# Django

Django is an excellent choice for any developer who wants to build modern, robust web applications with a minimal amount of code. It is popular, under active development, and

thoroughly battle-tested by the largest websites in the world. In the next chapter we’ll learn how to configure any computer for Django development.

**Chapter 1 : initial set up**

* The command line
* Install python 3
* Virtual environments
* Install Django
* Install git

**The command line**

* cd (change down a directory)
* cd .. (change up a directory)
* ls (list files in your current directory on Mac)
* dir (list files in your current directory on Windows)
* pwd (print working directory)
* mkdir (make directory)
* touch (create a new file on Mac)

**Install python 3**

* [https://wsvincent.com/install-python/#install-python-on-windows](https://wsvincent.com/install-python/" \l "install-python-on-windows)

**Virtual environments**

* Virtual environments are an indispensable part of Python programming.
* In this book we will use Pipenv21 to manage virtual environments. Pipenv is similar to npm and yarn
* from the JavaScript/Node ecosystem: it creates a Pipfile containing software dependencies
* and a Pipfile.lock for ensuring deterministic builds. “Determinism” means that each and every
* time you download the software in a new virtual environment, you will have exactly the same
* configuration.

**Install Django**

* Create folder and enter it
* Create virtual environment using pipenv

pipenv install django~=3.1.0

* Activate the virtual environment

Pipenv shell

* Start django-admin project and if installs then you are good to go

django-admin startproject config .

* To manage and migrate database in django run

python manage.py migrate

* Run django web server

python manage.py runserver

**Chapter 2 : Hello world**

**Folder structure in django**

* The config/**settings.py** file controls our project’s settings,
* **urls.py** tells Django which pages to build in response to a browser or URL request .
* **wsgi.py**, which stands for Web Server Gateway Interface33, helps Django serve our eventual web pages.
* The **manage.py** file is used to execute various Django commands such as running the local web server or creating a new app.
* **asgi.py** file, new to Django as of version 3.0 which allows for an optional Asynchronous Server Gateway Interface34 to be run.

**Create An App**

* To create app in django

python manage.py startapp pages

• **admin.py** is a configuration file for the built-in Django Admin app

• **apps.py** is a configuration file for the app itself

• **migrations**/ keeps track of any changes to our models.py file so our database and models.py stay in sync

• **models.py** is where we define our database models which Django automatically translates into database tables

• **tests.py** is for our app-specific tests

• **views.py** is where we handle the request/response logic for our web app

in order django know your app register your app to /setting.py

**URLs, Views, Models, Templates**

* Django **views** determine what content is displayed on a given page while
* **URLConfs** determine where that content is going.
* **The model** contains the content from the database and
* **The template** provides styling for it.

**URL -> View -> Model (typically) -> Template**

Remember how I said it can take three or four files for a given page? That’s because a model is

not always needed, in which case three files are enough.

**Difference between Django project vs django app**

A project refers to the entire application and all its parts.

An app refers to a sub module of the project. It's self-sufficient and not intertwined with the other apps in the project such that, in theory, you could pick it up and plop it down into another project without any modification. An app typically has its own models.py (which might actually be empty). You might think of it as a standalone python module. A simple project might only have one app.

For your example, the project is the whole website. You might structure it so there is an app for articles, an app for ranking tables, and an app for fixtures and results. If they need to interact with each other, they do it through well-documented public classes and accessor methods

**Chapter 3 :pages app**

**objectives**

* Templates
* build , test and deploy django website
* **Templates** : Every web framework needs a convenient way to generate HTML files and in Django the approach is to use templates: individual HTML files that can be linked together and also include basic logic.

