What is Django?

Django is a Python framework that makes it easier to create web sites using Python.

Django takes care of the difficult stuff so that you can concentrate on building your web applications.

Django emphasizes reusability of components, also refereed 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).

Django is especially helpful for database driven websites.

You will learn how to install Django in the next chapter.

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.

The most common way to extract data from a database is SQL. One problem with SQL is that you have to have a pretty good understanding of the database structure to be able to work with it.

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 models.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.

Templates are often .html files, with HTML code describing the layout of a web page, but it can also be in other file formats to present other results, but we will concentrate on .html files.

Django uses standard HTML to describe the layout, but uses 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 provide 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 you first Django web application, and the browser requests the URL, this is basically what happens:

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

Django can do a lot more than this, but this is basically what you will learn in this tutorial, and are the basic steps in a simple web application made with Django.

Django History

Django was invented by Lawrence Journal-World in 2003, to meet the short deadlines in the newspaper and at the same time meeting the demands of experienced web developers.

Initial release to the public was in July 2005.

Latest version of Django is 4.1 (Aug 2022).

## Django Requires Python

To check if your system has Python installed, run this command in the command prompt:

python --version

If Python is installed, you will get a result with the version number, like this

Python 3.9.2

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: <https://www.python.org/>

## PIP

To install Django, you must use a package manager like PIP, which is included in Python from version 3.4.

To check if your system has PIP installed, run this command in the command prompt:

pip --version

If PIP is installed, you will get a result with the version number.

For me, on a windows machine, the result looks like this:

pip 20.2.3 from c:\python39\lib\site-packages\pip (python 3.9)

If you do not have PIP installed, you can download and install it from this page: <https://pypi.org/project/pip/>

## Virtual Environment

It is suggested to have a dedicated virtual environment for each Django project, and one way to manage a virtual environment is [venv](https://docs.python.org/3/tutorial/venv.html" \t "_blank), which is included in Python.

With venv, you can create a virtual environment by typing this in the command prompt, remember to navigate to where you want to create your project:

Windows:

py -m venv myproject

Unix/MacOS:

python -m venv myproject

This will set up a virtual environment, and create a folder named "myproject" with subfolders and files, like this:

myproject  
  Include  
  Lib  
  Scripts  
  pyvenv.cfg

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

Windows:

myproject\Scripts\activate.bat

Unix/MacOS:

source myproject/bin/activate

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

Windows:

(myproject) C:\Users\Your Name>

Unix/MacOS:

(myproject) ... $

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

## Install Django

Finally, we can install Django.

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

Django is installed using pip, with this command:

Windows:

(myproject) C:\Users\Your Name>py -m pip install Django

Unix/MacOS:

(myproject) ... $ python -m pip install Django

Which will give a result that looks like this (at least on my Windows machine):

Collecting Django  
  Downloading Django-4.0.3-py3-none-any.whl (8.0 MB)  
      |████████████████████████████████| 8.0 MB 2.2 MB/s  
Collecting sqlparse>=0.2.2  
  Using cached sqlparse-0.4.2-py3-none-any.whl (42 kB)  
Collecting asgiref<4,>=3.4.1  
  Downloading asgiref-3.5.0-py3-none-any.whl (22 kB)  
Collecting tzdata; sys\_platform == "win32"  
  Downloading tzdata-2021.5-py2.py3-none-any.whl (339 kB)  
      |████████████████████████████████| 339 kB 6.4 MB/s  
Installing collected packages: sqlparse, asgiref, tzdata, Django  
Successfully installed Django-4.0.3 asgiref-3.5.0 sqlparse-0.4.2 tzdata-2021.5  
WARNING: You are using pip version 20.2.3; however, version 22.0.4 is available.  
You should consider upgrading via the 'C:\Users\Your Name\myproject\Scripts\python.exe -m pip install --upgrade pip' command.

That's it" Now you have installed Django in your new project, running in a virtual environment!

## Windows, Mac, or Unix?

You can run this project on either one. There are some small differences, like when writing commands in the command prompt, Windows uses py as the first word in the command line, while Unix and MacOS use python:

Windows:

py --version

Unix/MacOS:

python --version

In the rest of this tutorial, we will be using the Windows command.

