

# Python - Django

[Django official documentation page](https://docs.djangoproject.com/en/4.0/) - <https://docs.djangoproject.com/en/4.0/>

"django-admin startproject monthly\_challenges"

"python3 manage.py runserver" -> runs the server locally on port 8000.  
(run this in command line).

"python3 manage.py startapp challenges" -> build app "Django" app.  
(run this in command line).

To Start a venv - "source tutorial-env/bin/activate" if the name of the venv is "tutorial-env".

EXAMPLE - "source /Users/remez19/vsProjects/Django/envDjango/bin/activate "

Views - in Django views are defined by functions or classes.

We write the code in the "App" directory in the views.py file.

The screenshot shows a code editor interface with two panes. The left pane displays the project structure:

```
OPEN EDITORS
  × views.py monthly_challenges/challenges
DJANGO
  > envDjango
  monthly_challenges
    challenges
      > migrations
      __init__.py
      admin.py
      apps.py
      models.py
      tests.py
      views.py
    > monthly_challenges
    db.sqlite3
    manage.py
```

The right pane shows the contents of the views.py file in the challenges app:

```
monthly_challenges > challenges > views.py
1  from django.shortcuts import render
2
3  # Create your views here.
4
```

We want to return response to the client. the function name can be anything we like.

The return type is an Object called - "HttpResponse()", inside the () we pass the data we want to response with."

The parameter that the function gets is a request. Django handle the request and calls the right function.

Inorder to bind the function with the event that triggers it we create a new file "urls.py".

Inside this file we create a list of all the urls we want in our website.

```

views.py          urls.py
monthly_challenges > challenges > urls.py > ...
1  from django.urls import path
2  from . import views
3  ...
4  The path function takes two arguments:
5  1. String of the url we want to support in the example we use 'january'.
6  2. A pointer to the view function that should be executed when a request comes to this url.
7  ...
8  urlpatterns = [
9      path("january", views.index)
10 ]
11

```

Next we need to connect the "App" to the project - we do this by adding a path to the 'urls.py' file in the project directory.

Like this: \*\* Add '/' after challenges.

```

monthly_challenges > monthly_challenges > urls.py > ...
6  Function views
7      1. Add an import: from my_app import views
8      2. Add a URL to urlpatterns: path('', views.home, name='home')
9  Class-based views
10     1. Add an import: from other_app.views import Home
11     2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
12 Including another URLconf
13     1. Import the include() function: from django.urls import include, path
14     2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
15 ...
16 from django.contrib import admin
17 from django.urls import path, include
18
19 urlpatterns = [
20     path('admin/', admin.site.urls),
21     path("challenges", include("challenges.urls")),
22 ]
23

```

### Dynamic Path Segments & Captured Values:

*Using '<>' let django to know that we don't care about the value of the url that comes after challenges,*

*Instead we tell him that all the urls that starting with 'challenges/' should handle by the same function.*

*Then we need to add an argument to the views.monthly\_challenges function with the name we chose in this case "month".*

```

def monthly_challenges(request, month):
    if month == "404":
        return HttpResponseNotFound("MotherFucker not supported!")
    else:
        return HttpResponse(month)

```

```
urlpatterns = [
    path("<month>", views.monthly_challenges),
```

```
]
```

#### Path Converters:

With this we can specify what we do expect the url to be (str, int , ...).

The order in with we write this patterns matter. In the exampmle below we first try to see if the url can be an int, then as a string.

```
urlpatterns = [
    path("<int:month>", views.monthly_challenges_number),
    path("<str:month>", views.monthly_challenges),
```

#### Redirects:

To redirect to other views we use the "HttpResponseRedirect()", as an argument we pass the url we want to redirect to.

```
def monthly_challenges_number(request, month):
    if month == 1:
        return HttpResponse(the_remez_david_var)
    else:
        return HttpResponseRedirect("/challenges/boyohboy")
```

#### The Reverse Function & Named URLs:

A method to make redirections more dynamic and not hard coded example below:

```
urlpatterns = [
    path("<int:month>", views.monthly_challenges_number),
    path("<str:month>", views.monthly_challenges, name='month_challenges'),
```

```
def monthly_challenges_number(request, month):
    if month == 404:
        return HttpResponseRedirect("MotherFucker not supported!")
    if month == 1:
        return HttpResponse(the_remez_david_var)
    else:
        # return HttpResponseRedirect("/challenges/Shame") Can be more dynamic by switching to :
        # effectively builds up a path -> /challenges/Shame
        redirect_path = reverse("month_challenges", args=["Shame"])
        return HttpResponseRedirect(redirect_path)
```

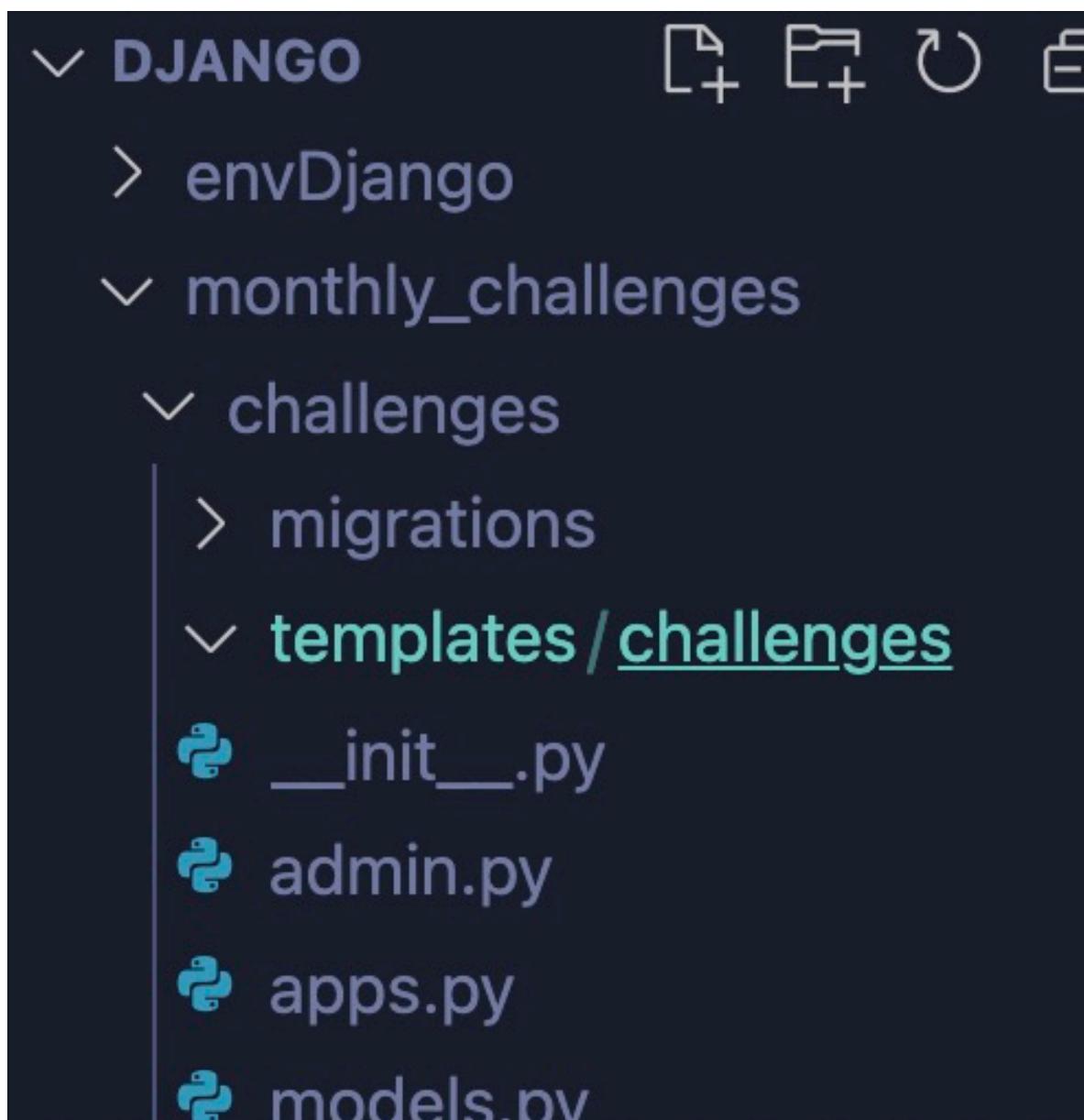
Sending html files and injecting data into them.

### Adding & Registering Templates:

The templates are html files that we can inject some data into them.

First we create a sub folder named "templates" inside the relevant Django app (This is a convention).

Then we create another sub-folder inside the "templates" folder. The name of this folder should be the name of the app.



```
from django.template.loader import render_to_string
```

In order to tell Django to look for our templates we need to change the "settings.py" file and add the relevant paths as in the example:

First way : (Making Django aware of our app )

```

monthly_challenges > monthly_challenges > settings.py > ...
23 SECRET_KEY = 'django-insecure-qb9@zig6*l$n7*olof+*owb72z#t!facqmsl3a*55dt#5n)rgv'
24
25 # SECURITY WARNING: don't run with debug turned on in production!
26 DEBUG = True
27
28 ALLOWED_HOSTS = []
29
30
31 # Application definition
32
33 INSTALLED_APPS = [
34     'challenges',
35     'django.contrib.admin',
36     'django.contrib.auth',
37     'django.contrib.contenttypes',
38     'django.contrib.sessions',
39     'django.contrib.messages',
40     'django.contrib.staticfiles',
41 ]
42

```

Second option - adding the path to the templates like this:

```

EXPLORER ... urls.py .../challenges views.py settings.py urls.py .../monthly_challenges
monthly_challenges > monthly_challenges > settings.py > ...
40     'django.contrib.auth.middleware.AuthenticationMiddleware',
41     'django.contrib.messages.middleware.MessageMiddleware',
42 ]
43
44 ROOT_URLCONF = 'monthly_challenges.urls'
45
46 TEMPLATES = [
47     {
48         'BACKEND': 'django.template.backends.django.DjangoTemplates',
49         'DIRS': [
50             BASE_DIR / "challenges" / "templates"
51         ],
52         'APP_DIRS': True,
53         'OPTIONS': {
54             'context_processors': [
55                 'django.template.context_processors.debug',
56                 'django.template.context_processors.request',
57                 'django.contrib.auth.context_processors.auth',
58                 'django.contrib.messages.context_processors.messages',
59             ],
60         },
61     },
62 }
63
64 WSGI_APPLICATION = 'monthly_challenges.wsgi.application'
65
66
67 # Database
68
69 # ...
70
71
72
73
74
75 # Database

```

After that we can use the like this:

```

def remez_page(request, month):
    response_to_send = ""
    youtubers = list(youtube_dic.keys())
    if month in youtubers:
        response_to_send = render_to_string(youtube_dic[month])
        return HttpResponse(response_to_send)
    else:
        return HttpResponseNotFound("Not a valid page!")

# Template

```

Or we can do it like this:

```

def remez_page(request, month):
    youtubers = list(youtube_dic.keys())
    if month in youtubers:
        return HttpResponse(render(request, youtube_dic[month]))
    else:
        return HttpResponseNotFound("Not a valid page!")

