# Django

Django is a back-end server side web framework.

Django is free, open source and written in Python.

Django makes it easier to build web pages using Python.

Django emphasizes reusability of components, also referred to as DRY (Don't Repeat Yourself), and comes with ready-to-use features like login system, database connection and CRUD operations (Create Read Update Delete).

# Framework

A software framework is a structure that you can use to build software. It acts as a foundation so you don't have to deal with creating unnecessary extra logic from scratch.

A framework is similar to a template in that you can modify it and add certain features and higher functionalities to create a complex and broad project that many people can use.

# How does Django Work?

Django follows the MVT design pattern (Model View Template).

* Model - The data you want to present, usually data from a database.
* View - A request handler that returns the relevant template and content - based on the request from the user.
* Template - A text file (like an HTML file) containing the layout of the web page, with logic on how to display the data.



## Model

The model provides data from the database.

In Django, the data is delivered as an Object Relational Mapping (ORM), which is a technique designed to make it easier to work with databases.

Django, with ORM, makes it easier to communicate with the database, without having to write complex SQL statements.

The models are usually located in a file called mode ls.py.

## View

A view is a function or method that takes http requests as arguments, imports the relevant model(s), and finds out what data to send to the template, and returns the final result.

The views are usually located in a file called views.py.

## Template

A template is a file where you describe how the result should be represented.

Django tags to add logic:

<h1>My Homepage</h1>

<p>My name is {{ firstname }}.</p>

The templates of an application is located in a folder named templates.

## URLs

Django also provides a way to navigate around the different pages in a website.

When a user requests a URL, Django decides which view it will send it to.

This is done in a file called urls.py.

## So, What is Going On?

When you have installed Django and created your first Django web application, and the browser requests the URL, this is basically what happens:

* Django receives the URL, checks the urls.py file, and calls the view that matches the URL.
* The view, located in views.py, checks for relevant models.
* The models are imported from the models.py file.
* The view then sends the data to a specified template in the template folder.
* The template contains HTML and Django tags, and with the data it returns finished HTML content back to the browser.

# Virtual Environment

Virtual environments are useful for managing dependencies and isolating projects from each other, ensuring that each project can have its own specific set of dependencies without interfering with other projects.

py -m venv myworld

myworld  
  Include  
  Lib  
  Scripts  
  pyvenv.cfg

Then you have to activate the environment, by typing this command:

myworld\Scripts\activate.bat

Once the environment is activated, you will see this result in the command prompt:

(myworld) C:\Users\Your Name>

**Note:** You must activate the virtual environment every time you open the command prompt to work on your project.

# Install Django

Now, that we have created a virtual environment, we are ready to install Django.

**Note:** Remember to install Django while you are in the virtual environment!

Django is installed using pip, with this command:

(myworld) C:\Users\Y e>py -m pip install Django

**Check Django Version**

Django-admin –version

# Create Project

**My First Project**

Once you have come up with a suitable name for your Django project, like mine: my\_tennis\_club, navigate to where in the file system you want to store the code (in the virtual environment), I will navigate to the myworld folder, and run this command in the command prompt:

django-admin startproject my\_tennis\_club

Django creates a my\_tennis\_club folder on my computer, with this content:

my\_tennis\_club  
    manage.py  
    my\_tennis\_club/  
        \_\_init\_\_.py  
        asgi.py  
        settings.py  
        urls.py  
        wsgi.py

**Run the Django Project**

Now that you have a Django project, you can run it, and see what it looks like in a browser.

Navigate to the /my\_tennis\_club folder and execute this command in the command prompt:

py manage.py runserver

# Create App

**What is an App?**

An app is a web application that has a specific meaning in your project, like a home page, a contact form, or a members database.

I will name my app members.

Start by navigating to the selected location where you want to store the app, in my case the my\_tennis\_club folder, and run the command below.

If the server is still running, and you are not able to write commands, press [CTRL] [BREAK], or [CTRL] [C] to stop the server and you should be back in the virtual environment.

py manage.py startapp members

# Views

Django views are Python functions that takes http requests and returns http response, like HTML documents.

A web page that uses Django is full of views with different tasks and missions.

Views are usually put in a file called views.py located on your app's folder.

There is a views.py in your members folder that looks like this:

my\_tennis\_club/members/views.py:

from django.shortcuts import render

Find it and open it, and replace the content with this:

my\_tennis\_club/members/views.py:

from django.shortcuts import render

from django.http import HttpResponse

def members(request):

return HttpResponse("Hello world!")

But how can we execute the view? Well, we must call the view via a URL.

# URLs

Create a file named urls.py in the same folder as the views.py file, and type this code in it:

my\_tennis\_club/members/urls.py:

from django.urls import path

from . import views

urlpatterns = [

path('members/', views.members, name='members'),

]

The urls.py file you just created is specific for the members application. We have to do some routing in the root directory my\_tennis\_club as well. This may seem complicated, but for now, just follow the instructions below.

There is a file called urls.py on the my\_tennis\_club folder, open that file and add the include module in the import statement, and also add a path() function in the urlpatterns[] list, with arguments that will route users that comes in via 127.0.0.1:8000/.

Then your file will look like this:

my\_tennis\_club/my\_tennis\_club/urls.py:

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

path('', include('members.urls')),

path('admin/', admin.site.urls),

]

If the server is not running, navigate to the /my\_tennis\_club folder and execute this command in the command prompt:

py manage.py runserver

# Templates

Create a templates folder inside the members folder, and create a HTML file named myfirst.html.

The file structure should be like this:

my\_tennis\_club  
    manage.py  
    my\_tennis\_club/  
    members/  
        templates/  
            myfirst.html

Open the HTML file and insert the following:

my\_tennis\_club/members/templates/myfirst.html:

<!DOCTYPE html>

<html> <body>

<h1>Hello World!</h1>

<p>Welcome to my first Django project!</p>

</body> </html>

**Modify the View**

Open the views.py file and replace the members view with this:

my\_tennis\_club/members/views.py:

from django.http import HttpResponse

from django.template import loader

def members(request):

template = loader.get\_template('myfirst.html')

return HttpResponse(template.render())

**Change Settings**

To be able to work with more complicated stuff than "Hello World!", We have to tell Django that a new app is created.

This is done in the settings.py file in the my\_tennis\_club folder.

Look up the INSTALLED\_APPS[] list and add the members app like this:

my\_tennis\_club/my\_tennis\_club/settings.py:

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'members'

]

Then run this command:

py manage.py migrate

Start the server by navigating to the /my\_tennis\_club folder and execute this command:

py manage.py runserver

# Model

A Django model is a table in your database.

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you’re storing. Generally, each model maps to a single database table.

The basics:

* Each model is a Python class that subclasses **[django.db.models.Model](https://docs.djangoproject.com/en/4.2/ref/models/instances/" \l "django.db.models.Model" \o "django.db.models.Model)**.
* Each attribute of the model represents a database field.
* With all of this, Django gives you an automatically-generated database-access API

Once you’ve created your [data models](https://docs.djangoproject.com/en/4.2/topics/db/models/), Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects.

## Create Table (Model)

To create a model, navigate to the models.py file in the app folder.

from django.db import models

class Employee(models.Model):

    name=models.CharField(max\_length=40)

    lname=models.CharField(max\_length=40)

    email=models.EmailField(max\_length=40)

    age=models.IntegerField()

    image=models.FileField(upload\_to='')

**SQLite Database**

When we created the Django project, we got an empty SQLite database.

It was created in  root folder, and has the filename db.sqlite3.

By default, all Models created in the Django project will be created as tables in this database.

**once model is created , you have to run two commands.**

## makemigration , migration , sqlmigrate

**makemigrations**

makemigration basically generates the SQL commands for preinstalled apps (which can be viewed in installed apps in settings.py) and your newly created apps' model which you add in installed apps.

It does not execute those commands in your database file. So tables doesn't created after makemigrations.

python manage.py makemigrations <app>: Create the migrations (generate the SQL commands).

After applying makemigrations you can see those SQL commands with**sqlmigrate** which shows all the SQL commands which has been generated by makemigrations.

**migrate**

migrate executes those SQL commands in database file.

So after executing migrate all the tables of your installed apps are created in your database file.

You can conform this by installing [sqlite browser](http://sqlitebrowser.org/) and opening db.sqlite3 you can see all the tables appears in the database file after executing migrate command.

python manage.py migrate: Run the migrations (execute the SQL commands).

**sqlmigrate**

As a side-note: you can view the SQL statement that were executed from the migration above. All you have to do is to run this command, with the migration number:

py manage.py sqlmigrate members 0001

## from django.db import models (explain)

**from**: This is a Python keyword used to import modules, classes, or functions from other modules.

**django.db:** This part specifies that we are importing something from the db

sub-module of the django package.