## Check Django Version

You can check if Django is installed by asking for its version number like this:

(myproject) C:\Users\Your Name>django-admin --version

If Django is installed, you will get a result with the version number:

4.0.3

## What's Next?

Now you are ready to create a Django project in a virtual environment on your computer.

In the next chapters of this tutorial we will create a Django project and look at the various features of Django and hopefully make you a Django developer.

## My First Project

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

django-admin startproject myworld

Django creates a myworld folder on my computer, with this content:

myworld  
    manage.py  
    myworld/  
        \_\_init\_\_.py  
        asgi.py  
        settings.py  
        urls.py  
        wsgi.py

These are all files and folders with a specific meaning, you will learn about some of them later in this tutorial, but for now, it is more important to know that this is the location of your project, and that you can start building applications in it.

## 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 /myworld folder and execute this command in the command prompt:

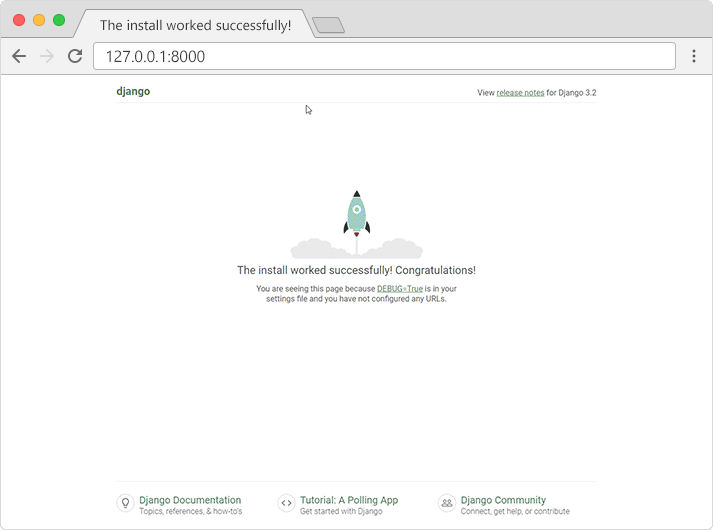
py manage.py runserver

Which will produce this result:

Watching for file changes with StatReloader  
Performing system checks...  
  
System check identified no issues (0 silenced).  
  
You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.  
Run 'python manage.py migrate' to apply them.  
December 02, 2021 - 13:14:51  
Django version 3.2.9, using settings 'myworld.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CTRL-BREAK.

Open a new browser window and type [127.0.0.1:8000](http://127.0.0.1:8000/) in the address bar.

The result:



## What's Next?

We have a Django project!

The next step is to make an app in your project.

You cannot have a web page created with Django without an 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.

In this tutorial we will create an app that allows us to list and register members in a database.

But first, let's just create a simple Django app that displays "Hello World!".

## Create App

I will name my app members.

Start by navigating to the selected location where you want to store the app, and run the command below.

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

py manage.py startapp members

Django creates a folder named members in my project, with this content:

myworld  
    manage.py  
    myworld/  
    members/  
        migrations/  
            \_\_init\_\_.py  
        \_\_init\_\_.py  
        admin.py  
        apps.py  
        models.py  
        tests.py  
        views.py

These are all files and folders with a specific meaning. You will learn about most of them later in this tutorial.

First, take a look at the file called views.py.

This is where we gather the information we need to send back a proper response.

You will learn more about views in the next chapter.

## 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:

members/views.py:

from django.shortcuts import render

# Create your views here.

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

members/views.py:

from django.shortcuts import render

from django.http import HttpResponse

def index(request):

return HttpResponse("Hello world!")

This is a simple example on how to send a response back to the browser.

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:

members/urls.py:

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

]

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

There is a file called urls.py on the myworld 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/members/.