**Steps for using templates**

* create folder called templates
* update config/settings

'DIRS': [str(BASE\_DIR.joinpath('Templates'))],

* update views by importing templateView which contains logic for showing out templates

from django.views.generic import TemplateView

Use classbasedviews instead of functional approach

**Steps for the Urls (class based views)**

-config/urls.py

from django.urls import path,include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('',include('pages.urls'))

]

-pages/urls

from django.urls import path

from .views import homeVPageView

urlpatterns =[

    path('',homeVPageView.as\_view(),name='home')

]

When using Class-Based Views, you always add as\_view() at the end of the view name.

**steps for the Tests**

**Chapter 4 : Message App**

Initial setup

…..

**Create database model**

* A **Django model** is the built-in feature that Django uses to create tables, their fields, and various constraints
* Our first task is to create a database model where we can store and display posts from our users.
* Django’s ORM will automatically turn this model into a database table for us

: posts/model.py

class posts(models.Model):

    text = models.TextField()

**Activating models**

Now that our new model is created we need to activate it. Going forward, whenever we create or modify an existing model we’ll need to update Django in a two-step process:

1. **First**, we create a migrations file with the **makemigrations** command. Migration files create a reference of any changes to the database models which means we can track changes–and debug errors as necessary–over time.

2. **Second**, we build the actual database with the migrate command which executes the instructions in our migrations file.

**Django-admin**

For us To use django-admin we need superuser

Python manage.py createsuperuser

To display our model in django-admin site we have to register

From .model import posts

Admin.site.register(posts)

To make our post descriptive add this function , This will display the first 50 characters of the text field.

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

return self.text(:50)

**Views**

-import listview to render model db

-import model name from .model

-create homepageclass to render the model

**Template**

**-**Create template and add template DIRS = BASE\_DIRS/”template”

-using for in loop render the list

**Test**

We will test : **models,urls, template**

//model test

-setup test data for models

-check if the model content contains the data you setup before

//url test

-test if url’s location exist

//template name , template content and url name test in one function

-test homepage = url name , template name , template content

**Chapter 5 Blog App**

In this chapter we will build a Blog application that allows users to create, edit, and delete posts.

The homepage will list all blog posts and there will be a dedicated detail page for each individual

blog post. We’ll also introduce CSS for styling and learn how Django works with static files.

**Initial Set Up**

As covered in previous chapters, our steps for setting up a new Django project are as follows:

• make a new directory for our code called blog

• install Django in a new virtual environment called .venv

• create a new Django project called django\_project

• create a new app blog

• perform a migration to set up the database

• update django\_project/settings.py and register your app

Step 1: **Models**

-create data models

-like title,body,author.

**Creating author field db In django** :

author = models.ForeignKey(

"auth.User",

on\_delete=models.CASCADE,

)

-use \_\_str\_\_ method for human readable post titles

-use get absolute url

def get\_absolute\_url(self):

return reverse("post\_detail", kwargs={"pk": self.pk})

Step 2 **Admin**

-create super user

-register data model

Step 3 :**Views**

- To show your database info to templates import ListView and model

from django.views.generic import ListView

from .models import Post

class BlogListView(ListView):

model = Post

template\_name = "home.html"

Step 4 : **URLS**

from .views import BlogListView

urlpatterns = [

path("", BlogListView.as\_view(), name="home"),

]

Step 5 : **Templates**

**-** Create template folder and home.html extending from base.html

**-**Register your template Directory in settings.py

-in the home.html

{% for post in post\_list %}

<div class="post-entry">

<h2><a href="">{{ post.title }}</a></h2>

<p>{{ post.body }}</p>

</div>

{% endfor %}

{% endblock content %}

**Individual Pages**

**-**in the **views** import DetailView

class BlogDetailView(DetailView):

# new

model = Post

template\_name = "post\_detail.html

**-**in the **url** import BlogDetailView

from django.urls import path

from .views import BlogListView, BlogDetailView

# new

urlpatterns = [

path("post/<int:pk>/",BlogDetailView.as\_view(), name="post\_detail"),

# new

path("", BlogListView.as\_view(), name="home"),

**Template**

{% extends "base.html" %}

{% block content %}

<div class="post-entry">

<h2>{{ object.title }}</h2>

<p>{{ object.body }}</p>

</div>

{% endblock content %}

**Static Files in Django**

If you recall, this is similar to how templates are treated as well.

