.vscode* folder and make the following changes.



Old *settings.json* (You might have different content in the file).

.vscode/settings.json

```
{
   "python.linting.pylintEnabled": true,
   "python.linting.enabled": true
}
```

New *settings.json*.

.vscode/settings.json

```
{
   "team.showWelcomeMessage": false,
   "editor.formatOnSave": true,
   "python.linting.pycodestyleEnabled": true,
   "python.linting.pylintArgs": [
      "--load-plugins",
      "pylint_django"
   ],
   "python.pythonPath": "/usr/bin/python3",    #path might be different for you
   "python.linting.pylintEnabled": true,
   "python.linting.enabled": true
}
```

4. After you save *settings.json*, the red line under your code should go away.

Note:

If after saving *settings.json*, you still have a problem saying line too long (91 > 79 characters), you can disregard that one or you can break the line that is causing problems into two lines. This problem appears because it is best practice to have < 79 characters when writing code in Python. By the way, the code will run just fine even if you don't correct this problem.

Django Simple Web App

We have covered a lot up to this point. It can be overwhelming, but the best way to deal with this huge amount of information is to take it step by step.

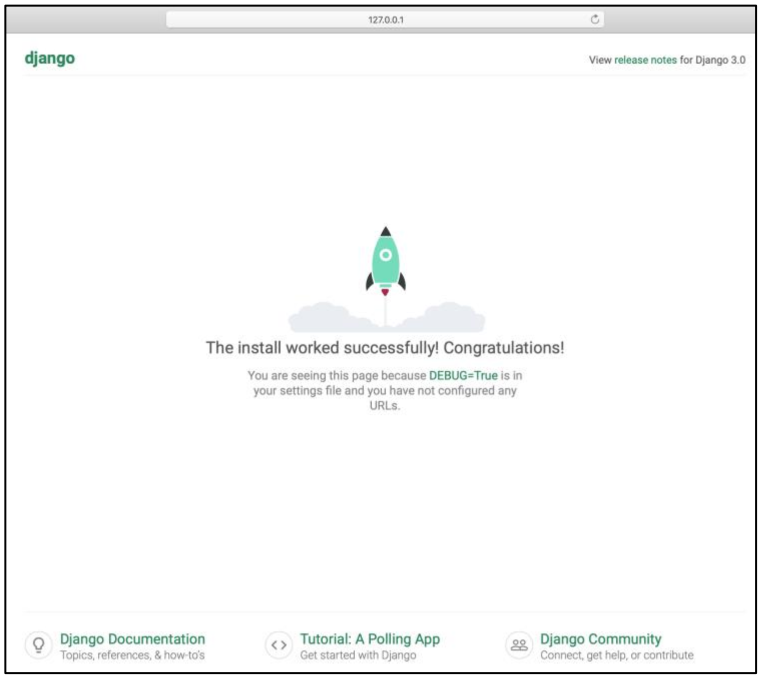
Run the following commands to see your first Django web app in action.

```
(venv) Bryans-MacBook-Pro-2:project balt1794$ python manage.py makemigrations
(venv) Bryans-MacBook-Pro-2:project balt1794$ python manage.py migrate
(venv) Bryans-MacBook-Pro-2:project balt1794$ python manage.py runserver
```