Then your file will look like this:

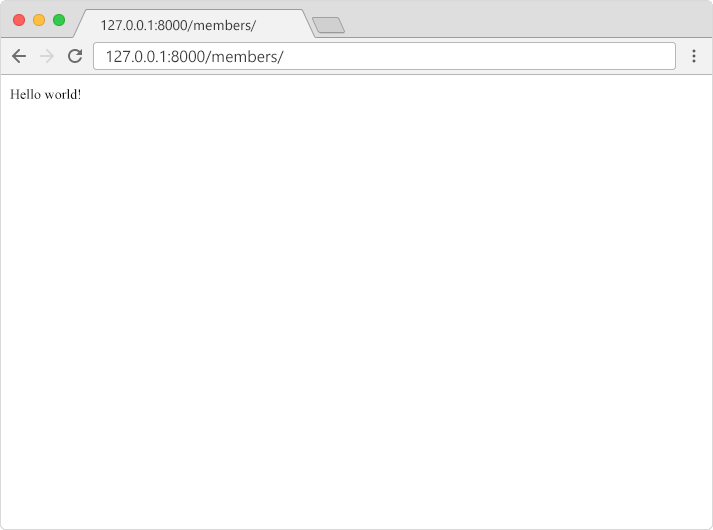
myworld/urls.py:

from django.contrib import admin  
from django.urls import include, path  
  
urlpatterns = [  
    path('members/', include('members.urls')),  
    path('admin/', admin.site.urls),  
]

If the server is not running, navigate to the /myworld folder and execute this command in the command prompt:

py manage.py runserver

In the browser window, type [127.0.0.1:8000/members/](http://127.0.0.1:8000/members/) in the address bar.



## Templates

In the [Django Intro](https://www.w3schools.com/django/django_intro.php) page, we learned that the result should be in HTML, and it should be created in a template, so let's do that.

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

The file structure should be something like this:

myworld  
    manage.py  
    myworld/  
    members/  
        templates/  
            myfirst.html

Open the HTML file and insert the following:

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 index view with this:

members/views.py:

from django.http import HttpResponse

from django.template import loader

def index(request):

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

return HttpResponse(template.render())

or

from django.shortcuts import render  
  
# Create your views here.  
  
def index(request):  
 return render(request,"myfirst.html")

## 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 myworld folder.

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

myworld/settings.py:

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'members.apps.MembersConfig'

]

Then run this command:

py manage.py migrate

Which will produce this output:

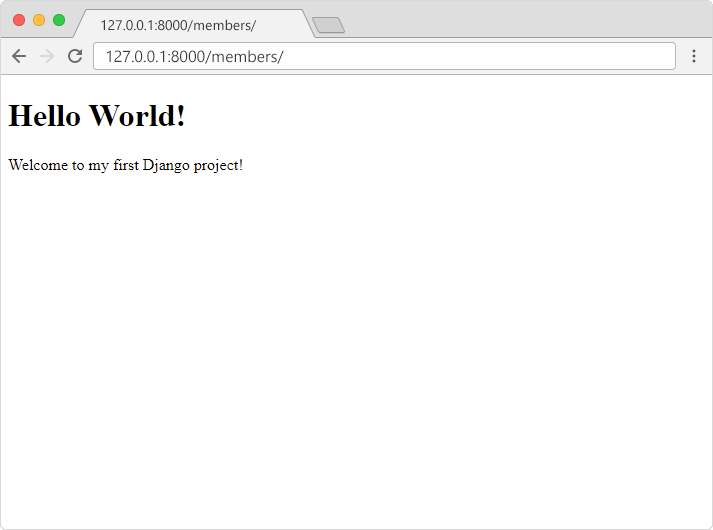
Operations to perform:  
  Apply all migrations: admin, auth, contenttypes, sessions  
Running migrations:  
  Applying contenttypes.0001\_initial... OK  
  Applying auth.0001\_initial... OK  
  Applying admin.0001\_initial... OK  
  Applying admin.0002\_logentry\_remove\_auto\_add... OK  
  Applying admin.0003\_logentry\_add\_action\_flag\_choices... OK  
  Applying contenttypes.0002\_remove\_content\_type\_name... OK  
  Applying auth.0002\_alter\_permission\_name\_max\_length... OK  
  Applying auth.0003\_alter\_user\_email\_max\_length... OK  
  Applying auth.0004\_alter\_user\_username\_opts... OK  
  Applying auth.0005\_alter\_user\_last\_login\_null... OK  
  Applying auth.0006\_require\_contenttypes\_0002... OK  
  Applying auth.0007\_alter\_validators\_add\_error\_messages... OK  
  Applying auth.0008\_alter\_user\_username\_max\_length... OK  
  Applying auth.0009\_alter\_user\_last\_name\_max\_length... OK  
  Applying auth.0010\_alter\_group\_name\_max\_length... OK  
  Applying auth.0011\_update\_proxy\_permissions... OK  
  Applying auth.0012\_alter\_user\_first\_name\_max\_length... OK  
  Applying sessions.0001\_initial... OK  
  