```

Django built in filters - <https://docs.djangoproject.com/en/4.0/ref/templates/builtins/>

Creating dynamic urls the right way using the built in DTL

```

<ul>
    {% for youtuber in list_of_youtubers %}
        <li><a href="{% url "home" youtuber%}">{{youtuber | title}}</a></li>
    {% endfor %}
</ul>

```

Template Inheritance:

We this we can inject a block of data into a HTML file like this:

1. Adding a folder named templates.
2. Creating the files that we want other templates to inherit from.
3. Adding the dir for Django to look for in the "settings.py" file.
4. Finally we tell Django where to inject our data.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ block_page_title }}</title>
  </head>
  <body>
    {{ block_page_content }}
  </body>
</html>

```

```

{% extends 'base.html' %}

{% block page_title %}
  Home Page
{% endblock %}

{% block page_content %}
  <h1>Welcome to the best youtubers site!</h1>
  <h2>Here is a list of a cool youtubers:</h2>
  <ul>
    {% for youtuber in list_of_youtubers %}
      <li><a href="{% url "home" youtuber%}">{{youtuber | title}}</a></li>
    {% endfor %}
  </ul>
{% endblock %}

```

## Including Partial Template Snippets:

Important to know that vars that we can use the vars that are in other files that use include

Example including a header:

```

<header>
  <nav>
    <a href="{% url "home" %}">All Youtubers List</a>
  </nav>
</header>

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

[08/Apr/2022 14:22:44] "GET /challenges/Pee%20Pee%20Master HTTP/1.1" 200 453
[08/Apr/2022 14:22:46] "GET /challenges/Moist%20Master HTTP/1.1" 200 442
[08/Apr/2022 14:22:48] "GET /challenges/Scambuster%20Master HTTP/1.1" 200 462
[08/Apr/2022 14:36:50] "GET /challenges/ HTTP/1.1" 200 666
[08/Apr/2022 14:36:51] "GET /challenges/Pee%20Pee%20Master HTTP/1.1" 200 559
[08/Apr/2022 14:36:51] "GET /challenges/Moist%20Master HTTP/1.1" 200 666
[08/Apr/2022 14:36:54] "GET /challenges/Peek%20Master HTTP/1.1" 200 559
[08/Apr/2022 14:37:03] "GET /challenges/ HTTP/1.1" 200 666
[08/Apr/2022 14:37:13] "GET /challenges/ HTTP/1.1" 200 666

```

{%extends "base.html"%}
  ...
  {%block page_title%}
    Youtuber Slide
  {%endblock%}
  {%block page_content%}
    {% include "challenges/includes/header.html" %}
    <h1>This is SCAMBUSTER MASTER!</h1>
    <h2>This youtuber is a sick one!</h2>
    <a href="https://www.youtube.com/c/SomeOrdinaryGamers">Scambuster - Youtube channel</a>
  {%endblock%}

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

[08/Apr/2022 14:22:44] "GET /challenges/Pee%20Pee%20Master HTTP/1.1" 200 453
[08/Apr/2022 14:22:46] "GET /challenges/Moist%20Master HTTP/1.1" 200 442
[08/Apr/2022 14:22:48] "GET /challenges/Scambuster%20Master HTTP/1.1" 200 462
[08/Apr/2022 14:36:50] "GET /challenges/ HTTP/1.1" 200 666
[08/Apr/2022 14:36:51] "GET /challenges/Pee%20Pee%20Master HTTP/1.1" 200 559
[08/Apr/2022 14:36:51] "GET /challenges/Moist%20Master HTTP/1.1" 200 666
[08/Apr/2022 14:36:54] "GET /challenges/Peek%20Master HTTP/1.1" 200 559
[08/Apr/2022 14:37:03] "GET /challenges/ HTTP/1.1" 200 666
[08/Apr/2022 14:37:13] "GET /challenges/ HTTP/1.1" 200 666

## 404 Template:

Adding a 404.html file to the “templates” folder in the root project and using “raise Http404()” will load our custom 404.html file to present the relevant data.

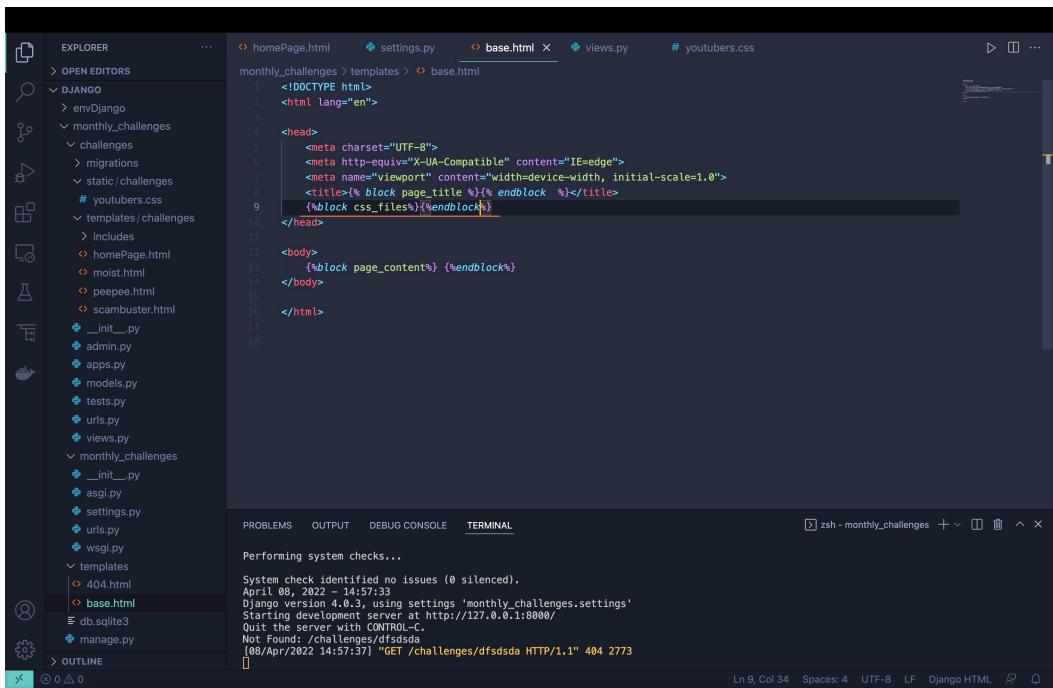
## Adding Static Files (CSS files):

1. Adding a folder named “static” to the app we want. (Same concept as the templates folder for the html)
2. Best practice is to create another sub-folder inside the static folder with name of our app.

```

    \ static/challenges
    \ templates/challenges
      > includes
      <> HomePage.html
      <> moist.html

```



The screenshot shows a code editor interface with the following details:

- EXPLORER:** Shows the project structure:
  - DJANGO
    - envDjango
    - monthly\_challenges
      - challenges
      - migrations
      - static/challenges
        - # youtubers.css
        - templates / challenges
          - > includes
          - <> HomePage.html
          - <> moist.html
          - <> peepoe.html
          - <> scambuster.html
      - \_\_init\_\_.py
      - admin.py
      - apps.py
      - models.py
      - tests.py
      - urls.py
      - views.py
    - monthly\_challenges
      - \_\_init\_\_.py
      - asgi.py
      - settings.py
      - urls.py
      - wsgi.py
    - templates
      - 404.html
      - base.html
      - manage.py
  - OPEN EDITORS:** Shows the current open files:
    - HomePage.html
    - settings.py
    - base.html
    - views.py
    - # youtubers.css
  - EDITOR:** Displays the content of the base.html file.
  - TERMINAL:** Shows the output of a system check command:

```

Performing system checks...
System check identified no issues (0 silenced).
April 08, 2022 - 14:57:33
Django version 4.0.3, using settings 'monthly_challenges.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[08/Apr/2022 14:57:37] "GET /challenges/dfsdsda HTTP/1.1" 404 2773

```

The screenshot shows a Django project structure on the left and a code editor on the right.

**Project Structure:**

- OPEN EDITORS
- DJANGO
  - envDjango
  - monthly\_challenges
    - challenges
    - migrations
    - static/challenges
      - # youtubers.css
      - templates/challenges
      - includes
        - homePage.html
        - moist.html
        - peepee.html
        - scambuster.html
    - \_\_init\_\_.py
    - admin.py
    - apps.py
    - models.py
    - tests.py
    - urls.py
    - views.py
  - monthly\_challenges
    - \_\_init\_\_.py
    - asgi.py
    - settings.py
    - urls.py
    - wsgi.py
  - OUTLINE

**Code Editor (homePage.html):**

```
<% extends "base.html" %>
<% load static %>
<% block css_files%>
<link rel="stylesheet" href="{% static "challenges/youtubers.css" %}">
</block>

<% block page_title%>
    Home Page
    </block>
<% block page_content%>
<h1>Welcome to the best youtubers site!</h1>
<h2>Here is a list of a cool youtubers:</h2>
<ul>
    {% for youtube in list_of_youtubers %}
        <li><a href="{% url "home" youtube%}">{{youtube | title}}</a></li>
    {% endfor %}
</ul>
</block>
```

**Terminal:**

```
zsh - monthly_challenges + □ ^ x
Performing system checks...
System check identified no issues (0 silenced).
April 08, 2022 - 14:57:33
Django version 4.0.3, using settings 'monthly_challenges.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
Not Found: /challenges/dfsdsda
[08/Apr/2022 14:57:37] "GET /challenges/dfsdsda HTTP/1.1" 404 2773
```

Ln 4, Col 66 Spaces: 4 UTF-8 LF Django HTML ↵

## Adding Global Static Files:

Same idea as the global templates file.

<https://fonts.google.com>

As an example we add a global css file to control our overall font style:

The screenshot shows a Visual Studio Code interface with the following details:

- Left Sidebar:** Explorer view showing the project structure:
  - OPEN EDITORS
  - DJANGO
    - envDjango
    - monthly\_challenges
      - challenges
      - migrations
      - static
        - templates / challenges
          - includes
          - homepage.html
          - moist.html
          - peeppee.html
          - scambuster.html
        - \_\_init\_\_.py
        - admin.py
        - apps.py
        - models.py
        - tests.py
        - urls.py
        - views.py
      - \_\_init\_\_.py
      - asgi.py
      - settings.py
      - urls.py
      - wsgi.py
    - static
    - # styles.css
    - templates
    - 404.html
    - base.html
    - db.sqlite3
  - Central Area:** Code editor showing `homePage.html` file content:

```
monthly_challenges > static > # styles.css > base.html > views.py > youtubers.css
monthly_challenges > static > # styles.css > body
1   @import url('https://fonts.googleapis.com/css2?family=Rubik+Moonrocks&display=swap');
2   html{
3     font-family: 'Rubik Moonrocks', cursive;
4   }
5
6   body{
7     margin: 0;
8 }
```
  - Bottom Bar:** PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL. The TERMINAL tab is active, displaying the output of a system check:

```
Performing system checks...
System check identified no issues (0 silenced).
April 08, 2022 - 15:31:34
Django version 4.0.3, using settings 'monthly_challenges.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[08/Apr/2022 15:31:34] "GET /challenges/ HTTP/1.1" 200 741
[08/Apr/2022 15:31:34] "GET /static/challenges/youtubers.css HTTP/1.1" 200 409
```
  - Right Sidebar:** Google Password Required message: Enter your password for "rdoneplus" in Internet Accounts.

The screenshot shows the VS Code interface with the following details:

- EXPLORER**: Shows the project structure under `DJANGO`, including `monthly_challenges`, `migrations`, `static`, `templates`, and `views`.
- CODE**: Displays the `settings.py` file content. Key parts include:
  - Internationalization: `LANGUAGE_CODE = 'en-us'`, `TIME_ZONE = 'UTC'`, `USE_I18N = True`, `USE_TZ = True`.
  - Static files: `STATIC_URL = 'static/'`, `STATICFILES_DIRS = [BASE_DIR / "static"]`.
  - Default primary key field type: `DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'`.
- TERMINAL**: Shows command-line output related to Django development.

## Migrations:

Creating classes that eventually Django will transform into a data base tables as such:

The screenshot shows the `models.py` file with the following content:

```

models.py ✘  settings.py ✘  0001_initial.py ✘
book_outlet > models.py > ...
2   from django.forms import CharField, IntegerField
3
4   # Create your models here.
5
6
7   # Represent a table with three coloms - id, title, rating
8   class Book(models.Model):
9       # id - comes automatically, but can be modified if needed
10      # by using id = models.AutoField()
11      title = models.CharField(max_length=50)
12      rating = models.IntegerField()
13

```

Next we need to make Django aware of this classes so we use the following command:

**"python3 manage.py makemigrations"**

Than we follow up this command by another command:

**"python3 manage.py migrate"**

With this command Django goes over all the migration folders (of each app) And make sure to run all migrations that not have been executed yet.

To work with the data base that we created we use the following command:

**"Python3 manage.py shell"** this will open up a interactive shell that will let us insert data or retrive data from the database that we created.

The pic below shows how we can save data in the data base and save it.

```
remez19 book_store %python3 manage.py shell
Python 3.8.9 (default, Feb 18 2022, 07:45:33)
[Clang 13.1.6 (clang-1316.0.21.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from book_outlet.models import Book
>>> harry_potter = Book(title="Harry poter 1", rating=5)
>>> harry_potter.save()
```

## Getting all Entries

Using the "all()" method will give us back all the data entries that we created

```
>>> Book.objects.all()
<QuerySet [<Book: Book object (1)>, <Book: Book object (2)>]
>>>
```

Overriding the str function in our classes:

We override the method with the name "\_\_str\_\_" (similar to `toString()` in Java)

```
6
7      # Represent a table with three coloms - id, title, rating
8  class Book(models.Model):
9      # id - comes automatically, but can be modified if needed
10     # by using id = models.AutoField()
11     title = models.CharField(max_length=50)
12     rating = models.IntegerField()
13
14     def __str__(self) -> str:
15         return f"{self.title} ({self.rating})"
16
```

Output Change after the override.

```

remez19 book_store %ls
book_outlet    book_store    db.sqlite3      manage.py
remez19 book_store %python3 manage.py shell
Python 3.8.9 (default, Mar 30 2022, 13:51:16)
[Clang 13.1.6 (clang-1316.0.21.2.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from book_outlet.models import Book
>>> Book.objects.all()
<QuerySet [<Book: Harry poter 1 (5)>, <Book: The Lord Of The Rings (4)>]>
>>> []

```

### Using Validators:

We want to use validators in order to control the data we want to insert.  
As an example in this book project we would like to control the rating field (  $1 \leq \text{rating} \leq 5$  ):  
Import built in validators (We can also construct our own validators see - <https://docs.djangoproject.com/en/4.0/ref/validators/>):

```

from django.core.validators import MinValueValidator, MaxValueValidator

# Create your models here.

```

```

7
8  # Represent a table with three coloms - id, title, rating
9  class Book(models.Model):
10     # id - comes automatically, but can be modified if needed
11     # by using id = models.AutoField()
12     title = models.CharField(max_length=50)
13     rating = models.IntegerField(validators=[          MinValueValidator(1),
14                                         MaxValueValidator(5),
15                                         ])
16
17
18     def __str__(self) -> str:
19         return f"{self.title} ({self.rating})"
20

```

If we already have records in our database that didn't have the files we added when we first insert them, than we have to set a default Value for them using: "null=True" (Setting null=True) key-argument or the "default=" key-argument.  
Also we can use the "blank=True" to allow empty fields in our database

```

    # This represents a table with three columns - id, title, rating
9   class Book(models.Model):
10      # id - comes automatically, but can be modified if needed
11      # by using id = models.AutoField()
12      title = models.CharField(max_length=50)
13      rating = models.IntegerField(validators=[
14          MinValueValidator(1),
15          MaxValueValidator(5),
16      ])
17      author = models.CharField(null=True, max_length=100)
18      is_bestselling = models.BooleanField(default=False)
19
20      def __str__(self) -> str:
21          return f'{self.title} ({self.rating})'
22

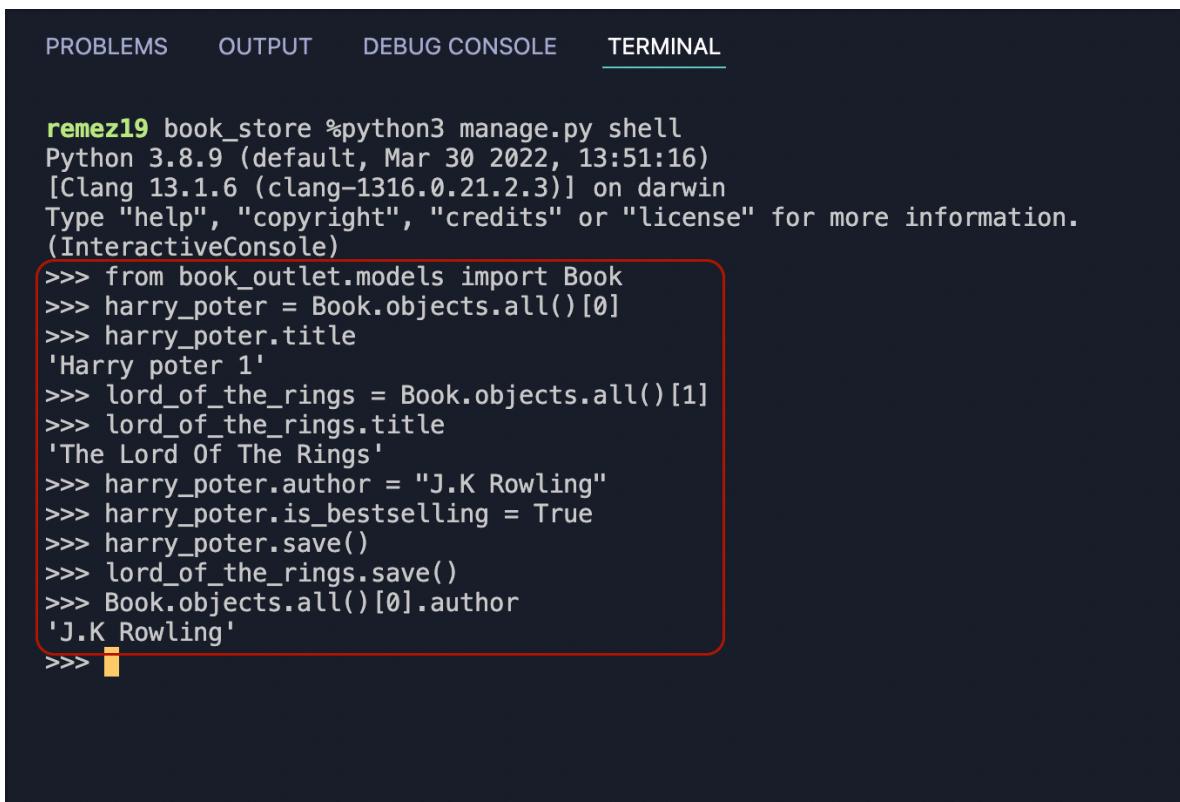
```

### After this changes we run two commands:

1. python3 manage.py makemigrations.
2. python3 manage.py migrate.

### Updating Data:

Updating the two new fields we added - "author" and "is\_bestselling":



```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

remez19 book_store %python3 manage.py shell
Python 3.8.9 (default, Mar 30 2022, 13:51:16)
[Clang 13.1.6 (clang-1316.0.21.2.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from book_outlet.models import Book
>>> harry_poter = Book.objects.all()[0]
>>> harry_poter.title
'Harry poter 1'
>>> lord_of_the_rings = Book.objects.all()[1]
>>> lord_of_the_rings.title
'The Lord Of The Rings'
>>> harry_poter.author = "J.K Rowling"
>>> harry_poter.is_bestselling = True
>>> harry_poter.save()
>>> lord_of_the_rings.save()
>>> Book.objects.all()[0].author
'J.K Rowling'
>>>

```

### Deleting Data:

```
>>> harry_poter = Book.objects.all()[0]
>>> harry_poter.delete()
(1, {'book_outlet.Book': 1})
>>> █
```

### Create Instead of Save:

```
>>> Book.objects.create(title="Harry Potter 1", author= "J.K Rowlling", rating=5, is_bestselling=True)
<Book:
    Book Ttitle: Harry Potter 1
    Book author: J.K Rowlling
    Book Rating: (5)
    Book Best Selling: True
    >
    ... Book.objects.all()
```

### Querying & Filtering Data:

By id:

```
>>> from book_outlet.models import Book
>>> Book.objects.get(id=3)
<Book:
    Book Ttitle: Harry Potter 1
    Book author: J.K Rowlling
    Book Rating: (5)
    Book Best Selling: True
    >
    ... Book.objects.all()
```

By title:

```
>>> Book.objects.get(title="My Story")
<Book:
    Book Ttitle: My Story
    Book author: Remez
    Book Rating: (3)
    Book Best Selling: False
    >
    ... Book.objects.all()
```

And just like this for every field we create.

**Get will always give just one entry even if we have multiple entries that have the same data.**

**If there are more than one entry than get will raise an error!**

If we want to get all the entries that fulfill some conditions that we will want to use “**filter**”:

```
>>> Book.objects.filter(is_bestselling=False)
<QuerySet [
```

```
>>> Book.objects.filter(is_bestselling=False, rating=4)
<QuerySet [
```

“rating\_\_lte = 4” => rating<=4

```
>]>
>>> Book.objects.filter(rating__lte=4)
<QuerySet [
```

Official filters documentation : <https://docs.djangoproject.com/en/4.0/topics/db/queries/>

### "or" Conditions

Starting by importing the built-in module "Q" and as follows in the example.

For using "or" we use -> "|", for "and" we use -> ","

```
>>> from django.db.models import Q
>>> Book.objects.filter(Q(rating__lt=3) | Q(is_bestselling=True))
<QuerySet [
```

### Rendering Queried Data in the Template:

Getting the list of books from our Book model and passing the data to the "index.html" template.

```
views.py  X  index.html
book_outlet > views.py > ...
1  from django.shortcuts import render
2  from .models import Book
3  # Create your views here.
4
5
6  def index(request):
7      books = Book.objects.all()
8      return render(request, "book_outlet/index.html", {
9          "books": books,
10     })
11
```

Then we continue to loop thru all the list of books in the "index.html" template.

```
views.py  X  index.html
book_outlet > templates > book_outlet > index.html
1  {% extends 'book_outlet/base.html' %}
2  {% block title%}
3  All Books
4  {% endblock %}
5  {% block content %}
6      <ul>
7          {% for book in books%}
8              <li>{{book.title}} (Rating: {{book.rating}})</li>
9          {% endfor %}
10     </ul>
11
12 {% endblock %}
```

Another examples:

A screenshot of a code editor showing two files: `views.py` and `urls.py`. The `views.py` file contains Python code for a Django application. It includes two functions: `index` which returns all books, and `book_detail` which returns a specific book based on its ID. The `book_detail` function also includes logic to check if the book is a bestseller. The `urls.py` file is partially visible at the top.

```
book_outlet > views.py > book_detail.html
6 def index(request):
7     books = Book.objects.all()
8     return render(request, "book_outlet/index.html", {
9         "books": books,
10    })
11
12
13 def book_detail(request, id):
14     book = Book.objects.get(pk=id)
15     return render(request, "book_outlet/book_detail.html", [
16         "title": book.title,
17         "author": book.author,
18         "rating": book.rating,
19         "is_bestselling": book.is_bestselling,
20    ])
21
```

A screenshot of a code editor showing two files: `index.html` and `book_detail.html`. The `book_detail.html` file is a Django template that extends the base template and defines a content block. It displays the book's title and author, and includes logic to determine if the book is a bestseller using an `if` statement. The `index.html` file is partially visible at the top.

```
book_outlet > templates > book_outlet > book_detail.html
1  {% extends 'book_outlet/base.html' %}
2  {% block title %}
3      {{title}}
4  {% endblock title %}
5
6  {% block content %}
7      <h1>{{title}}</h1>
8      <h2>{{author}}</h2>
9      <p>The book has a rating of {{rating}} |
10         {% if is_bestselling %}
11             and is a bestseller.
12         {% else %}
13             but isn't a bestseller.
14         {% endif %}
15     </p>
16  {% endblock content %}
```

## Model URLs:

```
views.py      urls.py      index.html      models.py      book_de
book_outlet > models.py > Book > get_absolute_url
15
16     MaxValueValidator(5),
17 ]
18     author = models.CharField(null=True, max_length=100)
19     is_bestselling = models.BooleanField(default=False)
20
21     def __str__(self) -> str:
22         return f"""
23             Book Title: {self.title}
24             Book Author: {self.author}
25             Book Rating: ({self.rating})
26             Book Best Selling: {self.is_bestselling}
27         """
28
29     def get_absolute_url(self):
30         return reverse("book-detail", kwargs={"pk": self.pk})
```

```
views.py      urls.py      index.html      models.py      book_detail.html
book_outlet > templates > book_outlet > index.html
1  {% extends 'book_outlet/base.html' %} 
2  {% block title%}
3  All Books
4  {% endblock %}
5  {% block content %}
6      <ul>
7          {% for book in books%}
8              <li><a href="{{book.get_absolute_url}}">{{book.title}}</a> (Rating: {{book.rating}})</li>
9          {% endfor %}
10     </ul>
11
12 {% endblock %}
```

## Adding a Slugfield & Overwriting Save:

```
from django.db import models
from django.forms import CharField, IntegerField
from django.core.validators import MinValueValidator, MaxValueValidator
from django.utils.text import slugify

# Create your models here.

# Represent a table with three coloms - id, title, rating
class Book(models.Model):
    # id - comes automatically, but can be modified if needed
    # by using id = models.AutoField()
    title = models.CharField(max_length=50)
    rating = models.IntegerField(validators=[MinValueValidator(1), MaxValueValidator(5)])
    author = models.CharField(null=True, max_length=100)
    is_bestselling = models.BooleanField(default=False)
    # Harry Potter 1 ==> harry-potter-1
    slug = models.SlugField(default="", null=False)

    def __str__(self) -> str:
        return f"""
        Book Title: {self.title}
        Book Author: {self.author}
        Book Rating: {self.rating}
        Book Best Selling: {self.is_bestselling}
        """

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

    def save(self, *args, **kwargs):
        self.slug = slugify(self.title)
        super().save(*args, **kwargs)
```

## Using the Slug & Updating Field Options:

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

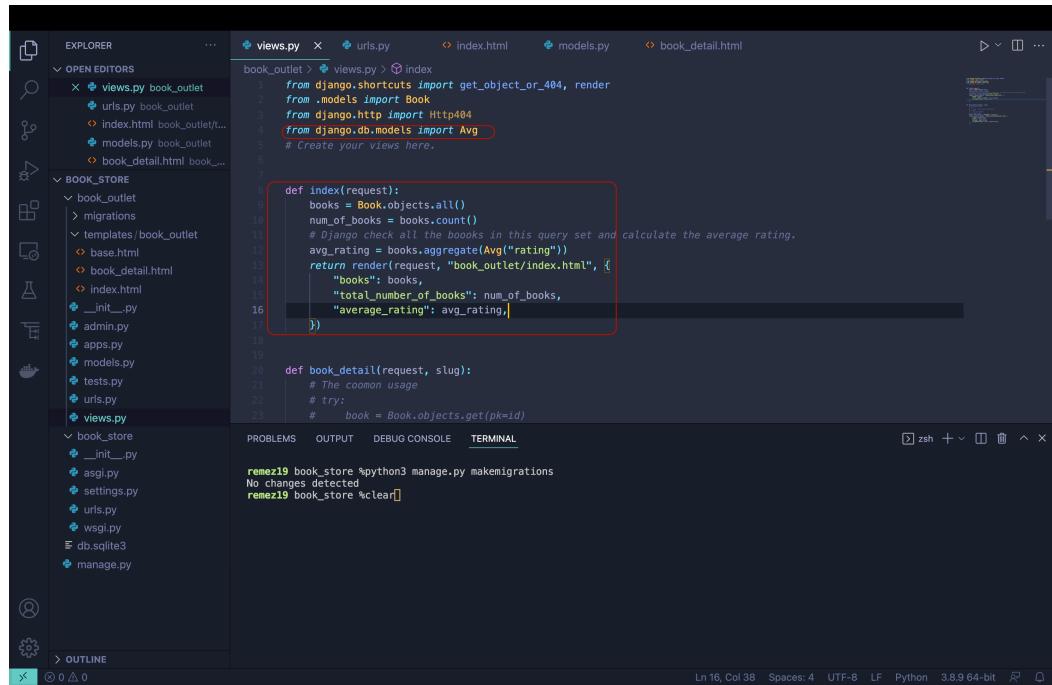
urlpatterns = [
    path("", views.index),
    path(["<slug:slug>"], views.book_detail, name="book-detail")
]
```

```

def book_detail(request, slug):
    # The common usage
    # try:
    #     book = Book.objects.get(pk=id)
    # except:
    #     raise Http404()
    # Alternative way:
    book = get_object_or_404(Book, slug=slug)
    return render(request, "book_outlet/book_detail.html",
                  {"title": book.title,
                   "author": book.author,
                   "rating": book.rating,
                   "is_bestselling": book.is_bestselling,
})

```

## Aggregation & Ordering:



The screenshot shows a code editor interface with the following details:

- EXPLORER:** Shows the project structure:
  - BOOK\_STORE** folder contains:
    - book\_outlet folder: views.py, urls.py, index.html, base.html, book\_detail.html, \_\_init\_\_.py, admin.py, models.py, tests.py, urls.py, wsgi.py.
    - migrations
    - templates/book\_outlet: index.html, book\_detail.html
  - book\_store folder: \_\_init\_\_.py, asgi.py, settings.py, urls.py, wsgi.py, db.sqlite3, manage.py.
- views.py:** The code is highlighted with a red box around the aggregation part.
 

```

from django.shortcuts import get_object_or_404, render
from .models import Book
from django.http import Http404
from django.db.models import Avg

# Create your views here.

def index(request):
    books = Book.objects.all()
    num_of_books = books.count()
    # Django check all the books in this query set and calculate the average rating.
    avg_rating = books.aggregate(Avg("rating"))
    return render(request, "book_outlet/index.html", {
        "books": books,
        "total_number_of_books": num_of_books,
        "average_rating": avg_rating
    })

def book_detail(request, slug):
    # The common usage
    # try:
    #     book = Book.objects.get(pk=id)