**-**Create folder called **“static”**

**-**Config/settings.py

STATICFILES\_DIRS = [BASE\_DIR / "static"]

**-**Create folder inside static static/css/style.css

-in the base.html templates add the beginning

{% load static %}

<linkrel="stylesheet"href="{%static'css/base.css'%}">

And Phew its done .

**Chapter 6 Forms (CRUD Operation)**

* **Create**

-in the **views** import **CreateView** from django.view.generic.edit

class BlogCreateView(CreateView):

# new

model = Post

template\_name = "post\_new.html"

fields = ["title", "author", "body"]

-In the **urls.py** Create path link for create view name blogCreateView

path("post/new/", BlogCreateView.as\_view(), name="post\_new")

/Home.html/

-In the base.html Create <a> tag to link url path name for create view

<a href="{% url 'post\_new' %}">+ New Blog Post</a>

-in the templates lets create post\_new.html page for the form adding

<!-- templates/post\_new.html -->

{% extends "base.html" %}

{% block content %}

<h1>New post</h1>

<form action="" method="post">{% csrf\_token %}

{{ form.as\_p }}

<input type="submit" value="Save">

</form>

{% endblock content %}

**Update**

**views**

-First Import UpdateView

**-**template name in the update view post\_edit.html

class BlogUpdateView(UpdateView):

# new

model = Post

template\_name = "post\_edit.html"

fields = ["title", "body"]

**Urls**

-create path link for updateview

-path("post/<int:pk>/edit/",BlogUpdateView.as\_view(), name="post\_edit")

The link is based on id of individual post

**Template**

**-**In the post\_detail template add this a tag

<a href="{% url 'post\_edit' post.pk %}">+ Edit Blog Post</a>

-Then create this Html for edit page

<!-- templates/post\_edit.html -->

{% extends "base.html" %}

{% block content %}

<h1>Edit post</h1>

<form action="" method="post">{% csrf\_token %}

{{ form.as\_p }}

<input type="submit" value="Update">

</form>

{% endblock content %}

**Delete**

**Views**

-Import DeleteView

class BlogDeleteView(DeleteView):

# new

model = Post

template\_name = "post\_delete.html"

success\_url = reverse\_lazy("home")

**Url**

path("post/<int:pk>/delete/",BlogDeleteView.as\_view(),name="post\_delete"),

**Template**

-in the post detail add delete button or a tag link to post\_delete url

<p><a href="{% url 'post\_delete' post.pk %}">+ Delete Blog Post</a></p>

-create post\_delete.html

<!-- templates/post\_delete.html -->

{% extends "base.html" %}

{% block content %}

<h1>Delete post</h1>

<form action="" method="post">{% csrf\_token %}

<p>Are you sure you want to delete "{{ post.title }}"?</p>

<input type="submit" value="Confirm">

</form>

{% endblock content %}

**Chapter 7 User Accounts**

Implementing proper user authentication is famously hard; there are many security gotchas along the way so you really don’t want to implement this yourself. Fortunately, Django comes with a powerful, built-in user authentication system98 that we can use and customize as needed. Whenever you create a new project, by default Django installs the auth app, which provides us

with a User object containing:

• username

• password

• email

• first\_name

• last\_name

**Log In**

-In the config/urls.py

path("accounts/", include("django.contrib.auth.urls")),

-Then create the new directory.

mkdir templates/registration

-then create a new template file, templates/registration/login.html,

filled with the following code:

<!-- templates/registration/login.html -->

{% extends "base.html" %}

{% block content %}

<h2>Log In</h2>

<form method="post">{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Log In</button>