(myproject)C:\Users\Your Name\myproject\myworld>

Start the server by navigating to the /myworld folder and execute this command:

py manage.py runserver

In the browser window, type [127.0.0.1:8000/members/](http://127.0.0.1:8000/members/) in the address bar.

The result should look like this:



## SQLite Database

When we created the Django project, we got an empty SQLite database. It was created in the myworld root folder.

We will use this database in this tutorial.

## Create Table (Model)

To create a new table, we must create a new model.

In the /members/ folder, open the models.py file. It is almost empty by default, with only an import statement and a comment:

members/models.py:

from django.db import models

# Create your models here.

To add a Members table in our database, start by creating a Members class, and describe the table fields in it:

members/models.py:

from django.db import models

class Members(models.Model):

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

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

The first field, "firstname" is a Text field, and will contain the first name of the members.

The second field, "lastname" is also a Text field, with the members' last name.

Both "firstname" and "lastname" is set up to have a maximum of 255 characters.

Then navigate to the /myworld/ folder and run this command:

py manage.py makemigrations members

Which will result in this output:

Migrations for 'members':  
  members\migrations\0001\_initial.py  
    - Create model Members  
  
(myproject) C:\Users\Your Name\myproject\myworld>

Django creates a file with any new changes and stores the file in the /migrations/ folder.

Next time you run py manage.py migrate Django will create and execute an SQL statement, based on the content of the new file in the migrations folder.

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.0001\_initial... OK  
  
(myproject) C:\Users\Your Name\myproject\myworld>

The SQL statement created from the model is:

CREATE TABLE "members\_members" (

"id" INT NOT NULL PRIMARY KEY AUTOINCREMENT,

"firstname" varchar(255) NOT NULL,

"lastname" varchar(255) NOT NULL);

Now you have a Members table in you database!

Add Records

The Members table is empty, we should add some members to it.

In the next chapters you will learn how to make a user interface that will take care of CRUD operations (Create, Read, Update, Delete), but for now, let's write Python code directly in the Python interpreter (Python shell) and add some members in our database, without the user interface.

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 Members

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

>>> Members.objects.all()

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

<QuerySet []>

A QuerySet is a collection of data from a database.

Read more about QuerySets in the [Django QuerySet](https://www.w3schools.com/django/django_queryset.php) chapter.

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

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

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

>>> Members.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 Members obejcts, and execute .save() on each entry:

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

Now there are 5 members in the Members table:

>>> Members.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'}]>

Admin view

from django.contrib import admin  
from .models import Members  
  
# Register your models here.  
# admin.site.register(Members)  
@admin.register(Members)  
class Members(admin.ModelAdmin):  
 list\_display = ['id','firstname','lastname']

View in Browser

We want to see the result in a web page, not in a Python shell environment.

To see the result in a web page, we can create a *view* for this particular task.

In the members app, open the views.py file, if you have followed the previous chapters of this tutorial, it should look like this:

members/views.py:

from django.http import HttpResponse

from django.template import loader

def index(request):

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

HttpResponse(template.render())

Change the content in the views.py file to look like this instead:

members/views.py:

from django.http import HttpResponse

from django.template import loader

from .models import Members

def index(request):

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

output = ""

for x in mymembers:

output += x["firstname"]

return HttpResponse(output)

As you can see in line 3, the Members model is imported, and the index view does the following:

* makes a mymembers object with all the values of the Members model.
* Loops through all items in the mymembers object to build a string with all the firstname values.
* Returns the string as output to the browser.