**import models:** Here, we are specifically importing the **models module** from Django's db sub-module. This module contains classes and tools for defining and managing database models.

In the context of Django, a "model" refers to a Python class that represents a database table. Each attribute of the class corresponds to a field in the table, and methods can define behaviors or interactions with the data.

from django.db import models

class Person(models.Model):

first\_name = models.CharField(max\_length=30)

last\_name = models.CharField(max\_length=30)

birth\_date = models.DateField()

In this example, we've defined a **Person** model with three fields:

**first\_name, last\_name,** and **birth\_date.**

The **models.CharField** and **models.DateField** are classes provided by Django's **models** module to define the types of these fields.

The **max\_length** argument specifies the maximum length of characters allowed for a **CharField.**

With this model definition ,Django can automatically generate the corresponding database table and provide methods to perform CRUD (Create, Read, Update, Delete) operations on the table rows.

**After creating model you can see table in admin panel .**

1. You have to create a superuser 🡪py manage.py createsuperuser
2. You have to add model class in apps 🡪 admin.py

from django.contrib import admin

from .models import Member

# Register your models here.

admin.site.register(Member) 🡪( Member =this is a class in model.py)

. (dot means same folder)

1. Go to server urls 🡪 <http://127.0.0.1:8000/admin/> and login
2. Now you can insert data .

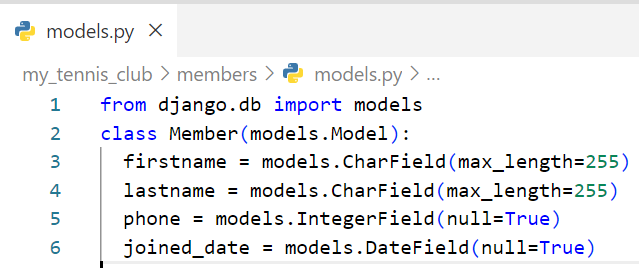
## Create User

To be able to log into the admin application, we need to create a user.

This is done by typing this command in the command view:

py manage.py createsuperuser

## Include Model In Admin Interface

You made a model in model.py

**Code explain**

**from django.db import models**

This line imports the necessary module from Django that provides classes and tools for working with databases and creating models.

**class Member(models.Model):** (models.Model =models module se Model classs useing)

Here, a new Python class named Member is defined, and it inherits from the models.Model class provided by Django.

This indicates that Member is a Django model and will be mapped to a database table.

The models module provides classes like CharField, IntegerField, DateField, and more, which you can use to define the structure and types of data that your application will store in the database.

This module is a crucial part of Django's Object-Relational Mapping (ORM) system, which provides an abstraction layer for working with databases in a more Pythonic way.

**After model is ready you have to add model in admin.py**

**1. `from django.contrib import admin`:**

This imports the `admin` module from Django's `contrib` package, which contains classes and functions related to the admin interface.

**2. `from .models import Member`:**

This imports the `Member` model from the current package (module), assuming that there's a `Member` model defined in the same Django app.

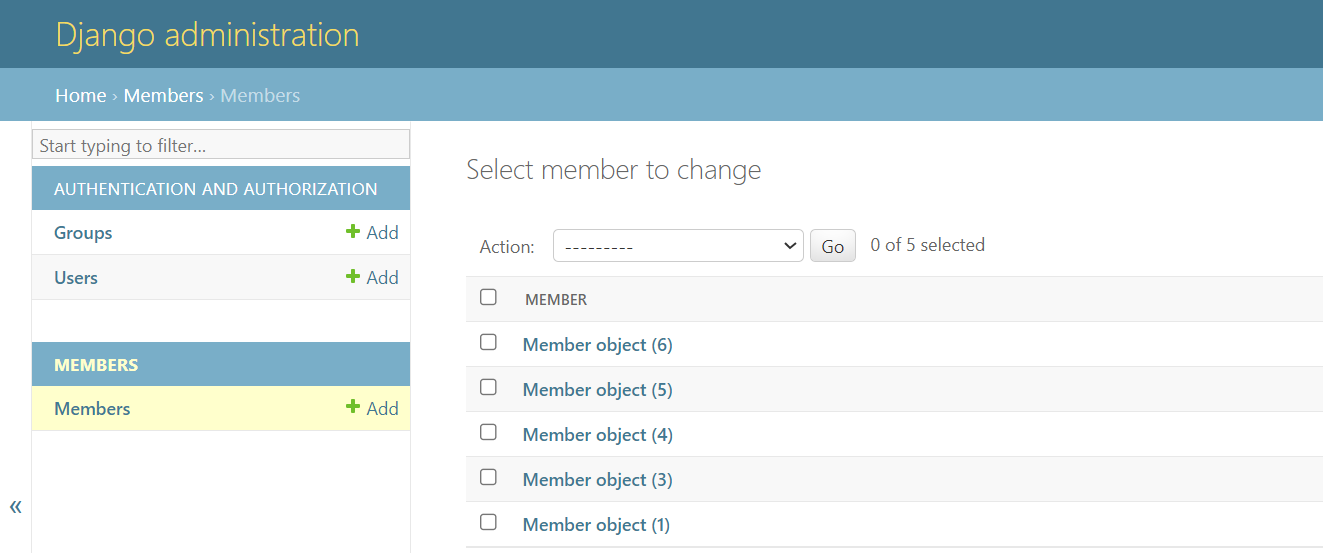
**3. `admin.site.register(Member)`:**

This line registers the `Member` model with the Django admin interface. By doing this, you enable the admin site to manage and display data from the `Member` model.

**Or you can also do this through python shell .**

## using \_\_str\_\_() function and list\_details in model.py Make the List Display More Reader-Friendly

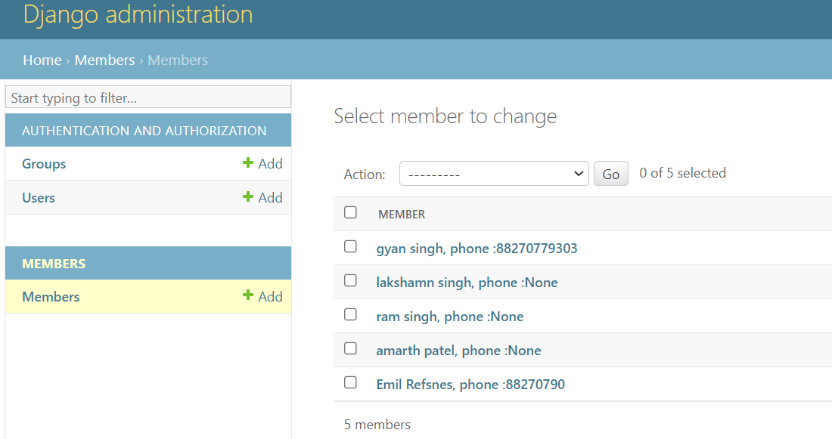
When you display a Model as a list, Django displays each record as the string representation of the record object, which in our case is "Member object (1)", "Member object(2)" etc.:



To change this to a more reader-friendly format, we have two choices:

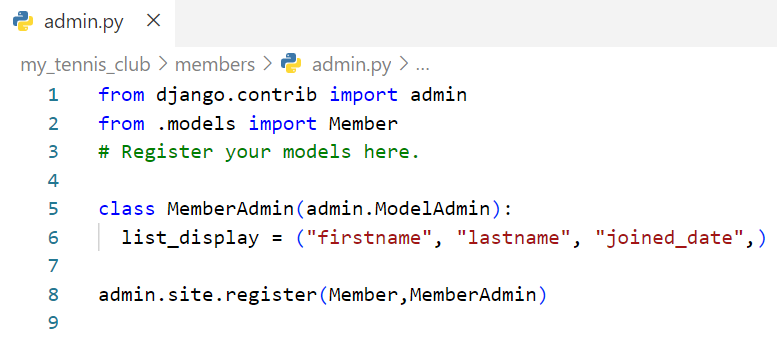
1. Change the string representation function, \_\_str\_\_() of the Member Model
2. Set the list\_details property of the Member Model