```
- PROBLEMS:** Shows command-line output from the terminal:
 

```

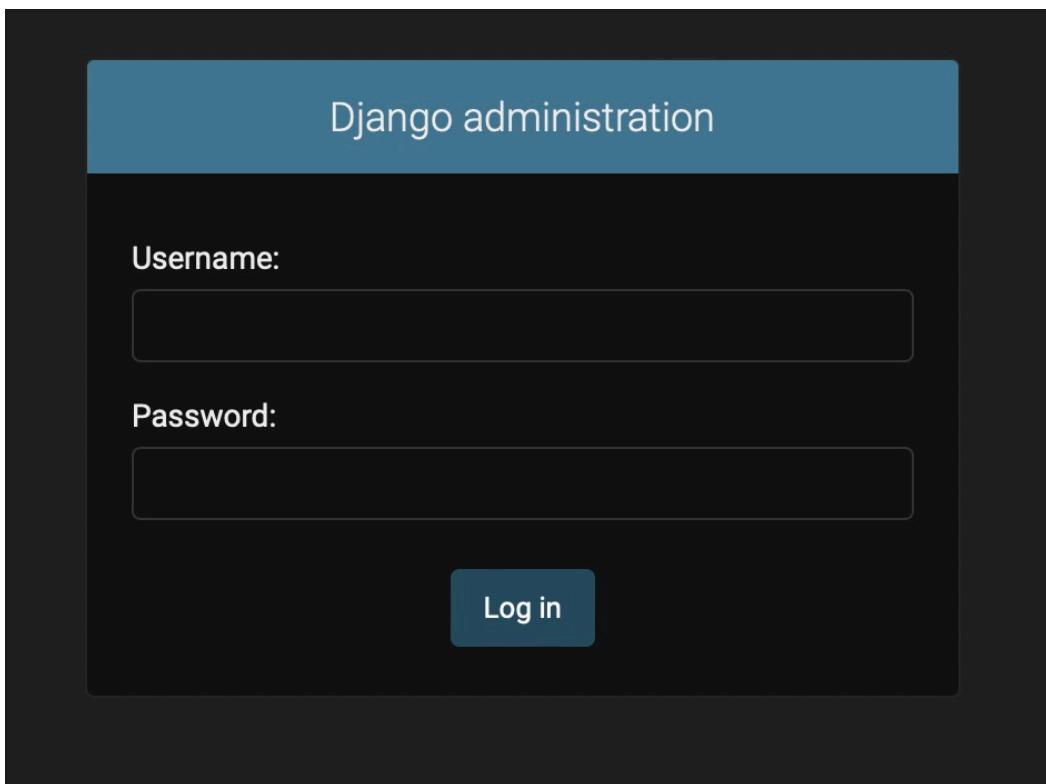
remez19 book_store %python3 manage.py makemigrations
No changes detected
remez19 book_store %clear

```
- OUTPUT, DEBUG CONSOLE, TERMINAL:** Standard terminal panes.
- STATUS:** Shows the current session details: Ln 16, Col 38, Spaces: 4, UTF-8, LF, Python 3.8.9 64-bit.

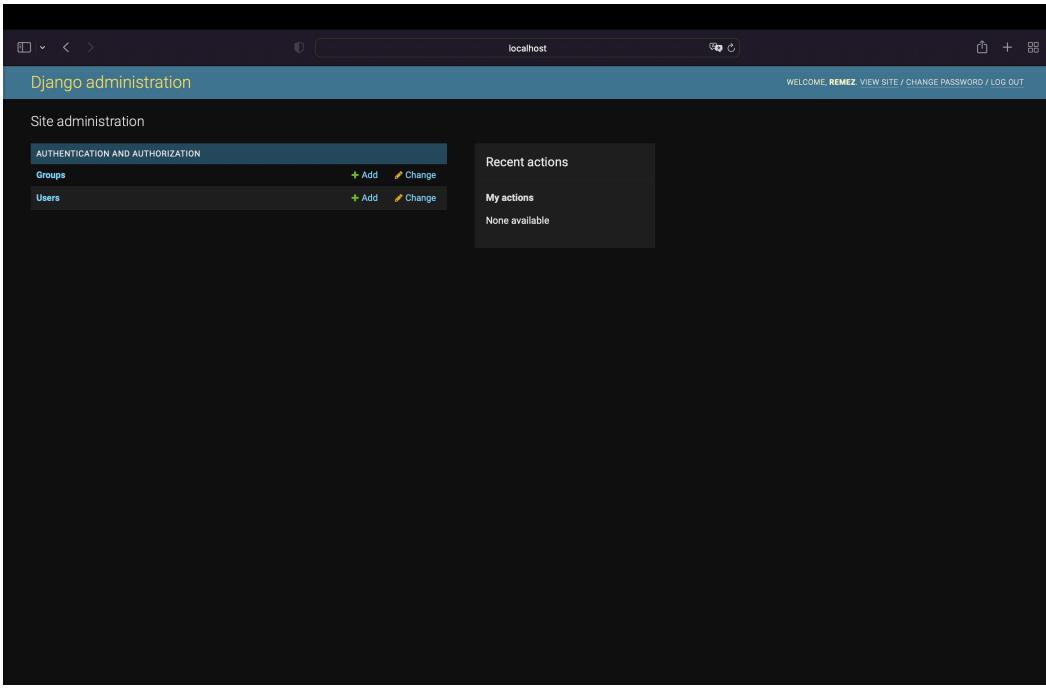
```
views.py      urls.py      index.html X      models.py      b
book_outlet > templates > book_outlet > index.html
1  {% extends 'book_outlet/base.html' %}           |
2  {% block title%}                                |
3  All Books                                     |
4  {% endblock %}                                 |
5  {% block content %}                            |
6    <ul>                                         |
7      {% for book in books%}                   |
8          <li><a href="{{book.get_absolute_url}}">{{book.title}}</a>|
9          {% endfor %}                         |
10     </ul>                                       |
11     <hr>                                         |
12     <p>                                         |
13         Total Number Of Books: {{total_number_of_books}}|
14     </p>                                         |
15     <p>Average Rating: {{average_rating.rating_avg}}</p>|
16 {% endblock %}
```

### Logging Data Into the Admin Panel:

First we need to run the command - "python3 manage.py createsuperuser". This command let us create a new super user "admin". With this user we can log into the Django admin section.



After login in:



## Adding Models to the Admin Area:

The screenshot shows the Visual Studio Code (VS Code) interface. On the left is the Explorer sidebar showing the project structure of a Django application named "book\_store". The "admin.py" file in the "book\_outlet" directory is open in the editor. The code in "admin.py" is:

```
from django.contrib import admin
# First we need to import our models.
from .models import Book
# Register your models here.
admin.site.register(Book)
```

Below the editor is the Terminal tab, which shows the output of running the Django development server and some log messages. The log includes:

```
Django version 4.0.3, using settings 'book_store.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[24/Apr/2022 12:24:38] "GET /admin/book_outlet/book/ HTTP/1.1" 200 7134
[24/Apr/2022 12:24:38] "GET /admin/i18n/ HTTP/1.1" 200 3343
/UUsers/remez19/vsProjects/Django/book_store/book_outlet/models.py changed, reloading.
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
April 24, 2022 - 12:24:47
Django version 4.0.3, using settings 'book_store.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[24/Apr/2022 12:24:58] "GET /admin/book_outlet/book/ HTTP/1.1" 200 7082
[24/Apr/2022 12:24:58] "GET /admin/i18n/ HTTP/1.1" 200 3343
```

At the bottom, status information is shown: "Ln 8, Col 1", "Spaces: 4", "UTF-8", "LF", "Python", "3.8.9 64-bit".

Now we will have this tab we can click on and see the "book" table:

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups + Add Change

Users + Add Change

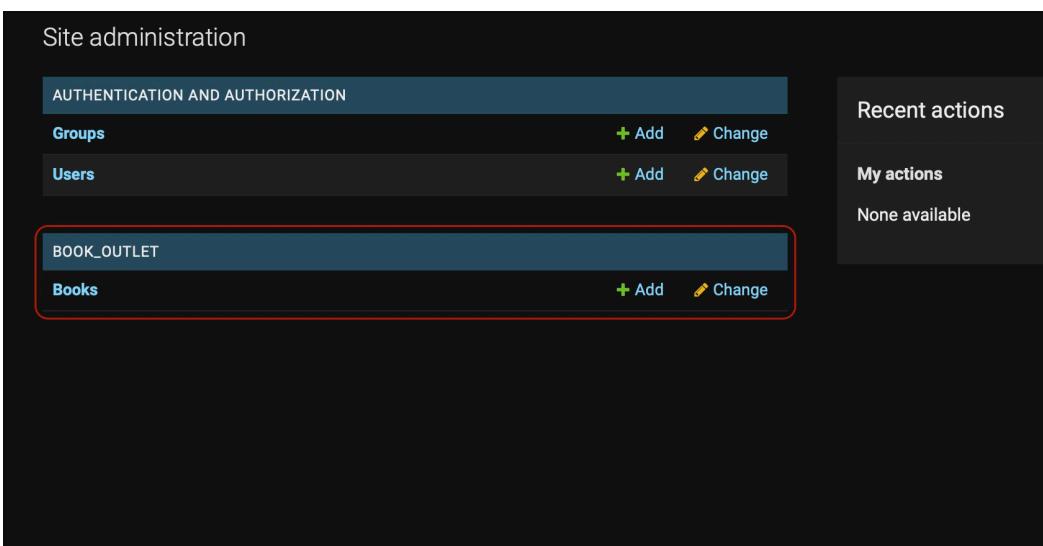
BOOK\_OUTLET

Books + Add Change

Recent actions

My actions

None available



We this we can click on "Books" and see all our books the are currently in our table and, we can also edit book data if we want to:

Django administration

Home > Book\_Outlet > Books

WELCOME, REMEZ. VIEW SITE / CHANGE PASSWORD / LOG OUT

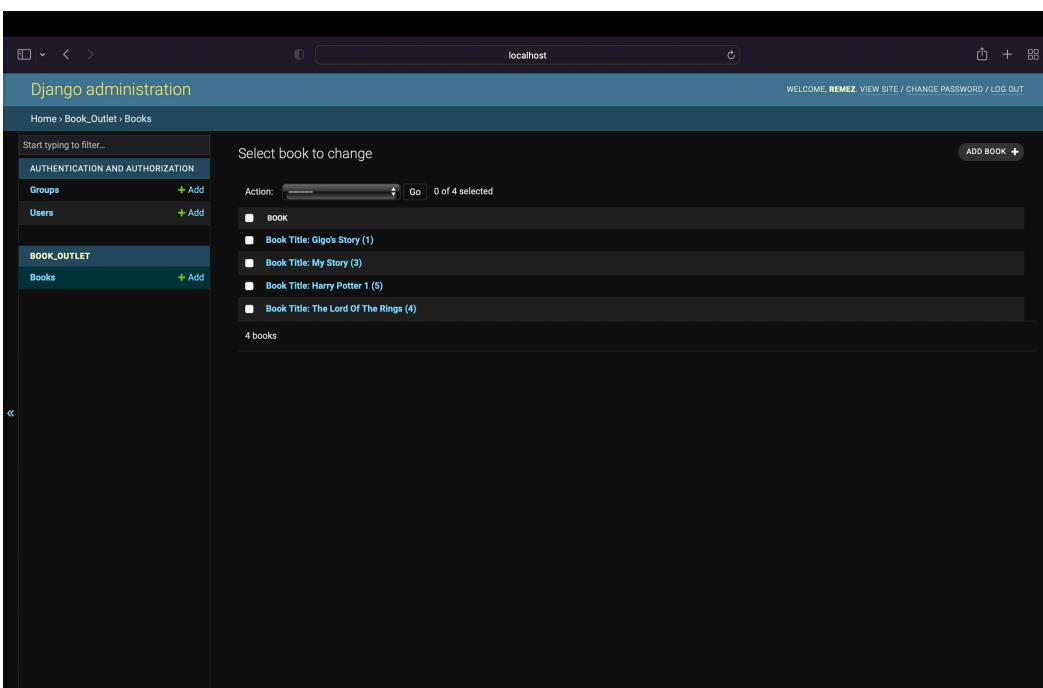
Select book to change

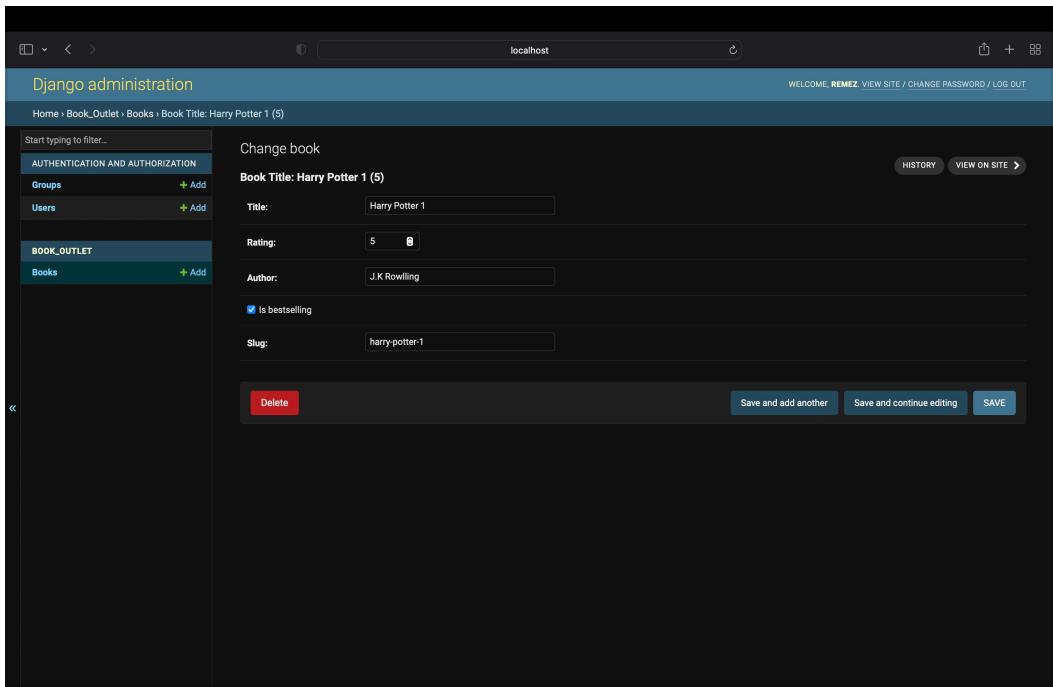
Action: — Go 0 of 4 selected

BOOK

- Book Title: Gigo's Story (1)
- Book Title: My Story (3)
- Book Title: Harry Potter 1 (5)
- Book Title: The Lord Of The Rings (4)

4 books





## Configuring Model Fields:

```

    admin.py      models.py ×

book_outlet > models.py > Book > __str__
15     rating = models.IntegerField(validators=[_
16         MinValueValidator(1),
17         MaxValueValidator(5),
18     ])
19     author = models.CharField(null=True, max_length=100)
20     is_bestselling = models.BooleanField(default=False)
21     # Harry Potter 1 => harry-poter-1
22     slug = models.SlugField(default="", null=False, blank=True, db_index=True)

```

With this field set to True - we can leave this field empty when we creating a new book thru the Django admin interface.

We can also make fields uneditable by:

```

author = models.CharField(null=True, max_length=100)
is_bestselling = models.BooleanField(default=False)
# Harry Potter 1 => harry-poter-1
slug = models.SlugField(default="", null=False, blank=True, editable=False, db_index=True)

def __str__(self) -> str:

```

## Configuring the Admin Settings:

```
py admin.py × py models.py

book_outlet > py admin.py > BookAdmin
1   from django.contrib import admin
2
3   # First we need to import our models.
4   from .models import Book
5   # Register your models here.
6
7
8   class BookAdmin(admin.ModelAdmin):
9       readonly_fields = ("slug",)
10
11
12   admin.site.register(Book, BookAdmin)
13
```

We can also make fields auto-populated by another fields as in this example:

```
class BookAdmin(admin.ModelAdmin):
    # readonly_fields = ("slug",)
    prepopulated_fields = {
        "slug": ("title",)
    }
```

We can also add filters as such:

```

    admin.py  X  models.py
book_outlet > admin.py > BookAdmin
1   from django.contrib import admin
2
3   # First we need to import our models.
4   from .models import Book
5   # Register your models here.
6
7
8   class BookAdmin(admin.ModelAdmin):
9       # readonly_fields = ("slug",)
10      prepopulated_fields = {
11          "slug": ("title",)
12      }
13      # The filter we want to use to present the
14      list_filter = ["author", "rating"]
15
16
17      admin.site.register(Book, BookAdmin)
18

```

The screenshot shows the Django admin interface for the 'Books' model. The left sidebar has a 'BOOK\_OUTLET' section with a 'Books' item. The main content area displays a list of four books with their titles and counts in parentheses. On the right, there is a 'FILTER' sidebar with two sections: 'By author' (listing 'All', 'Gigo', 'J.K Rowling', and 'Remez') and 'By rating' (listing 'All', '1', '3', '4', and '5'). The 'list\_filter' configuration in the code is reflected in these filter options.

We can also change the way we display the data as such:

```

admin.py  X  models.py

book_outlet > admin.py > BookAdmin
1   from django.contrib import admin
2
3   # First we need to import our models.
4   from .models import Book
5   # Register your models here.
6
7
8   class BookAdmin(admin.ModelAdmin):
9       # readonly_fields = ("slug",)
10      prepopulated_fields = {
11          "slug": ("title",)
12      }
13      # The filter we want to use to present the books in the admin se
14      list_filter = ("author", "rating", )
15      list_display = ["title", "author", "rating", "is_bestselling"]

```

TITLE	AUTHOR	RATING	IS BESTSELLING
Gigo's Story	Gigo	1	●
My Story	Remez	3	●
Harry Potter 1	J.K. Rowling	5	●
The Lord Of The Rings	Remez	4	●

## Understanding Relationship Types:

### Adding a one-to-many Relation & Migrations:

The screenshot shows a code editor interface with a sidebar and a main editing area.

**Left Sidebar:**

- EXPLORER
- OPEN EDITORS
- BOOK\_STORE
  - book\_outlet
    - migrations
    - templates /book\_outlet
      - base.html
      - book\_detail.html
      - index.html
    - \_\_init\_\_.py
    - admin.py
    - apps.py
    - models.py
    - tests.py
    - urls.py
    - views.py
  - book\_store
    - \_\_init\_\_.py
    - asgi.py
    - settings.py
    - urls.py
    - wsgi.py
  - db.sqlite3
  - manage.py

> OUTLINE

**Right Main Area:**

File: admin.py

```
from django.db import models
from django.forms import CharField, IntegerField
from django.core.validators import MinValueValidator, MaxValueValidator
from django.urls import reverse
from django.utils.text import slugify

# Create your models here.

class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    # Represent a table with three columns - id, title, rating

class Book(models.Model):
    # id - comes automatically, but can be modified if needed
    # by using id = models.AutoField()
    title = models.CharField(max_length=50)
    rating = models.IntegerField(validators=[MinValueValidator(1), MaxValueValidator(5)])
    author = models.ForeignKey(Author, on_delete=models.SET_NULL)
    is_bestselling = models.BooleanField(default=False)
    # Harry Potter 1 => harry-potter-1
    slug = models.SlugField(default="", null=False, blank=True, db_index=True)

    def __str__(self) -> str:
        return f"""
        Book Title: {self.title}
        ({self.rating})
        """

    def get_absolute_url(self):
```

Ln 27, Col 65   Spaces: 4   UTF-8   LF   Python   3.8.9 64-bit

We have a lot of options for "on\_delete=". Each option define what happens when an author is deleted. In other words - what happens to the book if he doesn't have an author.

## **Working with Relations in Python Code:**

```
>>> from book_outlet.models import Book, Author
>>> jkRowling = Author(first_name="J.K.", last_name="Rowling")
>>> jkRowling.save()
>>> Author.objects.all()
<QuerySet [
```

## Cross Model Queries:

```
>>> from book_outlet.models import Book, Author
>>> book_by_rowling = Book.objects.filter(author__last_name="Rowling")
>>> book_by_rowling
<QuerySet [
```

In the next example we can use that we can "chain" query's by "`__`" -> we step inside the author model and can access its fields like "last\_name":

```
>> >>> book_by_rowling = Book.objects.filter(author__last_name__contains="ling")
>> book_by_rowling
QuerySet [<Book:
    Book Title: Harry Potter 1
    (5)
    >] >
>> █
```

### Adding a one-to-one Relation:

Creating address and assigning them to author.