See the result in your browser. If you are still in the Python shell, write this command to exit the shell:

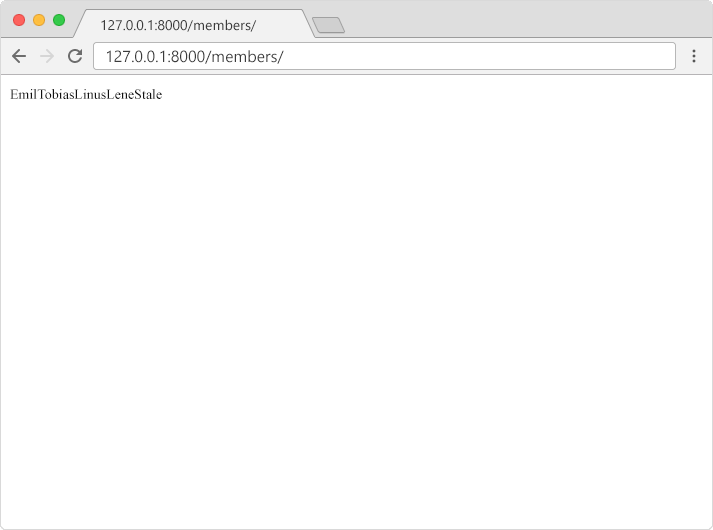
>>> quit()

Navigate to the /myworld/ folder and type this to start the server:

py manage.py runserver

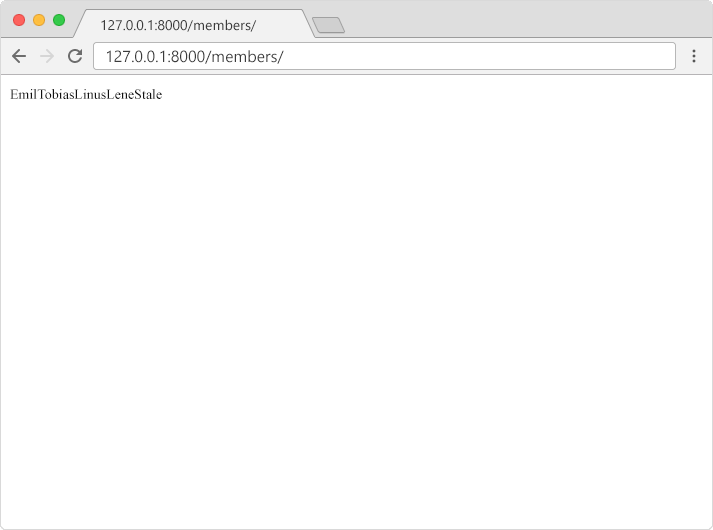
In the browser window, type [127.0.0.1:8000/members/](http://127.0.0.1:8000/members/) in the address bar.

The result:



Adding a Template to the Application

In the previous chapter, we managed to display the content of a database table in a web page:



To add some HTML around the values, we will create a template for the application.

All templates must be located in the templates folder off your app, if you have not already created a templates folder, do it now.

In the templates folder, create a file named index.html, with the following content:

members/templates/index.html:

<h1>Members</h1>

<table border="1">

{% for x in mymembers %}

<tr>

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

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

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

</tr>

{% endfor %}

</table>

Did you notice the {%  %} and {{  }} parts? They are called template tags.

Template tags allows you to perform logic and render variables in your templates, you will learn more about template tags later.