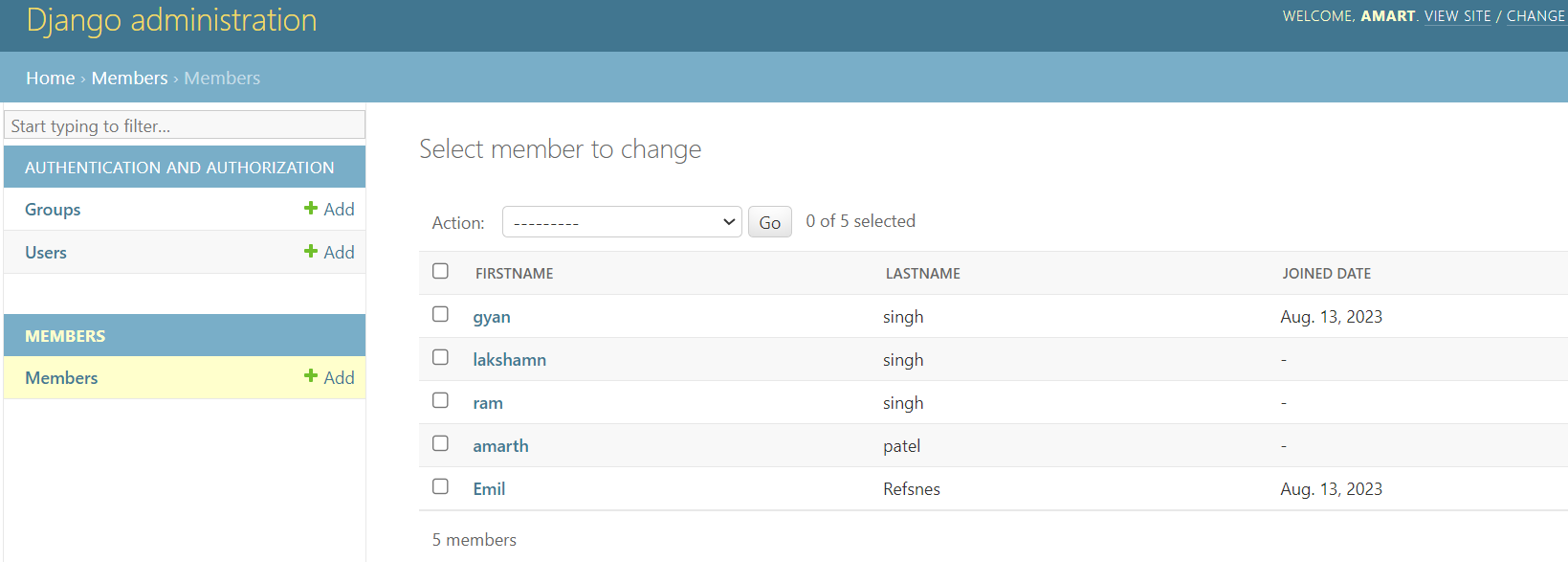
**To change the string representation, we have to define the \_\_str\_\_() function of the Member Model in models.py, like this:**

****

**Set list\_display**

We can control the fields to display by specifying them in in a list\_display property in the admin.py file.

****First create a MemberAdmin() class and specify the list\_display tuple, like this:

****

# Insert Data

**Add Records**

We will use the Python interpreter (Python shell) to add some members to it.

To open a Python shell, type this command:

py manage.py shell

Now we are in the shell, the result should be something like this:

Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
(InteractiveConsole)  
>>>

At the bottom, after the three >>> write the following:

>>> from members.models import Member

Hit [enter] and write this to look at the empty Member table:

>>> Member.objects.all()

This should give you an empty QuerySet object, like this:

<QuerySet []>

A QuerySet is a collection of data from a database.

Add a record to the table, by executing these two lines:

>>> member = Member(firstname='Emil', lastname='Refsnes')  
>>> member.save()

Execute this command to see if the Member table got a member:

>>> Member.objects.all().values()

Hopefully, the result will look like this:

<QuerySet [{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes'}]>

**Add Multiple Records**

You can add multiple records by making a list of Member objects, and execute .save() on each entry:

>>> member1 = Member(firstname='Tobias', lastname='Refsnes')  
>>> member2 = Member(firstname='Linus', lastname='Refsnes')  
>>> member3 = Member(firstname='Lene', lastname='Refsnes')  
>>> member4 = Member(firstname='Stale', lastname='Refsnes')  
>>> member5 = Member(firstname='Jane', lastname='Doe')  
>>> members\_list = [member1, member2, member3, member4, member5]  
>>> for x in members\_list:  
>>>   x.save()

Now there are 6 members in the Member table:

>>> Member.objects.all().values()  
<QuerySet [{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes'},  
{'id': 2, 'firstname': 'Tobias', 'lastname': 'Refsnes'},  
{'id': 3, 'firstname': 'Linus', 'lastname': 'Refsnes'},  
{'id': 4, 'firstname': 'Lene', 'lastname': 'Refsnes'},  
{'id': 5, 'firstname': 'Stale', 'lastname': 'Refsnes'},  
{'id': 6, 'firstname': 'Jane', 'lastname': 'Doe'}]>

# Update Records

To update records that are already in the database, we first have to get the record we want to update:

>>> from members.models import Member  
>>> x = Member.objects.all()[4]

x will now represent the member at index 4, which is "Stale Refsnes", but to make sure, let us see if that is correct:

This should give you this result:

'Stale'

Now we can change the values of this record:

>>> x.firstname = "Stalikken"  
>>> x.save()

Execute this command to see if the Member table got updated:

>>> Member.objects.all().values()

Hopefully, the result will look like this:

<QuerySet [{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes'},

{'id': 2, 'firstname': 'Tobias', 'lastname': 'Refsnes'},

{'id': 3, 'firstname': 'Linus', 'lastname': 'Refsnes'},

{'id': 4, 'firstname': 'Lene', 'lastname': 'Refsnes'},

{'id': 5, 'firstname': 'Stalikken', 'lastname': 'Refsnes'},

{'id': 6, 'firstname': 'Jane', 'lastname': 'Doe'}]>

# Delete Records

To delete a record in a table, start by getting the record you want to delete:

>>> from members.models import Member

>>> x = Member.objects.all()[5]

>>> x.delete()

The result will be:

(1, {'members.Member': 1})

Which tells us how many items were deleted, and from which Model.

If we look at the Member Model, we can see that 'Jane Doe' is removed from the Model:

>>> Member.objects.all().values()

<QuerySet [{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes'},

{'id': 2, 'firstname': 'Tobias', 'lastname': 'Refsnes'},

{'id': 3, 'firstname': 'Linus', 'lastname': 'Refsnes'},

{'id': 4, 'firstname': 'Lene', 'lastname': 'Refsnes'},

{'id': 5, 'firstname': 'Stalikken', 'lastname': 'Refsnes'}]>

# Update Model

from django.db import models

class Member(models.Model):

firstname = models.CharField(max\_length=255)

lastname = models.CharField(max\_length=255)

phone = models.IntegerField()

joined\_date = models.DateField()

As you can see, we want to add phone and joined\_date to our Member Model.

This is a change in the Model's structure, and therefor we have to make a migration to tell Django that it has to update the database:

py manage.py makemigrations members

Which, in my case, will result in a prompt, because I try to add fields that are not allowed to be null, to a table that already contains records.

As you can see, Django asks if I want to provide the fields with a specific value, or if I want to stop the migration and fix it in the model:

py manage.py makemigrations members

You are trying to add a non-nullable field 'joined\_date' to members without a default; we can't do that (the database needs something to populate existing rows).

Please select a fix:

1) Provide a one-off default now (will be set on all existing rows with a null value for this column)

2) Quit, and let me add a default in models.py

Select an option:

I will select option 2, and open the models.py file again and allow NULL values for the two new fields:

from django.db import models

class Member(models.Model):

firstname = models.CharField(max\_length=255)

lastname = models.CharField(max\_length=255)

phone = models.IntegerField(null=True)

joined\_date = models.DateField(null=True)

And make the migration once again:

py manage.py makemigrations members

Which will result in this:

Migrations for 'members':

members\migrations\0002\_member\_joined\_date\_member\_phone.py

- Add field joined\_date to member

- Add field phone to member

Run the migrate command:

py manage.py migrate

Which will result in this output:

Operations to perform:

Apply all migrations: admin, auth, contenttypes, members, sessions

Running migrations:

Applying members.0002\_member\_joined\_date\_member\_phone... OK

**Insert Data**

We can insert data to the two new fields with the same approach as we did in the Update Data chapter:

>>> from members.models import Member

>>> x = Member.objects.all()[0]

>>> x.phone = 5551234

>>> x.joined\_date = '2022-01-05'

>>> x.save()

This will insert a phone number and a date in the Member Model, at least for the first record, the four remaining records will for now be left empty.