</form>

{% endblock content %}

-Then in the settings.py

LOGIN\_REDIRECT\_URL = "home"

**Updated Homepage**

-Create if else condition and is\_authenticated method and welcome user with his name home.html

{%If user.is\_authenticated%}

<p>Welcome {{user.username}} </p>

{%else%}

<p>You are not logged in </p>

{%endfor%}

**Log Out Link**

-in the settings.py

LOGOUT\_REDIRECT\_URL = "home"

-in the base.html add this

{%If user.is\_authenticated%}

<p>Welcome {{user.username}} </p>

<a href={%url ‘logout’%}>logout</a> #new

{%else%}

<p>You are not logged in </p>

{%endfor%}

**Sign up**

We need to write our own view for a sign up page to register new users, but Django provides us with a form class, UserCreationForm103, to make things easier. By default it comes with three fields: username, password1, and password2.

-Start new app called accounts

python manage.py startapp accounts

-in the settings.py add

“accounts”

-config/urls.py register accounts url

Path(“accounts /”include(“accounts.urls”)),

-in the **views**

from django.contrib.auth.forms import UserCreationForm

from django.urls import reverse\_lazy

from django.views.generic import CreateView

class SignUpView(CreateView):

form\_class = UserCreationForm

success\_url = reverse\_lazy("login")

template\_name = "registration/signup.html"

-in the accounts/urls.py import signupview

from django.urls import path

From .views import SignUpView

urlpatterns[

Path(“signup/”,SignUpView.as\_view(),name=”signup”)]

-in the template/registration/signup.html

{% extends "base.html" %}

{% block content %}

<h2>Sign Up</h2>

<form method="post">{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Sign Up</button>

</form>

{% endblock content %}

**Sign Up Link**

<!-- templates/base.html-->

...

<p>You are not logged in.</p>

<a href="{% url 'login' %}">Log In</a> |

<a href="{% url 'signup' %}">Sign Up</a>

...

**Chapter 8: Custom User Model**

To create custom user model we will use Abstract user

After Initial setup for account app and config project

Creating our custom user model requires four steps:

• update django\_project/settings.py

AUTH\_USER\_MODEL = "accounts.CustomUser"

• Create a new CustomUser model

from django.db import models

from django.contrib.auth.models import AbstractUser

*# Create your models here.*

class CustomUser(AbstractUser):

    age = models.PositiveIntegerField(null=True, blank=True)

• Create new forms for UserCreationForm and UserChangeForm

from django.contrib.auth.forms import UserCreationForm, UserChangeForm

from .models import CustomUser

class CustomUserForm(UserCreationForm):

    class Meta(UserCreationForm):

        model = CustomUser

        fields = UserCreationForm.Meta.fields + ("age",)

class CustomUserChange(UserChangeForm):

    class Meta:

        model = CustomUser

        fields = UserChangeForm.Meta.fields

• Update accounts/admin.py

from django.contrib import admin

from django.contrib.auth.admin import UserAdmin

from .forms import CustomUserForm, CustomUserChange

from .models import CustomUser

class CustomUserAdmin(UserAdmin):

    add\_form = CustomUserForm

    form = CustomUserChange

    model = CustomUser

    list\_display = [ "email", "username", "age", "is\_staff", ]

    fieldsets = UserAdmin.fieldsets + ((None, {"fields": ("age",)}),)

add\_fieldsets = UserAdmin.add\_fieldsets + ((None, {"fields": ("age",)}),)

admin.site.register(CustomUser, CustomUserAdmin)

**Chapter 9: User Authentication**

-Create templates folder and inside templates folder create registration folder

mkdir templates / Mkdir registration

-Register your templates inside settings.py for django to know

TEMPLATES = [

{

...

"DIRS": [BASE\_DIR / "templates"], *# this is the line you need*

...

}

]

If you think about what happens when you log in or log out of a site, you are immediately redirected to a subsequent page. We need to tell Django where to send users in each case.