If everything went well, you should see a message like the one below. If not, please read the error in the terminal and troubleshoot accordingly.

```
System check identified no issues (0 silenced).
July 29, 2020 - 23:22:19
Django version 3.0.8, using settings 'example.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Go to http://127.0.0.1:8000, you should see the following.



The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Command	Explanation
django-admin startproject name .	- Creates a directory in your current working directory. - Important files such as settings.py and urls.py will be created. - name = your project name
python manage.py startapp name	- Creates an app which is essentially a submodule of your project. - name = your app name
python manage.py makemigrations	- Consolidates new changes made to models.py. - When you run this command for the first time, you should see the message, *No changes detected*. We get this message because we haven't edited models.py, but we will in the future.
python manage.py migrate	- All tables for your app are created in the database when executed.
python manage.py runserver	- Starts a web server on the local machine. The default port is 8000 and 127.0.0.1 for the IP address.

Projects vs Apps

A project is made up of the different apps and Python files needed to create and run a website. On the other hand, an app is a web application which is a part of a website. Apps perform a function or task on a website. For example, in our case, the listings app will take care of posts on the website and the users app will register new users.

Chapter 2: Django Basics

Django MVC Pattern

Django is a web framework which is also known as a Model-View-Controller (MVC) framework. Django follows the MVC, but there is a slight change when it comes to these concepts. The three most important parts to understand are the following.

Models

Django uses Object-Relational Mapping (ORM). This simplifies the interaction to the database since we don't have to write complex SQL to create tables and other stuff in the database. Models contain the fields and types of data to be stored in your database.

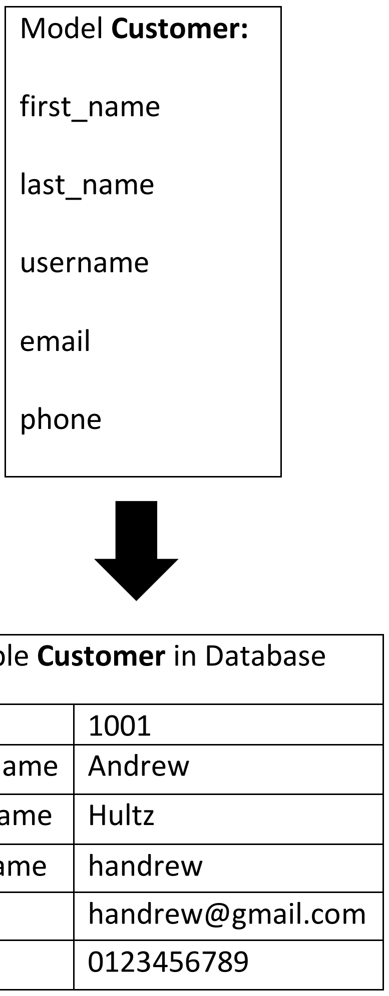


Table **Customer** in Database

ID	1001
first_name	Andrew
last_name	Hultz
username	handrew
email	handrew@gmail.com
phone	0123456789

Templates

Templates provide an interface between users and the website. The templates in Django are HTML templates. The templates don't have to be completely static; you can also add JavaScript to make the frontend more dynamic, or even use a totally different frontend such as ReactJS instead of the templates provided by Django.

Views

Django's views pass the data from the database to the templates, so that it can be shown to users in a more elegant and organized way, and not just as a bunch of tables.

There are two types of views; function-based views (used in this book) and class-based views.

URLs

To navigate around websites, Django uses URLs just like any other website. When a user wants to go to a page and clicks on a link, Django receives a URL request from the user and looks in *urls.py* to see to which view the URL being request is tied to. After Django calls the right view, this view renders the content to a template which is what user will see.

Django Admin Site

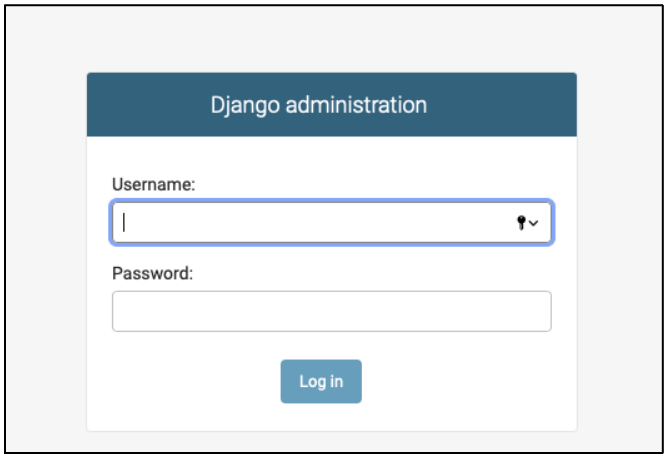
Django provides an admin panel which allows us to manage the content of our site. To access the admin panel, we need to create admin credentials first. Only people that you trust to manage the content of your site should have access to the admin site.

Let's put in practice all the theory by creating the listings model and going through the process of using the admin panel to create and manage some listings. We won't be using templates since the Django admin site already provides with one for us.

Create admin credentials by issuing the following command. You can use the credentials below, but feel free to enter your own as you go through the process.

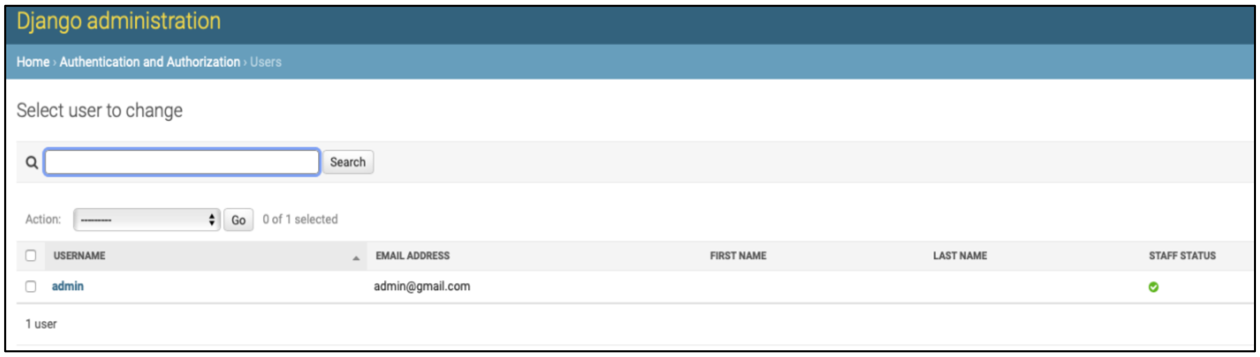
```
(venv) Bryans-MacBook-Pro-2:project balt1794$ python manage.py createsuperuser