Execute this command to see if the Member table got updated:

>>> Member.objects.all().values()

The result should look like this:

<QuerySet [

{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes', 'phone': 5551234, 'joined\_date': datetime.date(2022, 1, 5)},

{'id': 2, 'firstname': 'Tobias', 'lastname': 'Refsnes', 'phone': None, 'joined\_date': None},

{'id': 3, 'firstname': 'Linus', 'lastname': 'Refsnes', 'phone': None, 'joined\_date': None},

{'id': 4, 'firstname': 'Lene', 'lastname': 'Refsnes', 'phone': None, 'joined\_date': None},

{'id': 5, 'firstname': 'Stalikken', 'lastname': 'Refsnes', 'phone': None, 'joined\_date': None}]>

# Display Data in template

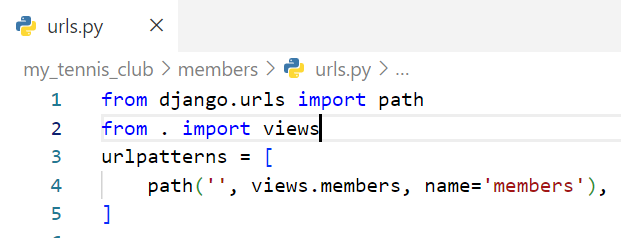
## Create Template & view

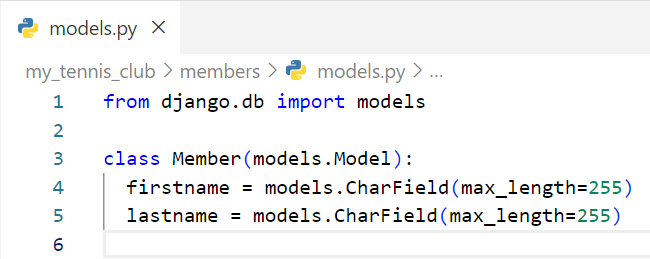
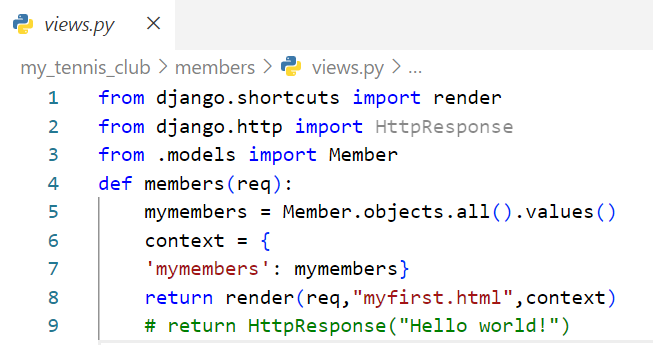
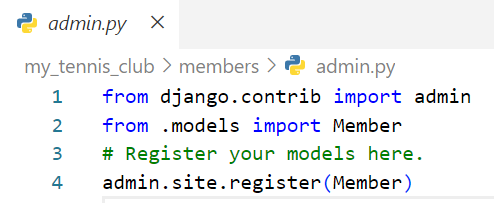
We made the following :

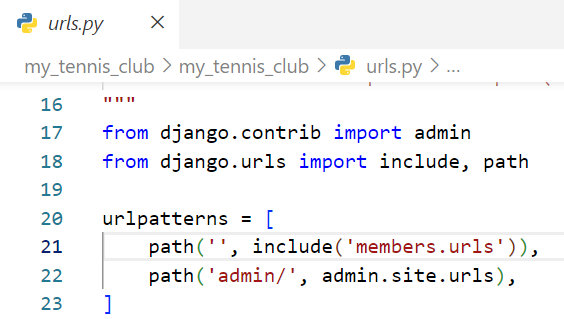
project name 🡪 my\_tennis\_club

App name 🡪members

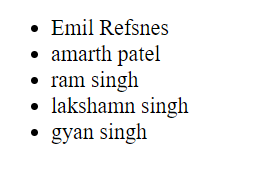
Template 🡪 myfirst.html

Model.py 🡪 Member





Output :



Expaining code :

**Import Statements:**

**from django.shortcuts import render:**

This imports the `render` function from the `django.shortcuts` module. The `render` function is used to render an HTML template with context data and return an `HttpResponse` object.

**from django.http import HttpResponse:**

This imports the `HttpResponse` class from the `django.http` module. `HttpResponse` is used to return HTTP responses from views.

**from .models import Member:**

This imports the `Member` model from the current package (module) where this code resides. It assumes that there's a `Member` model defined in the same Django app.

**View Function:**

def members(req):

    mymembers = Member.objects.all().values()

    context = {

    'mymembers': mymembers}

    return render(req,"myfirst.html",context)

The code defines a view function named `members` that takes a `req` parameter (which represents an HTTP request).

**Querying the Database:**

**mymembers = Member.objects.all().values():**

This line queries the database using the `Member` model.

It retrieves all records from the `Member` table.

**all()** method retrieves all objects (records) from the `Member` model

**mymembers = Member.objects.all()**

**print(mymembers)**

**output:**

**<QuerySet [<Member: Member object (1)>, <Member: Member object (3)>, <Member: Member object (4)>, <Member: Member object (5)>, <Member: Member object (6)>]>**

**It return all object(record) in QuerySet.**

values() returns a QuerySet containing dictionaries with the field names as keys and corresponding values.

**mymembers = Member.objects.all().values()**

**print(mymembers)**

**OUTPUT:**

<QuerySet [

{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes', 'phone': 88270790, 'joined\_date': datetime.date(2023, 8, 13)},

{'id': 3, 'firstname': 'amarth', 'lastname': 'patel', 'phone': None, 'joined\_date': None},

{'id': 4, 'firstname': 'ram', 'lastname': 'singh', 'phone': None, 'joined\_date': None},

{'id': 5, 'firstname': 'lakshamn', 'lastname': 'singh', 'phone': None, 'joined\_date': None},

{'id': 6, 'firstname': 'gyan', 'lastname': 'singh', 'phone': 88270779303, 'joined\_date': datetime.date(2023, 8, 13)}

]>

**Context Dictionary:**

**context = {'mymembers': mymembers**}:

This creates a dictionary named `context` with a single key-value pair.

The key is `'mymembers'`, and the value is the QuerySet of dictionaries obtained from the database query.

**context = {'mymembers': mymembers}**

**print(context)**

**output:**

{'mymembers' : <QuerySet [{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes', 'phone': 88270790, 'joined\_date': datetime.date(2023, 8, 13)}, {'id': 3, 'firstname': 'amarth', 'lastname': 'patel', 'phone': None, 'joined\_date': None}, {'id': 4, 'firstname': 'ram', 'lastname': 'singh', 'phone': None, 'joined\_date': None}, {'id': 5, 'firstname': 'lakshamn', 'lastname': 'singh', 'phone': None, 'joined\_date': None}, {'id': 6, 'firstname': 'gyan', 'lastname': 'singh', 'phone': 88270779303, 'joined\_date': datetime.date(2023, 8, 13)}]>}

**Rendering the Template:**

**return render(req, "myfirst.html", context):**

This line uses the `render` function to render an HTML template named `"myfirst.html"` with the provided `context` data.

The `req` parameter is the incoming HTTP request.

The function returns an `HttpResponse` object containing the rendered HTML content.

## Add Link to Details

**Details Template**

The next step in our web page will be to add a Details page, where we can list more details about a specific member.

Model.py updated add phone and joined\_date (you can see in Update Model)

from django.db import models

class Member(models.Model):

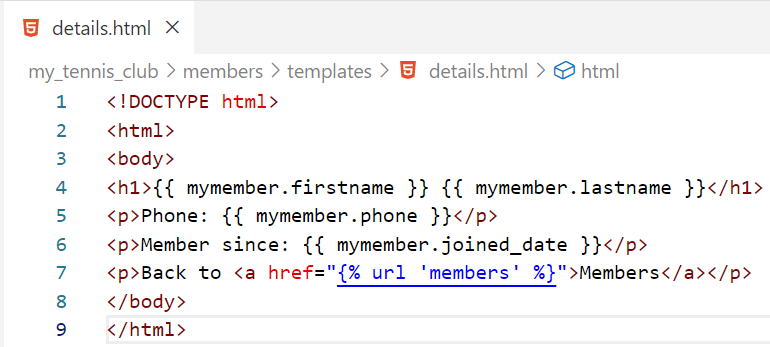
firstname = models.CharField(max\_length=255)

lastname = models.CharField(max\_length=255)

phone = models.IntegerField(null=True)

joined\_date = models.DateField(null=True)

Start by creating a new template called details.html:

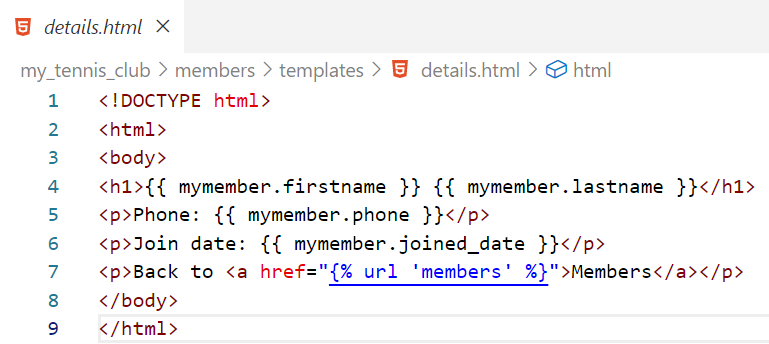


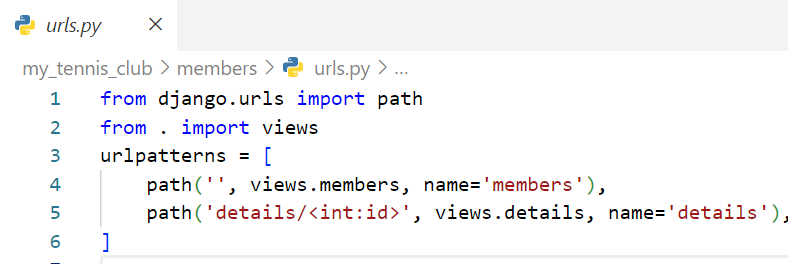
Now we update our myfirst.html page , when we link on user name through anchor tag , when we click on link it will go to details page .