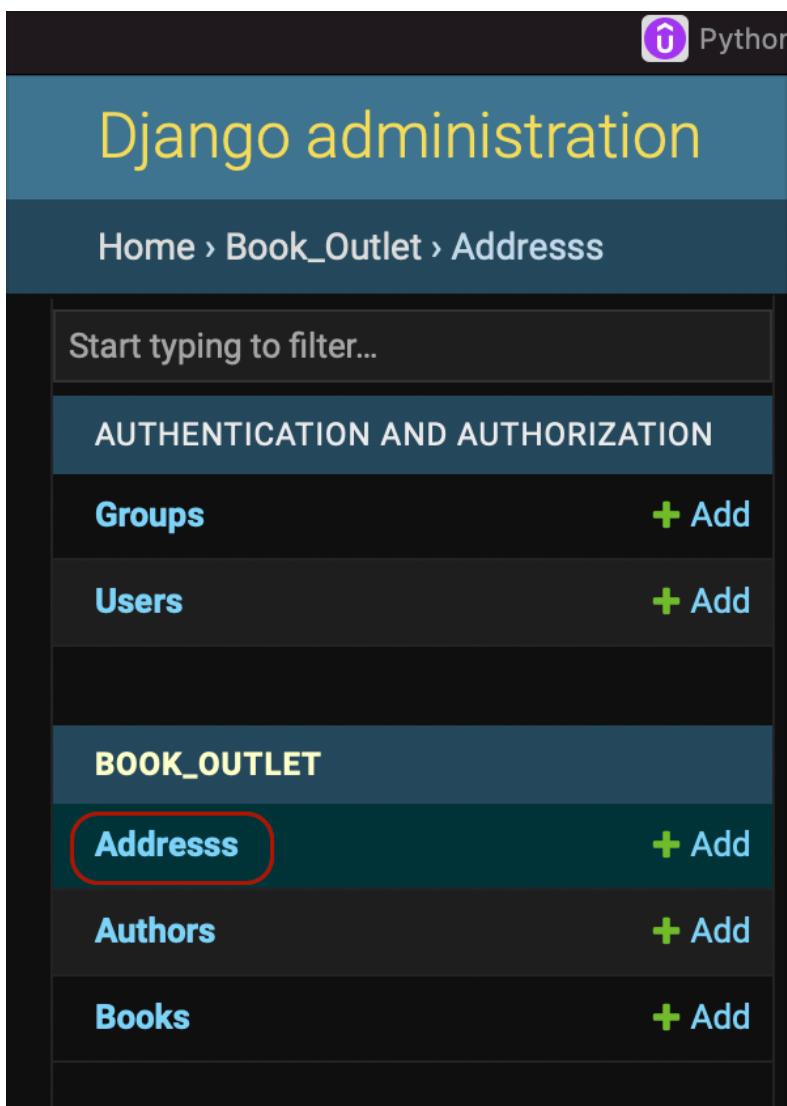
```
class Address(models.Model):
    street = models.CharField(max_length=80)
    postal_code = models.CharField(max_length=5)
    city = models.CharField(max_length=50)

class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    address = models.OneToOneField(Address, on_delete=models.CASCADE) # setup a one to one relation
```

```
>> from book_outlet.models import Author, Address, Book
>> addr1 = Address(street="some street", postal_code="some postal", city="some city")
>> addr2 = Address(street="another street", postal_code="12345", city="New York")
>> addr1.save()
>> addr2.save()
>> Author.objects.all()
<QuerySet [<Author: J.K. Rowling>]>
>> jkr = Author.objects.get(first_name="J.K.")
>> jkr.address
>> jkr.address = addr1
>> jkr.save()
>> █
```

### Change the model presentation in the admin section:

From this:



To this:

# Django administration

Home › Book\_Outlet › Address Entries

Start typing to filter...

## AUTHENTICATION AND AUTHORIZATION

Groups

+ Add

Users

+ Add

## BOOK\_OUTLET

Address Entries

+ Add

Authors

+ Add

Books

+ Add

By creating a nested loop with the name "Meta":

admin.py      models.py ●

book\_outlet > models.py > Address > Meta

```
3   from django.forms import CharField, IntegerField
4   from django.core.validators import MinValueValidator, Max
5   from django.urls import reverse
6   from django.utils.text import slugify
7
8   # Create your models here.
9
10
11  class Address(models.Model):
12      street = models.CharField(max_length=80)
13      postal_code = models.CharField(max_length=5)
14      city = models.CharField(max_length=50)
15
16      def __str__(self):
17          return f"{self.street}, {self.postal_code}, {self.city}"
18
19      class Meta:
20          # how model output should look in the admin side.
21          verbose_name_plural = "Address Entries"
22
23
```

### Setting-up many-to-many:

```
9
10     class Country(models.Model):
11         name = models.CharField(max_length=80)
12         code = models.CharField(max_length=2)
13
```

```

class Book(models.Model):
    # id - comes automatically, but can be modified if needed
    # by using id = models.AutoField()
    title = models.CharField(max_length=50)
    rating = models.IntegerField(validators=[
        MinValueValidator(1),
        MaxValueValidator(5),
    ])
    # Saving a pointer in the "author" entry to an entry in the "Author" table
    # Setting up a relation between "Book" and "Author"
    author = models.ForeignKey(Author, on_delete=models.CASCADE, null=True)
    is_bestselling = models.BooleanField(default=False)
    # Harry Potter 1 => harry-poter-1
    slug = models.SlugField(default="", null=False,
                           blank=True, db_index=True)
    published_countries = models.ManyToManyField(Country)

```

```

>>> germany = Country(name="Germany", code="DE")
>>> hp1.published_countries.add(germany)

```

### Circular Relations & Lazy Relations:

To see that go to right tab of the course and read about it.

### Fetching Posts for Starting Page:

```

models.py M          views.py      admin.

blog > views.py > starting_page
1   from datetime import date
2   from django.shortcuts import render
3   from .models import Post
4   # Create your views here.

```

### Section 10 - Forms:

#### CSRF Protection:

Django provides a csrf token to check post requests that comes in to the server.

We want to use this token in order to validate that the data we getting from users, really came from our website.

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Your Review</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
        <label for="username">Your name</label>
        <input type="text" id="username" name="username">
        <button>Send</button>
    </form>
</body>
</html>

```

## Handling Form Submission & Extracting Data:

```

from django.shortcuts import render
from django.http import HttpResponseRedirect
# Create your views here.

def review(request):
    if request.method == "POST":
        # request.POST - returns a dictionary of all the data that has been sent.
        entered_username = request.POST["username"]
        print(entered_username)
        return HttpResponseRedirect("/thank-you")
    else:
        return render(request, "reviews/review.html")

def thank_you(request):
    return render(request, "reviews/thank_you.html")

```

We can also validate our data:

```

def review(request):
    if request.method == "POST":
        # request.POST - returns a dictionary of all the data that has been sent.
        entered_username = request.POST["username"]
        if entered_username == "":
            return render(request, "reviews/review.html", {
                "has_error": True,
            })
        print(entered_username)
        return HttpResponseRedirect("/thank-you")
    else:
        return render(request, "reviews/review.html", {
            "has_error": False,
        })

```

thank\_you.html    review.html    views.py    urls.py feedback    urls.

reviews > templates > reviews > review.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Your Review</title>
8  </head>
9  <body>
10     <form method="POST">
11         {% csrf_token %}
12         {% if has_error %}
13             <p>This form is invalid - Please enter a valid user name!</p>
14         {% endif %}
15         <label for="username">Your name</label>
16         <input type="text" id="username" name="username">
17         <button>Send</button>
18     </form>
19  </body>
20  </html>

```

But its not the way - we do not check data manually.

## Using the Django Form Class:

EXPLORER    OPEN EDITORS

FEEDBACK

- feedback
  - \_\_init\_\_.py
  - asgi.py
  - settings.py
  - urls.py
  - wsgi.py
- reviews
  - migrations
  - templates/reviews
    - review.html
    - thank\_you.html
  - \_\_init\_\_.py
  - admin.py
  - apps.py
  - forms.py
  - models.py
  - tests.py
  - urls.py
  - views.py
- db.sqlite3
- manage.py

forms.py    views.py

reviews > forms.py > ...

```

1  from django import forms
2
3  # The name of the class is up to us. its a convention to end it with "Form"
4
5
6  class ReviewForm(forms.Form):
7      # Here we can define the fields the form will have. (kinda like the models classes),
8      user_name = forms.CharField()

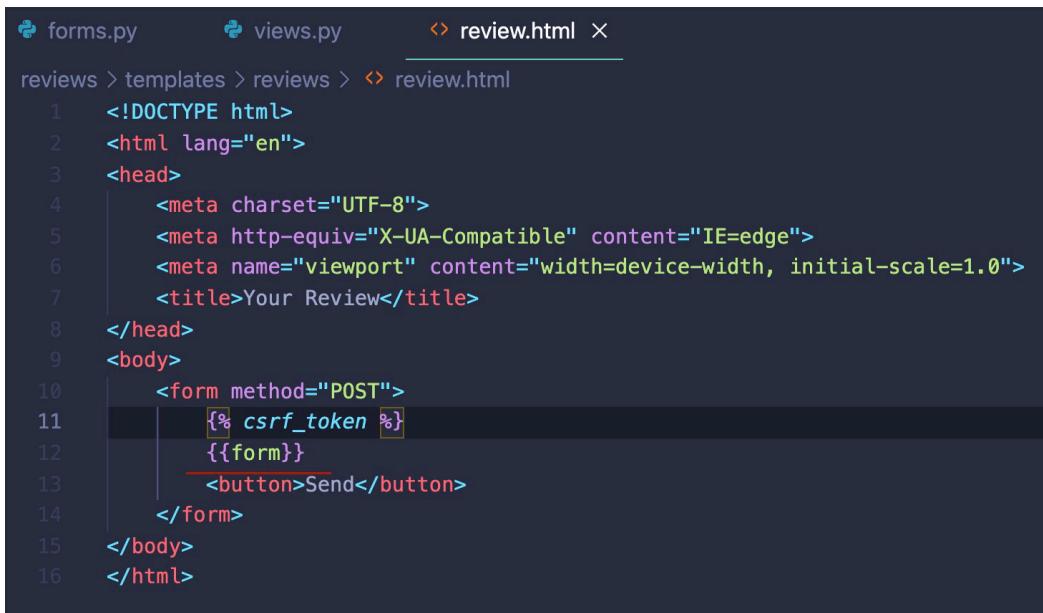
```

```

3   from .forms import ReviewForm
4   # Create your views here.
5
6
7   def review(request):
8       # if request.method == "POST":
9       #     # request.POST - returns a dictionary of all the
10      #     # entered_username = request.POST["username"]
11      #     if entered_username == "":
12      #         return render(request, "reviews/review.html"
13      #                     "has_error": True,
14      #                     })
15      #     print(entered_username)
16      #     return HttpResponseRedirect("/thank-you")
17   form = ReviewForm()
18
19   return render(request, "reviews/review.html", {
20       "form": form,
21   })
22
23
24