The LOGIN\_REDIRECT\_URL and LOGOUT\_REDIRECT\_URL settings do that. We’ll configure both to redirect

to our homepage which will have the named URL of 'home'.

Settings.py

LOGIN\_REDIRECT\_URL = "home"

LOGOUT\_REDIRECT\_URL = "home"

Now we can create four new templates within our text editor:

• templates/base.html

• templates/home.html

• templates/registration/login.html

• templates/registration/signup.html

templates/base.html

<!DOCTYPE html>

<html> <head> <meta charset="utf-8"> <title>{% block title %}Newspaper App{% endblock title %}</title>

</head> <body> <main>

{% block content %}

{% endblock content %}

</main>

</body>

</html>

templates/home.html

{% extends "base.html" %}

{% block title %}Home{% endblock title %}

{% block content %}

{% if user.is\_authenticated %}

Hi {{ user.username }}!

<p><a href="{% url 'logout' %}">Log Out</a></p>

{% else %}

<p>You are not logged in</p> <a href="{% url 'login' %}">Log In</a> |

<a href="{% url 'signup' %}">Sign Up</a>

{% endif %}

{% endblock content %}

templates/registration/login.html

{% extends "base.html" %}

{% block title %}Log In{% endblock title %}

{% block content %}

<h2>Log In</h2> <form method="post">{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Log In</button>

</form>

{% endblock content %}

templates/registration/signup.html

{% extends "base.html" %}

{% block title %}Sign Up{% endblock title %}

{% block content %}

<h2>Sign Up</h2>

<form method="post">{% csrf\_token %}

    {{ form.as\_p }}

    <button type="submit">Sign Up</button>

</form>

{% endblock content %}

**URLS**

-config/urls.py

from django.contrib import admin

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

from django.views.generic.base import TemplateView *# new*

urlpatterns = [

    path("admin/", admin.site.urls),

    path("accounts/", include("accounts.urls")), *# new*

    path("accounts/", include("django.contrib.auth.urls")), *# new*

    path("", TemplateView.as\_view(template\_name="home.html"),name="home"), *# new*

]

-accounts/urls.py

from django.urls import path

from .views import SignUpView

urlpatterns = [

    path("signup/", SignUpView.as\_view(), name="signup"),

]

**Views**

View to set up signup logic because login and logout already have builtin

-Signup view

from django.urls import reverse\_lazy

from django.views.generic import CreateView

from .forms import CustomUserForm

class SignUpView(CreateView):

    form\_class = CustomUserForm

    success\_url = reverse\_lazy('login')

    template\_name = "registration/signup.html"

-Forms

Add email field in the forms.py

from django.contrib.auth.forms import UserCreationForm, UserChangeForm

from .models import CustomUser

class CustomUserForm(UserCreationForm):

    class Meta(UserCreationForm):

        model = CustomUser

      fields = ( "username","email", "age", ) *#*

class CustomUserChange(UserChangeForm):

    class Meta:

        model = CustomUser

        fields = ( "username","email", "age", ) *#*

**Chapter 10 : Bootstrap**

-To use bootstrap go to the base.html and cdn link of bootstrap min css and bootstrap js bundle

-To restyle the text that comes with signup page 150 characters etc we will use django-crispy-forms that makes working with forms easier DRY,

-installation

python -m pip install django-crispy-forms==1.13.0

python -m pip install crispy-bootstrap5==0.6

-register in the installed apps

*# 3rd party app*

    "crispy\_forms", *# new*

    "crispy\_bootstrap5", *#*

-add last part of settings.py

*# crispy*

CRISPY\_ALLOWED\_TEMPLATE\_PACKS = "bootstrap5" *# new*

CRISPY\_TEMPLATE\_PACK = "bootstrap5"