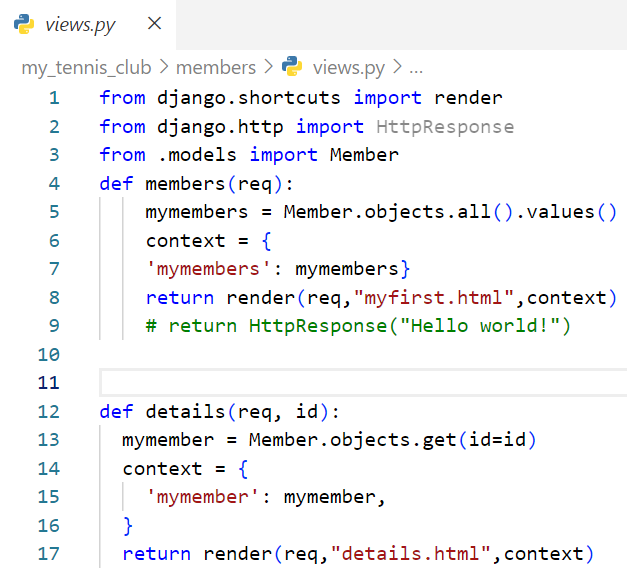
**Add Link in all-members Template**

 should be clickable, and take you to the details page with the ID of the member you clicked on:

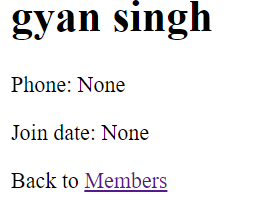
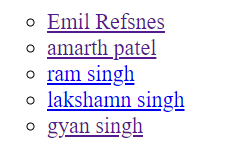








when we click on this links We go to details page



Explaining Code :

{% for x in mymembers %}

    <li><a href="details/{{ x.id }}">{{ x.firstname }} {{ x.lastname }}</a></li>

  {% endfor %}

**<a href="details/{{ x.id }}">**:

This creates a hyperlink (**<a>**) where the **href** attribute is constructed dynamically using the member's **id**.

This will likely generate a link to a URL that provides detailed information about the member with the specified **id**.

**app / urls.py:**

**path('details/<int:id>', views.details, name='details'):**

**path:** This is a function provided by Django's URL routing system for creating URL patterns.

**'details/<int:id>':** This is the URL pattern string. In this case, it includes a path segment named <int:id> which indicates that an integer value will be captured from the URL and passed as an argument to the associated view function.

def details(req, id):

  mymember = Member.objects.get(id=id) (id(object jo save h uski id)= id(jo as a argument aai hai) )

  context = {

    'mymember': mymember,

  }

  return render(req,"details.html",context)

**def details(req, id):**

This details function takes two parameters: req (representing the incoming HTTP request) and id (representing the unique identifier of a member).

**mymember = Member.objects.get(id=id):**

This line queries the database using the Member model to retrieve a single member with the specified id.

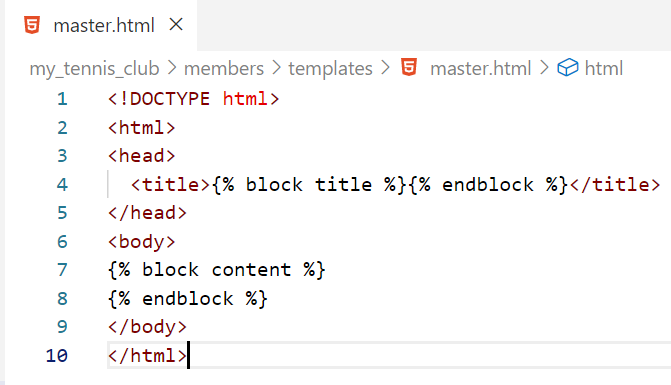
The .get() method is used to retrieve a single object that matches the given condition.

## Add Master Template

we created two templates, details.html and myfirst.html

The templates have a set of HTML code that are the same for both templates.

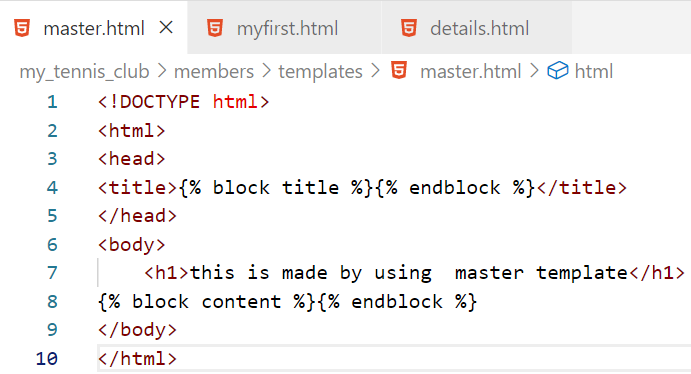
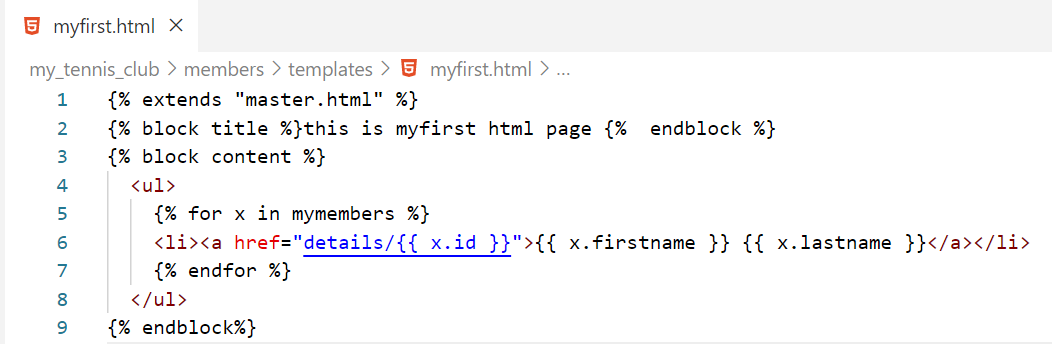
Django provides a way of making a "parent template" that you can include in all pages to do the stuff that is the same in all pages.

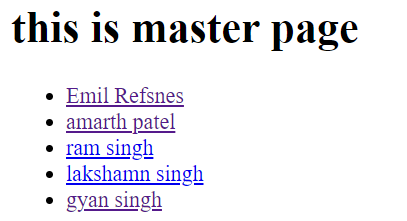
Start by creating a template called master.html, with all the necessary HTML elements:

Do you see Django block Tag inside the <title> element, and the <body> element?

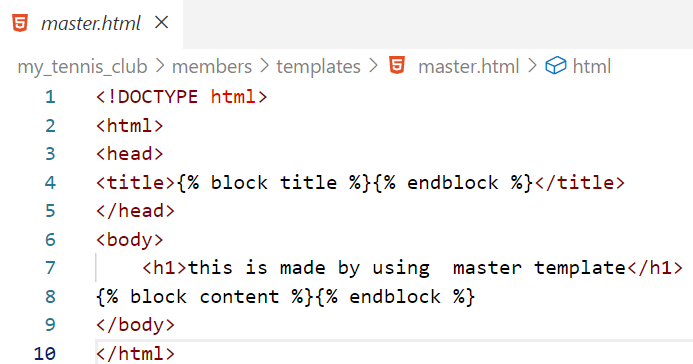
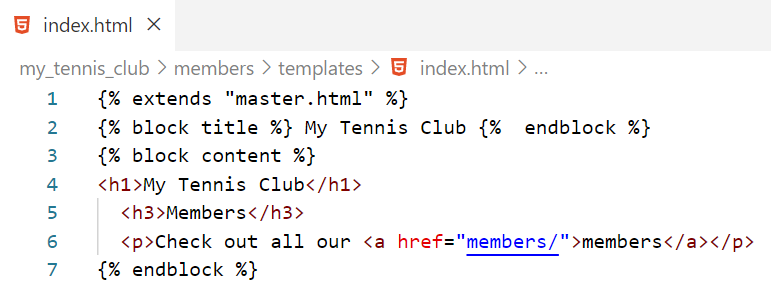
They are placeholders, telling Django to replace this block with content from other sources.