Username: admin
Email address: admin@gmail.com
Password: password123
Password(again): password123
Superuser created successfully.
```

Issue the command *python manage.py runserver* and go to http://127.0.0.1:8000/admin/ to see the login page for the Django admin site.



Enter your credentials and log in. The admin site appears almost empty as of now since we haven't created any models yet, but you should see two fields which correspond to the *Authentication and Authorization* section which is provided by Django.

Django provides us with an authentication system that manages user accounts, groups, permissions, and more. Later on, we will use the Django authentication system to register and create users. For now, let's click on users to see the superuser that we created from the terminal.


Django administration
Home › Authentication and Authorization › Users
Select user to change
Search
Action: --------- Go 0 of 1 selected

USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
admin	admin@gmail.com			✓

1 user

You should see the username and email address we entered when creating the superuser. All the way to the right, you will see the staff status which shows a green check which means that this user can access the admin site since it's part of the staff.

The first and last name fields are empty, but if you select the user, you can add information, add the user to specific groups, edit fields, and even delete the user. Explore the admin panel before moving to the next section.

Listings Model

Since we already created the listings app and added it to *settings.py*, we can proceed to create the listings model in *models.py*.

Open *models.py* and start adding the following imports.

```
listings/models.py

from django.db import models
from django.utils.timezone import now
from datetime import datetime
```

from django.db import models

This import allows us to create models. Every model that we create will inherit from django.db.models.Model, in other words, each model is a subclass of this import.

from django.utils.timezone import now

This import gets the current time for each user based on their individual time zones. If users live in different time zones, this import will adjust that accordingly.

from datetime import datetime

Allows us to get the date and time by providing attributes such as year, month, day, hour, minute, second, and more.

The listings model will have different attributes also known as fields which users will be able to enter or select in order to create a listing. There are many fields that you can add to your listings model depending on your project requirements, so don’t feel obligated to include all the fields listed here. Feel free to edit and add your own fields as needed.

After we’ve added these imports, we can add the following code which will create the listings model.

listings/models.py

```
from django.db import models
from django.utils.timezone import now
from datetime import datetime
…
class Listings(models.Model):
   class SaleType(models.TextChoices):
      PICK_UP = "Available for pickup"
      SHIP = "Available for shipping"

   class ConditionType(models.TextChoices):
      USED = "Used"
      NEW = "New"

   class ProductType(models.TextChoices):
      BIKE= "Bike"
      PARTS = "Parts"
      OTHER = "Other"

   title = models.CharField(max_length=100)
   condition = models.CharField(
                  max_length=50, choices=ConditionType.choices,
                  default=ConditionType.USED)
   product_type = models.CharField(
                       max_length=50, choices=ProductType.choices,
                       default=ProductType.PRINTER)
   sale_type = models.CharField(
                  max_length=50, choices=SaleType.choices,
                  default=SaleType.SHIP)
   price = models.FloatField()
   address = models.CharField(max_length=100)
   city = models.CharField(max_length=100)
   state = models.CharField(max_length=100)
   zipcode = models.CharField(max_length=100)
   main_photo = models.ImageField()
   photo_1 = models.ImageField()
   photo_2 = models.ImageField(blank=True)
   list_date = models.DateTimeField(default=now)
   contact_email = models.CharField(max_length=50)