-To use in the signup or login page both are same way

*<!-- templates/registration/login.html -->*

{% extends "base.html" %}

{%load crispy\_forms\_tags%} // load crispy

{% block title %}

    Log In

{% endblock title %}

{% block content %}

    <h2>Log In</h2>

    <form method="post">

        {% csrf\_token %}

        {{ form|crispy }} // then use like this

        <button class="btn btn-success  mt-3" type="submit">Log In</button>

    </form>

{% endblock content %}

**Chapter 11 : Password Change and Reset**

**Password Change**

Letting users change their passwords is a common feature on many websites. Django provides a default implementation that already works at this stage. To try it out first click on the “Log In” button to make sure you’re logged in. Then navigate to the **“Password change”** page, which is located at http://127.0.0.1:8000/accounts/password\_change/.

**Customizing Password Change**

Let’s customize these two password change pages so that they match the look and feel of our Newspaper site. Because Django already has created the views and URLs for us we only need Password Change and Reset

to change the templates. In your text editor create two new template files in the registration directory:

• templates/registration/password\_change\_form.html

{% extends 'base.html' %}

{% load crispy\_forms\_tags %}

{% block content %}

    <form action="" method="post">

        {% csrf\_token %}

        {{ form|crispy }}

        <input class="btn btn-success" type="submit" value="Change my password">

    </form>

{% endblock content %}

• templates/registration/password\_change\_done.html

{% extends 'base.html' %}

{%load crispy\_forms\_tags%}

{% block title %}Password Change Done{% endblock title %}

{% block content %}

<p>Password was succesfully changed</p>

<a href="{% url 'home'  %}">Go back home</a>

{% endblock content %}

**Password Reset**

* Password reset handles the common case of users forgetting their passwords. The steps are very similar to configuring password change.
* The only configuration required is telling Django how to send emails.
* for testing purposes we can rely on Django’s console backend134 setting which outputs the email text to our command line console instead.
* At the bottom of the django\_project/settings.py file add the following one-line change.

EMAIL\_BACKEND = "django.core.mail.backends.console.EmailBackend"

* And we’re all set! Django will take care of all the rest for us. Let’s try it out. Navigate to
* http://127.0.0.1:8000/accounts/password\_reset/ to view the default password reset page. Reset form

{% extends 'base.html' %}

{%load crispy\_forms\_tags%}

{% block title %}forgot Password{% endblock title %}

{% block content %}

<h1>Forgot your password?</h1>

<p>Enter your email address below, and we'll email instructions

    for setting a new one.</p>

<form action="" method="post"> {% csrf\_token %}

    {{form|crispy}}

    <input class="btn btn-primary" type="submit" value="Send me instructions!">

</form>

{% endblock content %}

* Upon submission you’ll then be redirected to the password reset done page at: http://127.0.0.1:8000/accounts/password\_reset/done/

{% extends "base.html" %}

{% block title %}

    Email Sent

{% endblock title %}

{% block content %}

    <h1>Check your inbox.</h1>

    <p>

        We've emailed you instructions for setting your password.

        You should receive the email shortly!

    </p>

{% endblock content %}

* In the terminal of vscode will be sent to reset confirm link follow that link for reset confirm

{% extends "base.html" %}

{% load crispy\_forms\_tags %}

{% block title %}

    Enter new password

{% endblock title %}

{% block content %}

    <h1>Set a new password!</h1>

    <form method="POST">

        {% csrf\_token %}

        {{ form|crispy }}

        <input class="btn btn-success" type="submit" value="Change my password">

    </form>

{% endblock content %}

* Reset complete

{% extends "base.html" %}

{% block title %}

    Password reset complete

{% endblock title %}

{% block content %}

    <h1>Password reset complete</h1>

    <p>Your new password has been set.</p>

    <p>

        You can log in now on the

        <a href="{% url 'login' %}">Log In page</a>.

    </p>

{% endblock content %}

* The last point is to add login.html forgot password

<p><a href="{% url 'password\_reset'  %}">Forgot Password</a></p>