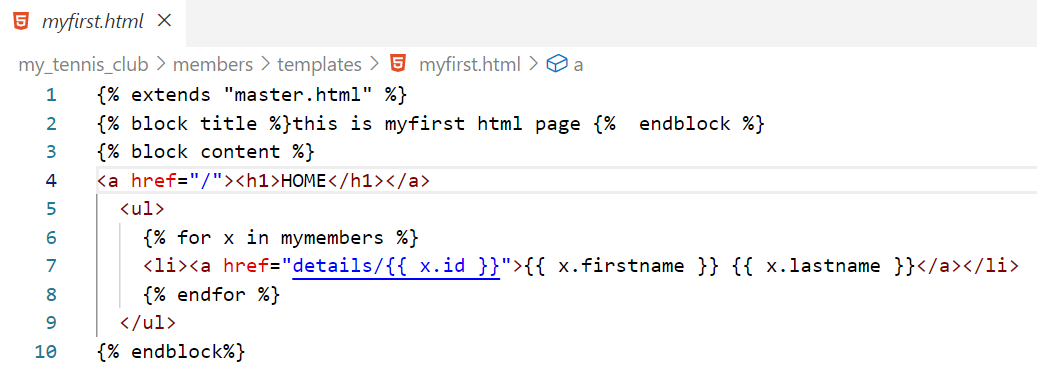
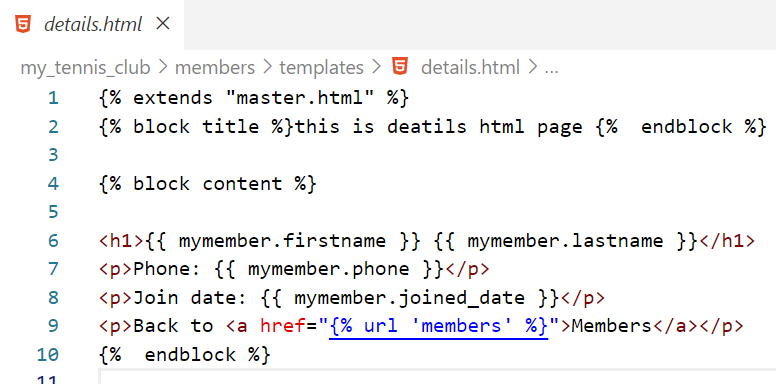
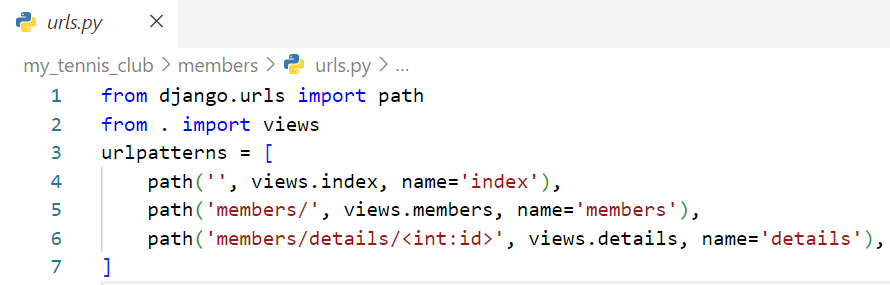
Now the two templates (details.html and myfirst.html) can use this master.html template.

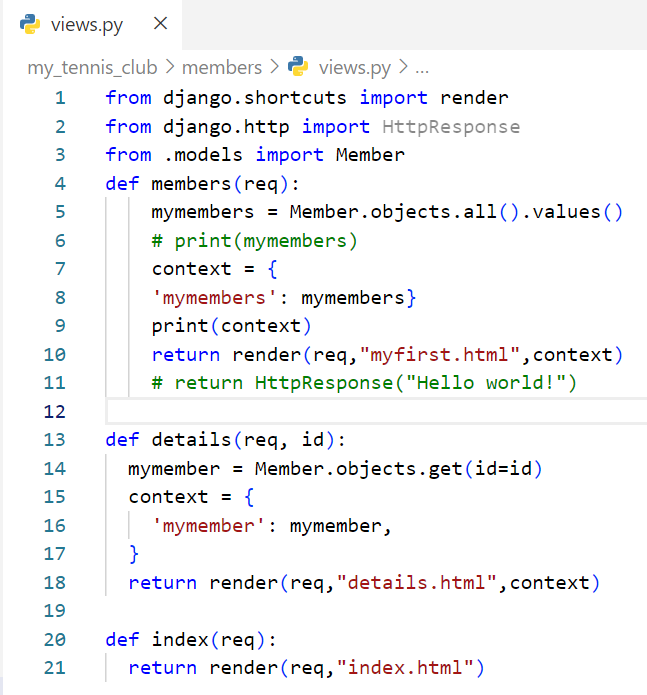
This is done by including the master template with the {% extends %} tag, and inserting a title block and a content block:



## Add Main Index Page







## 404 (page not found)

VISIT W3SCHOOL SITE FOR THIS

## Test View

When testing different aspects of Django, it can be a good idea to have somewhere to test code without destroying the main project.

# Django Admin

Django Admin is a really great tool in Django, it is actually a CRUD\* user interface of all your models!

\*CRUD stands for Create Read Update Delete.

It is free and comes ready-to-use with Django:

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

path('', include('members.urls')),

path('admin/', admin.site.urls),]

The urlpatterns[] list takes requests going to admin/ and sends them to admin.site.urls, which is part of a built-in application that comes with Django, and contains a lot of functionality and user interfaces.

# Django Variable

## Template varibable

{{ }} but we gernally not you this we use data from model class

{% with firstname="Tobias" %}

<h1>Hello {{ firstname }}, how are you?</h1>

{% endwith %}

{% %} Django tag

<ul>

{% for x in mymembers %}

<li>{{ x.firstname }}</li>

{% endfor %}

</ul>

## If elif else

|  |  |
| --- | --- |
| from django.template import loader  def testing(request):  context = {'greeting': 3}  return render(req,”main.html”,context)  output:  Goodbye | <!DOCTYPE html>  <html> <body>  {% if greeting == 1 %}  <h1>Hello</h1>  {% elif greeting == 2 %}  <h1>Welcome</h1>  {% else %}  <h1>Goodbye</h1>  {% endif %} }  </body> </html> |

## And

Returns True if both statements are true

|  |  |
| --- | --- |
| from django.template import loader  def testing(request):  context = {'greeting': 3}  return render(req,”main.html”,context)  output:  Hello | <!DOCTYPE html>  <html> <body>  {% if greeting == 1 and greeting==1 %}  <h1>Hello</h1>    {% endif %} }  </body> </html> |

## or :

Returns True if one statements are true

|  |  |
| --- | --- |
| from django.template import loader  def testing(request):  context = {'greeting': 3}  return render(req,”main.html”,context)  output:  Hello | <!DOCTYPE html>  <html> <body>  {% if greeting == 1 or greeting==2 %}  <h1>Hello</h1>    {% endif %} }  </body> </html> |

## And , or

|  |  |
| --- | --- |
| <!DOCTYPE html>  <html>  <body>  {% if greeting == 1 and day == "Friday" or greeting == 5 %}  <h1>Hello Weekend!</h1>  {% endif %}  <p>In views.py you can see what the variables look like.</p>  </body>  </html>  Output :  Hello Weekend | from django.http import HttpResponse  from django.template import loader  def testing(request):  template = loader.get\_template('template.html')  context = {  'greeting': 5,  'day': 'Friday',  }  return HttpResponse(template.render(context, request)) |

## in

To check if a certain item is present in an object.

{% if 'Banana' in fruits %}

<h1>Hello</h1>

{% else %}

<h1>Goodbye</h1>

{% endif %}

## not in

To check if a certain item is not present in an object.

{% if 'Banana' not in fruits %}

<h1>Hello</h1>

{% else %}

<h1>Goodbye</h1>

{% endif %}

## Is

Check if two objects are the same.

|  |  |
| --- | --- |
| <!DOCTYPE html>  <html>  <body>  {% if x is y %}  <h1>YES</h1>  {% else %}  <h1>NO</h1>  {% endif %}  </body> </html>  Output: NO | from django.http import HttpResponse  from django.template import loader  def testing(request):  template = loader.get\_template('template.html')  context = {  'x': ['Apple', 'Banana', 'Cherry'],  'y': ['Apple', 'Banana', 'Cherry'],  }  return HttpResponse(template.render(context, request)) |

Let us try the same example with the == operator instead:

How can two objects be the same? Well, if you have two objects that points to the same object, then the is operator evaluates to true:

We will demonstrate this by using the {% with %} tag, which allows us to create variables in the template:

{% with var1=x var2=x %}

{% if var1 is var2 %}

<h1>YES</h1>

{% else %}

<h1>NO</h1>

{% endif %}

{% endwith %}

OUTPUT : YES

## is not

To check if two objects are not the same.