Modify the View

Change the index view to include the template:

members/views.py:

from django.http import HttpResponse

from django.template import loader

from .models import Members

def index(request):

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

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

context = {

'mymembers': mymembers,

}

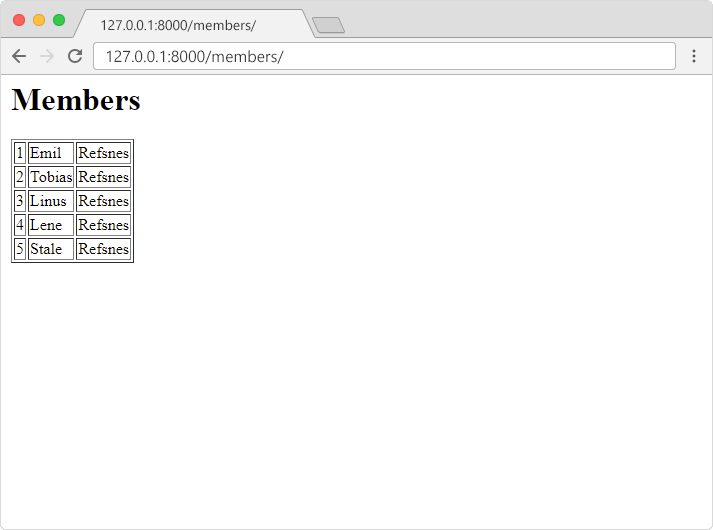
return HttpResponse(template.render(context, request))

The index view does the following:

* Creates a mymembers object with all the values of the Members model.
* Loads a the index.html template.
* Creates an object containing the mymember object.
* Sends the object to the template.
* Outputs the HTML that is rendered by the template.

In the browser window, type [127.0.0.1:8000/members/](http://127.0.0.1:8000/members/) in the address bar.

The result:



Adding Records

So far we have created a Members table in our database, and we have inserted five records by writing code in the Python shell.

We have also made a template that allows us to display the content of the table in a web page.

Now we want to be able to create new members from a web page.

Template

Start by adding a link in the members template:

members/templates/index.html:

<h1>Members</h1>

<table border="1">

{% for x in mymembers %}

<tr>

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

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

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

</tr>

{% endfor %}

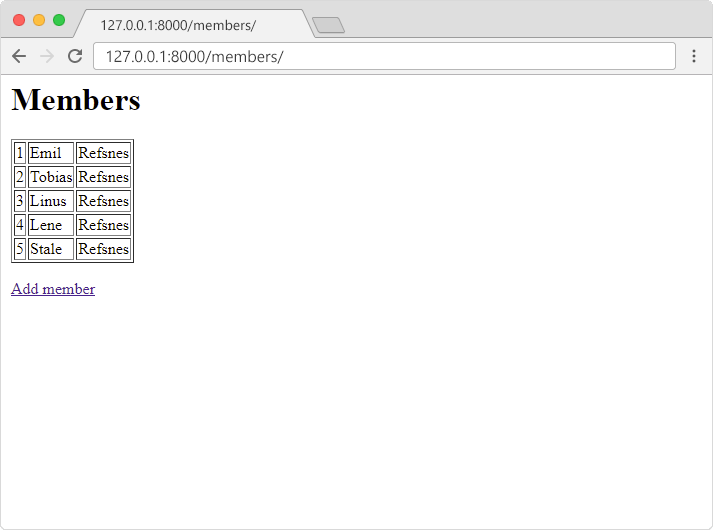
</table>

<p>

<a href="add/">Add member</a>

</p>

The result will look like this:



New Template

Add a new template in the templates folder, named add.html:

members/templates/add.html:

<h1>Add member</h1>

<form action="addrecord/" method="post">

{% csrf\_token %}

First Name:<br>

<input name="first">

<br><br>

Last Name:<br>

<input name="last">

<br><br>

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

</form>

The template contains an empty HTML form with two input fields and a submit button.

Note: Django requires this line in the form:  
{% csrf\_token %}  
to handle Cross Site Request Forgeries in forms where the method is POST.