   def __str__(self):
      return self.title
```

CharField

Used for small to large-sized strings.

ImageField

This class has the same attributes and methods from FileField, but also checks if the uploaded file is a valid image. The FileField class can also be used to upload images.

DateTimeField

This class is used to display date and time.

FloatField

Used for floating-point numbers.

TextChoices

This is a subclass of the Choices class which allows us to display string choices.

blank

By default, blank is set to False which means that the field will appear as mandatory. This means that the field needs to be filled out otherwise the form won't be submitted.

When blank is set to True, users can leave the field empty and the form will still be submitted successfully.

__str__

This method allows us to convert an object into a string. This string representation will show in the admin panel with the field of the object that we return (in this case, the title field).

If we don't call this method, after we migrate the changes, the admin panel will return the object, but it wouldn't show a clear name in the admin panel hence difficult to identify.

Before we make migrations to the database, let's register the model in *admin.py*, so that it appears in our admin panel.

Go to *admin.py* in your app's folder and make the following changes.

```
listings/admin.py

from django.contrib import admin
from .models import Listings

admin.site.register(Listings)
```

from django.contrib import admin

Imports the admin module.

from .models import model_name

Imports the model into *admin.py*, so that we can use it without having to write it again. The dot tells Django to search for the model in *models.py* of the same directory as *admin.py*.

admin.site.register(model_name)

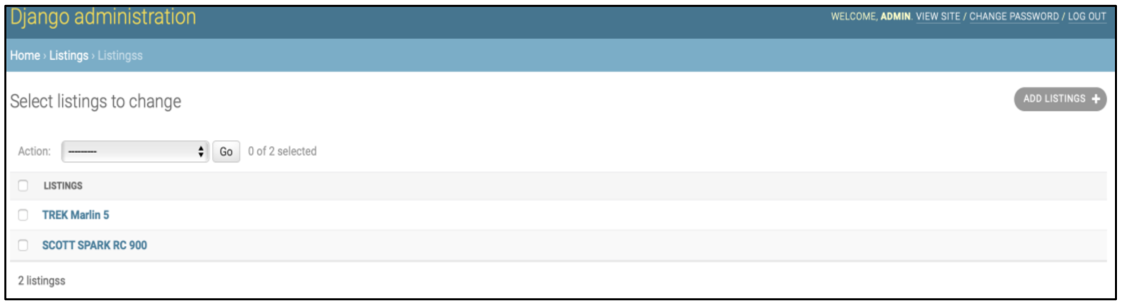
Registers the model in the admin site.

After we've made the respective changes to *models.py*, issue the migration commands and run the server.

```
(venv) Bryans-MacBook-Pro-2:project balt1794$ python manage.py makemigrations
(venv) Bryans-MacBook-Pro-2:project balt1794$ python manage.py migrate
(venv) Bryans-MacBook-Pro-2:project balt1794$ python manage.py runserver
```

After you run the server, access the admin site again. You should see a new option for *Listings* below *Groups and Users*. Select *Listings > Add Listings*.

Create some listings of your own. After you do, the listings page should look like the screenshot below. You can see and edit specific listings by clicking on them. Feel free to add more.


Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Listings › Listingss
ADD LISTINGS +
Action:
Go

Select listings to change

0 of 2 selected

LISTINGS
TREK Marlin 5
SCOTT SPARK RC 900

2 listingss

Chapter 3: Homepage

Let’s create a home page for our site, so that users can navigate from there to other pages and vice versa.

Open the *urls.py* file from your project folder.

```
example/urls.py

from django.contrib import admin
from django.urls import path

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

We need to include the URLs from the app that we created in order to be found by the main *urls.py* file which sits in our project folder. We do this every time that we create a new app.