{% if x is not y %}

<h1>YES</h1>

{% else %}

<h1>NO</h1>

{% endif %}

## For Loops

A for loop is used for iterating over a sequence, like looping over items in an array, a list, or a dictionary.

|  |  |
| --- | --- |
| <!DOCTYPE html>  <html> <body>  {% for x in fruits %}  <h1>{{ x }}</h1>  {% endfor %}  </body> </html>  OUTPUT:  Apple  Banana  Cherry | from django.http import HttpResponse  from django.template import loader  def testing(request):  template = loader.get\_template('template.html')  context = {  'fruits': ['Apple', 'Banana', 'Cherry'], }  return HttpResponse(template.render(context, request)) |

Loop through a list of dictionaries: (QuerySet)

|  |  |
| --- | --- |
| <!DOCTYPE html>  <html>  <body>  {% for x in cars %}  <p>{{ x.brand }}</p>  <p>{{ x.model }}</p>  <p>{{ x.year }}</p>  {% endfor %}  </body>  </html>  Output:  Ford  Mustang  1964  Ford  Bronco  1970 | from django.http import HttpResponse  from django.template import loader  def testing(request):  template = loader.get\_template('template.html')  context = {  'cars': [  {  'brand': 'Ford',  'model': 'Mustang',  'year': '1964',  },  {  'brand': 'Ford',  'model': 'Bronco',  'year': '1970',  }  ]  }  return HttpResponse(template.render(context, request)) |

## Empty and reversed keyword

## Loop Variables

Django has some variables that are available for you inside a loop:

* forloop.counter
* forloop.counter0
* forloop.first
* forloop.last
* forloop.parentloop
* forloop.revcounter
* forloop.revcounter0

## Comments

<h1>Welcome Everyone</h1>

{% comment "this was the original welcome message" %}

<h1>Welcome ladies and gentlemen</h1>

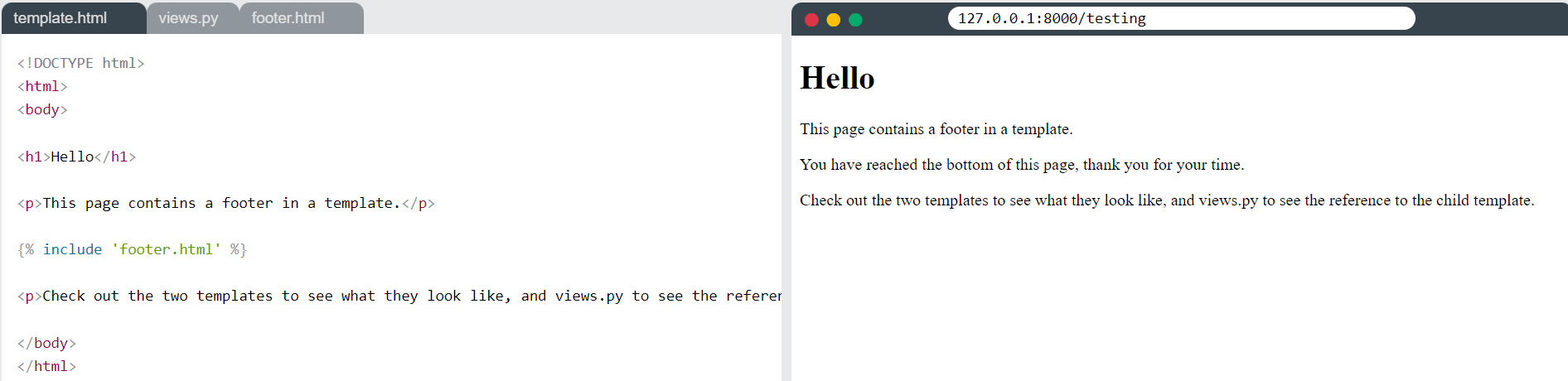
{% endcomment %}

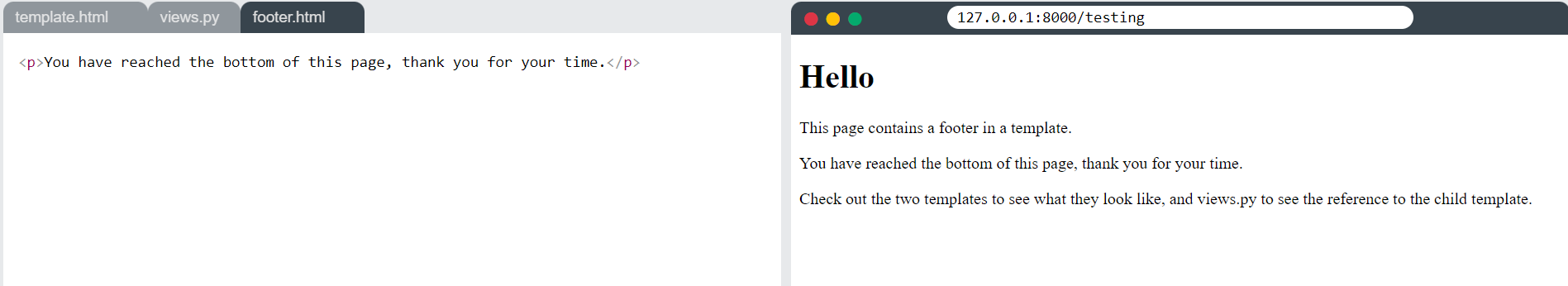
<h1>Welcome{# Everyone#}</h1>

## Include

The include tag allows you to include a template inside the current template.

This is useful when you have a block of content that is the same for many pages.





# QuerySet

1. A QuerySet is a collection of data from a database.
2. A QuerySet is built up as a list of objects.
3. QuerySets makes it easier to get the data you actually need, by allowing you to filter and order the data at an early stage.

we use the .all() method to get all the records and fields of the Member model:

from django.http import render

from .models import Member

def testing(request):

mydata = Member.objects.all()

context = { 'mymembers': mydata}

return render(reques,”index.html’,context)

The object is placed in a variable called mydata, and is sent to the template via the context object as mymembers, and looks like this:

<QuerySet [  
  <Member: Member object (1)>,  
  <Member: Member object (2)>,  
  <Member: Member object (3)>,  
  <Member: Member object (4)>,  
  <Member: Member object (5)>  
]>

As you can see, our Member model contains 5 records, and are listed inside the QuerySet as 5 objects.

## Get Data

There are different methods to get data from a model into a QuerySet.

### values() Method

The values() method allows you to return each object as a Python dictionary, with the key/value pairs:

<!DOCTYPE html> <html> <body>

<p>this return the data a list of dict :</p>

{{ mymembers }}

<table border='1'> <tr>

<th>ID</th>

<th>Firstname</th>

<th>Lastname</th>

</tr>

{% for x in mymembers %}

<tr>

<td>{{ x.id }}</td>

<td>{{ x.firstname }}</td>

<td>{{ x.lastname }}</td>

</tr>

{% endfor %}

</table> </body> </html>

Views.py

from django.shortcuts import render

from django.http import HttpResponse

from .models import Member

def members(req):

    mymembers = Member.objects.all().values()

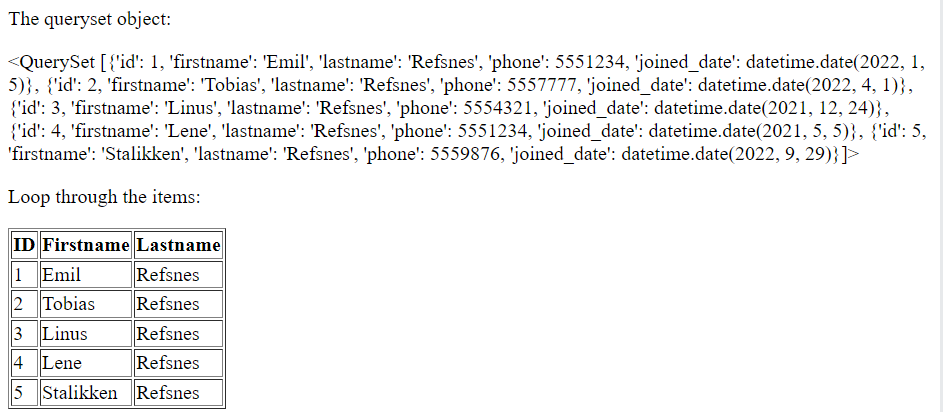
    context = {

    'mymembers': mymembers}

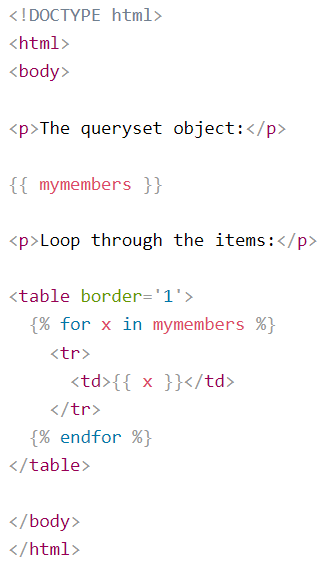
    print(context)