```

Presenting the form in the relevant template.



```

forms.py      views.py      review.html ×
reviews > templates > reviews > review.html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Your Review</title>
8  </head>
9  <body>
10     <form method="POST">
11         {% csrf_token %}
12         {{ form }}
13         <button>Send</button>
14     </form>
15  </body>
16  </html>

```

```
forms.py      views.py  X  review.html
reviews > views.py > review
1  from django.shortcuts import render
2  from django.http import HttpResponseRedirect
3  from .forms import ReviewForm
4  # Create your views here.
5
6
7  def review(request):
8      if request.method == "POST":
9          form = ReviewForm(request.POST)
10         # Validate the input, return True if ok False otherwise. also, auto fill a field name "cleaned_data"
11         if form.is_valid():
12             print(form.cleaned_data)
13             return HttpResponseRedirect("/thank-you")
14         form = ReviewForm()
15         return render(request, "reviews/review.html", {
16             "form": form,
17         })
18
```

## Customizing the Form Controls:

```
class ReviewForm(forms.Form):
    # Here we can define the fields the form will have. (kinda like the models classes).
    # label - the text besides the input (a label with text we decide)
    # max_length - the max length of data the user can enter into this field
    # error_messages - Dictionary where the keys are the error names django knows and values are the error messages
    # we want to present to the user.
    user_name = forms.CharField(max_length=20, required=True)
        label="Remez19 is the best gamer", max_length=20, error_messages={
            "required": "Dame you left the field empty!"}
```

## Customizing the Rendered HTML:

```
forms.py      views.py  X  review.html
reviews > templates > reviews > review.html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Your Review</title>
8  </head>
9  <body>
10     <form method="POST">
11         {% csrf_token %}
12         {{ form.user_name.label_tag }}
13         {{ form.user_name }}
14         {{ form.user_name.errors }}
15         <button>Send</button>
16     </form>
17 </body>
18 </html>
```

## Adding More Form Controls:

```

6   class ReviewForm(forms.Form):
7       # Here we can define the fields the form will have. (kinda like the models classes).
8       # label - the text besides the input (a label with text we decide)
9       # max_length - the max length of data the user can enter into this field
10      # error_messages - Dictionary where the keys are the error names django knows and values
11      # we want to present to the user.
12      user_name = forms.CharField(
13          label="User Name:", max_length=20, error_messages={
14              "required": "Dame you left the field empty!"
15          })
16      review_text = forms.CharField(
17          label="Your Feedback", widget=forms.Textarea, max_length=200)
18      rating| = forms.IntegerField(label="Your Rating", min_value=1, max_value=5)
19

```

Looping over the form fields:

```

forms.py      views.py      review.html X  # styles.css
reviews > templates > reviews > review.html
1  {% load static %}
2  <!DOCTYPE html>
3  <html lang="en">
4  <head>
5      <meta charset="UTF-8">
6      <meta http-equiv="X-UA-Compatible" content="IE=edge">
7      <meta name="viewport" content="width=device-width, initial-scale=1.0">
8      <title>Your Review</title>
9      <link rel="stylesheet" href="{% static 'reviews/styles.css' %}">
10     </head>
11    <body>
12        <form method="POST">
13            {% csrf_token %}
14            {% for field in form %}
15                <div class="form-control {% if field.errors%} errors{% endif %}">
16                    {{ field.label_tag }}
17                    {{ field }}
18                    {{field.errors}}
19                </div>
20            {% endfor %}
21            <button>Send</button>
22        </form>
23    </body>
24    </html>

```

Storing Form Data in a Database:

```
forms.py views.py models.py ×

reviews > models.py > ...
1  from django.db import models
2
3  # Create your models here.
4
5
6  class Review(models.Model):
7      user_name = models.CharField(max_length=100)
8      review_text = models.TextField()
9      rating = models.IntegerField()
10
```

```
def review(request):
    if request.method == "POST":
        form = ReviewForm(request.POST)
        # Validate the input, return True if ok False otherwise
        if form.is_valid():
            review = Review(
                user_name=form.cleaned_data["user_name"],
                review_text=form.cleaned_data["review_text"],
                rating=form.cleaned_data["rating"])
            review.save()
            return HttpResponseRedirect("/thank-you")
    else:
        form = ReviewForm()
    return render(request, "reviews/review.html", {
        "form": form,
    })

def thank_you(request):
    return render(request, "reviews/thank_you.html")
```

## Introducing ModelForms:

```

forms.py  X  models.py  views.py
reviews > forms.py > ReviewForm
6   # class ReviewForm(forms.Form):
7   #     # Here we can define the fields the form will have. (kinda like the models classes).
8   #     # label - the text besides the input (a label with text we decide)
9   #     # max_length - the max length of data the user can enter into this field
10  #     # error_messages - Dictionary where the keys are the error names django knows and va
11  #     # we want to present to the user.
12  #         user_name = forms.CharField(
13  #             label="User Name:", max_length=20, error_messages={
14  #                 "required": "Dame you left the field empty!"
15  #             })
16  #         review_text = forms.CharField(
17  #             label="Your Feedback", widget=forms.Textarea, max_length=200)
18  #         rating = forms.IntegerField(label="Your Rating", min_value=1, max_value=5)
19
20 class ReviewForm(forms.ModelForm):
21     class Meta:
22         model = Review
23         # The fields that we shuold present in the form based on the fields in the model.
24         fields = ["user_name", "review_text", "rating"]
25         # Alternatively if we want all the model fields in our form we can use:
26         # fields = "__all__"
27
28     pass
29

```

## Configuring the Modelform:

```

20 class ReviewForm(forms.ModelForm):
21     class Meta:
22         model = Review
23         # The fields that we should present in the form based on the fields in the model.
24         fields = ["user_name", "review_text", "rating"]
25         # Alternatively if we want all the model fields in our form we can use:
26         # fields = "__all__"
27         # we can exclude fields by using:
28         # exclude = ["field names in the model"]
29
30         # We this we can customize the labels of the field form.
31         # The keys in this dic are the var names in our model in this case "from .models import Review"
32         labels = {
33             "user_name": "Your Name:",
34             "review_text": "Your Feedback",
35             "rating": "Your Rating"
36         }
37         # Same idea as seen above
38         error_messages = {
39             "user_name": {
40                 # The model field names
41                 # This dic will have keys as the error type and values as the output in
42                 # cases this kind of an error accourd.
43                 "required": "Dame you left the field empty!",
44                 "max_length": "Please Enter a Shorter Name"
45             }
46         }
47

```

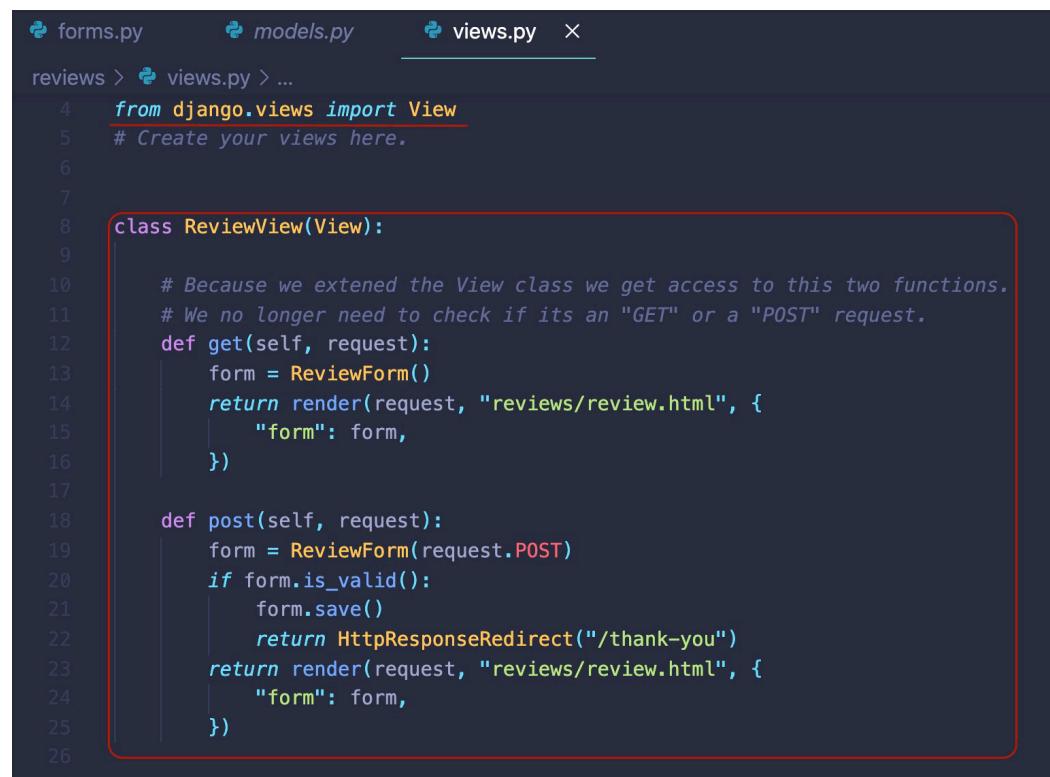
## Saving Data with a Modelform:

```

def review(request):
    if request.method == "POST":
        form = ReviewForm(request.POST)
        # Validate the input, return True if ok False otherwise
        if form.is_valid():
            form.save()
            return HttpResponseRedirect("/thank-you")
    else:
        form = ReviewForm()
    return render(request, "reviews/review.html", {
        "form": form,
    })

```

## Class Based Views:



The screenshot shows a code editor with three tabs at the top: forms.py, models.py, and views.py. The views.py tab is active. The code is as follows:

```

forms.py      models.py      views.py  X
reviews > views.py > ...
4   from django.views import View
5   # Create your views here.
6
7
8   class ReviewView(View):
9
10  # Because we extended the View class we get access to this two functions.
11  # We no longer need to check if its an "GET" or a "POST" request.
12  def get(self, request):
13      form = ReviewForm()
14      return render(request, "reviews/review.html", {
15          "form": form,
16      })
17
18  def post(self, request):
19      form = ReviewForm(request.POST)
20      if form.is_valid():
21          form.save()
22          return HttpResponseRedirect("/thank-you")
23      return render(request, "reviews/review.html", {
24          "form": form,
25      })
26

```

A red rectangular box highlights the entire class-based view implementation from line 8 to line 26.

A screenshot of a code editor showing the `urls.py` file for the `reviews` application. The file contains the following code:

```
1  from django.urls import path
2  from . import views
3
4  urlpatterns = [
5      path("", views.ReviewView.as_view()),
6      path("thank-you", views.thank_you)
7 ]
8
```