To include the path to the URLs of your app, add the following code.

```
example/urls.py

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

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

from django.urls import path

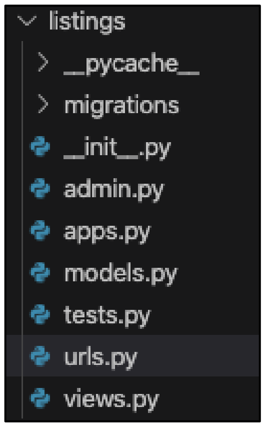
This module helps Django look for the variable *urlpatterns* which points to different paths in the website.

from django.urls import include

Django uses this module to include URLs from different apps inside the project.

Now we need to create a *urls.py* file in the directory of your app (*listings*) that will match the path set in the *urls.py* of the project.

The screenshot below shows how your app's directory should look after you create the new *urls.py* file.



The following steps will repeat every time we want to create a new page with a few exceptions.

Homepage URL Path

Open *urls.py* from your app's folder and add the following code. This is the path for the homepage.

listings/urls.py

```
from django.urls import path
from .import views

app_name = 'listings'

urlpatterns = [
   path(' ', views.index, name='index'),
]
```

from .import views

This module imports the views from your app's directory. We need this module because Django maps the URLs to the views and calls the functions in *views.py*.

In this particular case, the empty string (‘’) matches the base URL for the project. This is going to be our homepage. Django will call the index view. That is the reason why we have views.index to specify which view we want to call. On the other hand, name= ‘index’, is just a way to reference the URL pattern without having to write the entire URL, so instead we can just use the name ‘index’ to refer to the entire URL.

app_name

We already have multiple *urls.py* files; one for our project and one for our app. We will have another one for another app that we will create, so *app_name* helps Django identify and choose the correct *urls.py* file from other *urls.py* files in your project’s directory.

Homepage View

After you create the URL for a specific page, a view for the page also needs to be created.

Open *views.py* in your app’s directory. Let’s add the following code to create the view for the homepage.

listings/views.py

```
from django.shortcuts import render

def index(request):

    return render(request, 'listings/index.html')
```

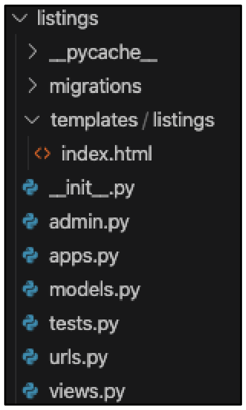
from django.shortcuts import render

This module is used to call different helper functions. We used it here to call render which helps generate a response that we can render in a template.

Homepage Template

After we have rendered the request into a template, we need to create such template, so that users can see the data being displayed. In Django, these templates are HTML files which display data provided by views.

First, let’s create a folder called *templates* inside your app’s directory and then inside the *templates* folder create a folder with the name of your app (*listings*). This helps Django interpret and find the templates for your app without any ambiguity. Inside that last folder created, let’s create a file named *index.html*. Your app’s main folder should look like the image below.



```
listings
  __pycache__
  migrations
  templates / listings
    index.html
  __init__.py
  admin.py
  apps.py
  models.py
  tests.py
  urls.py
  views.py
```

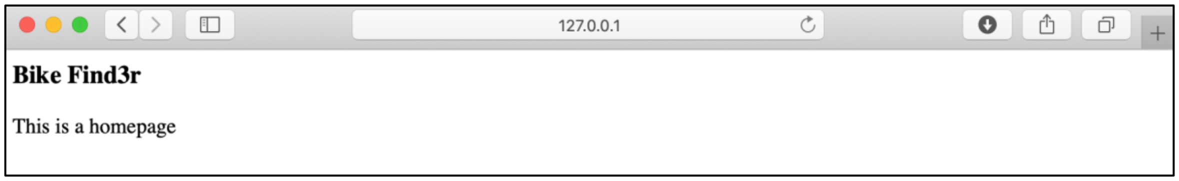
Open *index.html* and add the following code.

listings/templates/listings/index.html

```
<h3>Bike Find3r</h3>

<p>This is a homepage</p>
```

It might be a good idea to get familiar with HTML in order to understand the templates as we go along. For this one, there's not much going on; just a header and a paragraph. For now, we'll keep the templates like this, but later on, we will start styling them using bootstrap.

Let's try to get the functionality of our website working first, and then we can start making it look pretty. Run *python manage.py runserver* and go to http://127.0.0.1:8000/ to see the homepage of your site.



Bike Find3r

This is a homepage