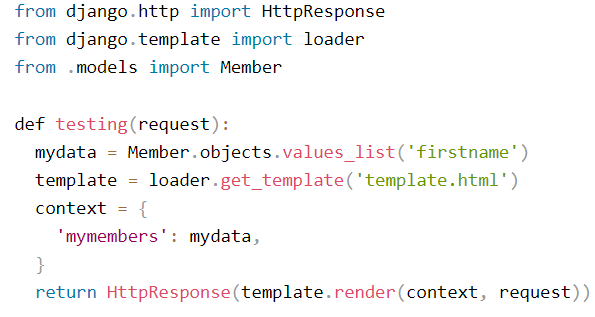
    return render(req,"myfirst.html",context)

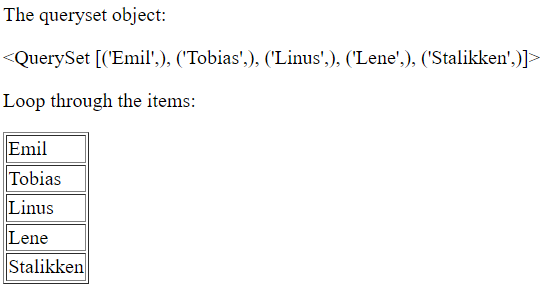
output



### values\_list()  method

The values\_list() method allows you to return only the columns that you specify.





output

## QuerySet Filter

The filter() method is used to filter your search, and allows you to return only the rows that matches the search term.

Return only the records where the firstname is 'Emil':

mydata = Member.objects.filter(firstname='Emil').values()

you can filter on more than one field by separating them by a comma.

Return records where lastname is "Refsnes" and id is 2:

mydata = Member.objects.filter(lastname='Refsnes', id=2).values()

To return records where firstname is Emil or firstname is Tobias (meaning: returning records that matches either query, not necessarily both) is not as easy as the AND example above.

We can use multiple filter() methods, separated by a pipe | character. The results will merge into one model.

Return records where firstname is either "Emil" or Tobias":

mydata = Member.objects.filter(firstname='Emil').values() | Member.objects.filter(firstname='Tobias').values()

**Another common method is to import and use Q expressions:**

Return records where firstname is either "Emil" or Tobias":

from django.db.models import Q

mydata= Member.objects.filter(Q(firstname='Emil') | Q(firstname='Tobias')).values()

### Field Lookups

To make specific where clauses in Django, use "Field lookups".

Use the \_\_startswith keyword:

.filter(firstname\_\_startswith='L');

The above statement will return records where firstname starts with 'L'.

### Field Lookups Reference

A list of all field look up keywords:

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| [contains](https://www.w3schools.com/django/ref_lookups_contains.php) | Contains the phrase |
| [icontains](https://www.w3schools.com/django/ref_lookups_icontains.php) | Same as contains, but case-insensitive |
| date | Matches a date |
| day | Matches a date (day of month, 1-31) (for dates) |
| [endswith](https://www.w3schools.com/django/ref_lookups_endswith.php) | Ends with |
| [iendswith](https://www.w3schools.com/django/ref_lookups_iendswith.php) | Same as endswidth, but case-insensitive |
| [exact](https://www.w3schools.com/django/ref_lookups_exact.php) | An exact match |
| [iexact](https://www.w3schools.com/django/ref_lookups_iexact.php) | Same as exact, but case-insensitive |
| [in](https://www.w3schools.com/django/ref_lookups_in.php) | Matches one of the values |
| isnull | Matches NULL values |
| [gt](https://www.w3schools.com/django/ref_lookups_gt.php) | Greater than |
| [gte](https://www.w3schools.com/django/ref_lookups_gte.php) | Greater than, or equal to |
| hour | Matches an hour (for datetimes) |
| [lt](https://www.w3schools.com/django/ref_lookups_lt.php) | Less than |
| [lte](https://www.w3schools.com/django/ref_lookups_lte.php) | Less than, or equal to |
| minute | Matches a minute (for datetimes) |
| month | Matches a month (for dates) |
| quarter | Matches a quarter of the year (1-4) (for dates) |
| [range](https://www.w3schools.com/django/ref_lookups_range.php) | Match between |
| regex | Matches a regular expression |
| iregex | Same as regex, but case-insensitive |
| second | Matches a second (for datetimes) |
| [startswith](https://www.w3schools.com/django/ref_lookups_startswith.php) | Starts with |
| [istartswith](https://www.w3schools.com/django/ref_lookups_istartswith.php) | Same as startswith, but case-insensitive |
| time | Matches a time (for datetimes) |
| week | Matches a week number (1-53) (for dates) |
| week\_day | Matches a day of week (1-7) 1 is sunday |
| iso\_week\_day | Matches a ISO 8601 day of week (1-7) 1 is monday |
| year | Matches a year (for dates) |
| iso\_year | Matches an ISO 8601 year (for dates) |

## QuerySet - Order By

To sort QuerySets, Django uses the order\_by() method:

Order the result alphabetically by firstname:

mydata = Member.objects.all().order\_by('firstname').values()

**Descending Order**

By default, the result is sorted ascending (the lowest value first), to change the direction to descending (the highest value first), use the minus sign (NOT), - in front of the field name:

Order the result firstname descending:

mydata = Member.objects.all().order\_by('-firstname').values()

**Multiple Order Bys**

To order by more than one field, separate the fieldnames with a comma in the order\_by() method:

Order the result first by lastname ascending, then descending on id:

mydata = Member.objects.all().order\_by('lastname', '-id').values()

# Django tags

|  |  |
| --- | --- |
| **Tag** | **Description** |
| [autoescape](https://www.w3schools.com/django/ref_tags_autoescape.php) | Specifies if autoescape mode is on or off |
| [block](https://www.w3schools.com/django/ref_tags_block.php) | Specifies a block section |
| [comment](https://www.w3schools.com/django/ref_tags_comment.php) | Specifies a comment section |
| csrf\_token | Protects forms from Cross Site Request Forgeries |
| [cycle](https://www.w3schools.com/django/ref_tags_cycle.php) | Specifies content to use in each cycle of a loop |
| debug | Specifies debugging information |
| [extends](https://www.w3schools.com/django/ref_tags_extends.php) | Specifies a parent template |
| [filter](https://www.w3schools.com/django/ref_tags_filter.php) | Filters content before returning it |
| [firstof](https://www.w3schools.com/django/ref_tags_firstof.php) | Returns the first not empty variable |
| [for](https://www.w3schools.com/django/ref_tags_for.php) | Specifies a for loop |
| [if](https://www.w3schools.com/django/ref_tags_if.php) | Specifies a if statement |
| [ifchanged](https://www.w3schools.com/django/ref_tags_ifchanged.php) | Used in for loops. Outputs a block only if a value has changed since the last iteration |
| [include](https://www.w3schools.com/django/ref_tags_include.php) | Specifies included content/template |
| load | Loads template tags from another library |
| [lorem](https://www.w3schools.com/django/ref_tags_lorem.php) | Outputs random text |
| [now](https://www.w3schools.com/django/ref_tags_now.php) | Outputs the current date/time |
| [regroup](https://www.w3schools.com/django/ref_tags_regroup.php) | Sorts an object by a group |
| [resetcycle](https://www.w3schools.com/django/ref_tags_resetcycle.php) | Used in cycles. Resets the cycle |
| [spaceless](https://www.w3schools.com/django/ref_tags_spaceless.php) | Removes whitespace between HTML tags |
| [templatetag](https://www.w3schools.com/django/ref_tags_templatetag.php) | Outputs a specified template tag |
| url | Returns the absolute URL part of a URL |
| [verbatim](https://www.w3schools.com/django/ref_tags_verbatim.php) | Specifies contents that should not be rendered by the template engine |
| widthratio | Calculates a width value based on the ratio between a given value and a max value |
| [with](https://www.w3schools.com/django/ref_tags_with.php) | Specifies a variable to use in the block |

# Difference between get () and filter ()

**Get()**

Use get() when you expect to retrieve a single unique object and want to raise an exception if there are multiple matches or no match at all.

**Filter()**

Use filter() when you want to retrieve a collection of objects that match the query criteria, and you don't want to raise exceptions if there are no matches.

# Django CRUD project with image