View

Next, add a view in the members/views.py file, name the new view add:

members/views.py:

from django.http import HttpResponse

from django.template import loader

from .models import Members

def index(request):

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

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

context = {

'mymembers': mymembers,

}

return HttpResponse(template.render(context, request))

def add(request):

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

return HttpResponse(template.render({}, request))

URLs

Add a path() function in the members/urls.py file, that points the url 127.0.0.1:8000/members/add/ to the right location:

members/urls.py:

from django.urls import path

from . import views

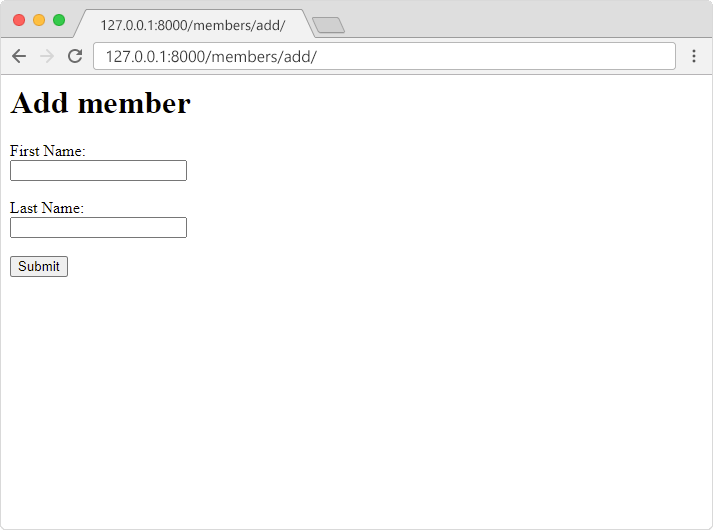
urlpatterns = [

path('', views.index, name='index'),

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

]

In the browser, click the "Add member" link and the result should look like this:



More URLs

Did you notice the action attribute in the HTML form? The action attribute specifies where to send the form data, in this case the form data will be sent to addrecord/, so we must add a path() function in the members/urls.py file that points to the right view:

members/urls.py:

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

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

path('add/addrecord/', views.addrecord, name='addrecord'),

]

Code for Adding Records

So far we have made the user interface, and we point the URL to the view called addrecord, but we have not made the view yet.

Make sure you add the addrecord view in the in the members/views.py file:

members/views.py:

from django.http import HttpResponse, HttpResponseRedirect

from django.template import loader

from django.urls import reverse

from .models import Members

def index(request):

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

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

context = {

'mymembers': mymembers,

}

return HttpResponse(template.render(context, request))

def add(request):

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

return HttpResponse(template.render({}, request))

def addrecord(request):

x = request.POST['first']

y = request.POST['last']

member = Members(firstname=x, lastname=y)

member.save()

return HttpResponseRedirect(reverse('index'))

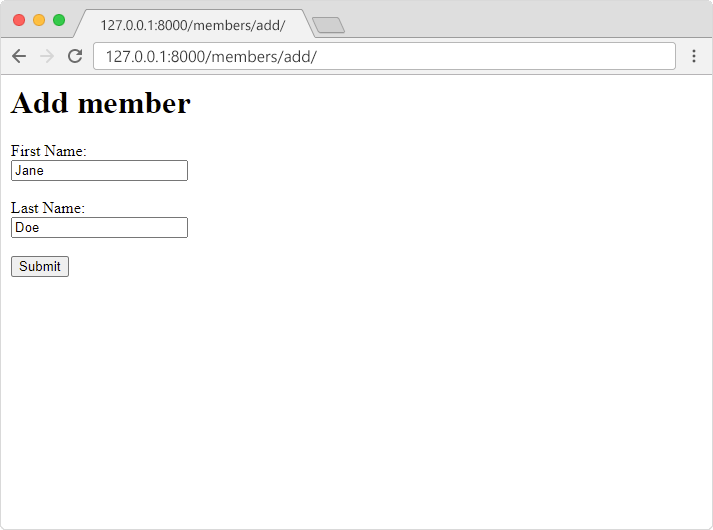
Changes that are made in the views.py file:

Line 1: import HttpResponseRedirect  
Line 3: import reverse

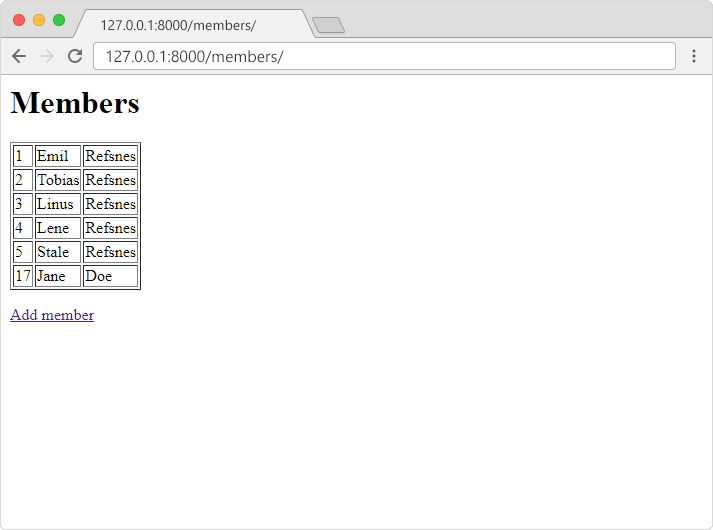
The addrecord view does the following:

* Gets the first name and last name with the request.POST statement.
* Adds a new record in the members table.
* Redirects the user back to the index view.

Try to add a new record and see how it works:



If you press the submit button, the members table should have been updated:



Deleting Records

To delete a record we do not need a new template, but we need to make some changes to the members template.

Of course, you can chose how you want to add a delete button, but in this example, we will add a "delete" link for each record in a new table column.

The "delete" link will also contain the ID of each record.

Modify Template

Add a "delete" column in the members template:

members/templates/index.html:

<h1>Members</h1>

<table border="1">

{% for x in mymembers %}

<tr>

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

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

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

<td><a href="delete/{{ x.id }}">delete</a></td>

</tr>

{% endfor %}

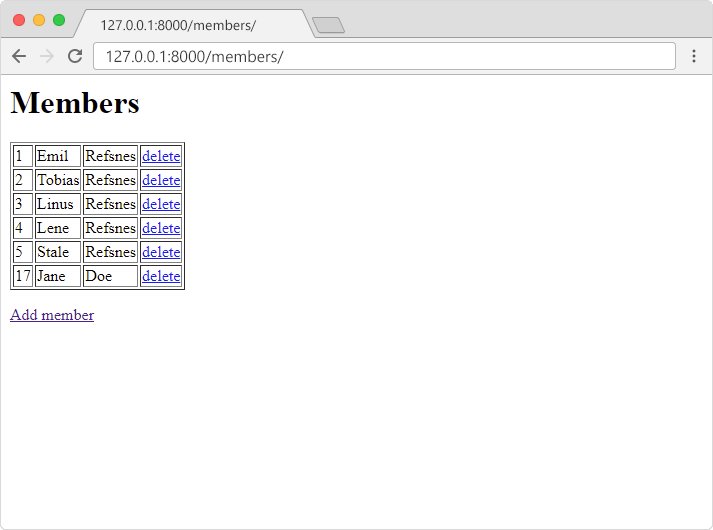
</table>

<p>

<a href="add/">Add member</a>

</p>

The result will look like this:



URLs

The "delete" link in the HTML table points to 127.0.0.1:8000/members/delete/ so we will add a path() function in the members/urls.py file, that points the url to the right location, with the ID as a parameter:

members/urls.py:

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

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

path('add/addrecord/', views.addrecord, name='addrecord'),

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

]

Code for Deleting Records

Now we need to add a new view called delete in the members/views.py file:

members/views.py:

from django.http import HttpResponse, HttpResponseRedirect

from django.template import loader

from django.urls import reverse

from .models import Members

def index(request):

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

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

context = {

'mymembers': mymembers,

}

return HttpResponse(template.render(context, request))

def add(request):

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

return HttpResponse(template.render({}, request))

def addrecord(request):

x = request.POST['first']

y = request.POST['last']

member = Members(firstname=x, lastname=y)

member.save()

return HttpResponseRedirect(reverse('index'))

def delete(request, id):

member = Members.objects.get(id=id)

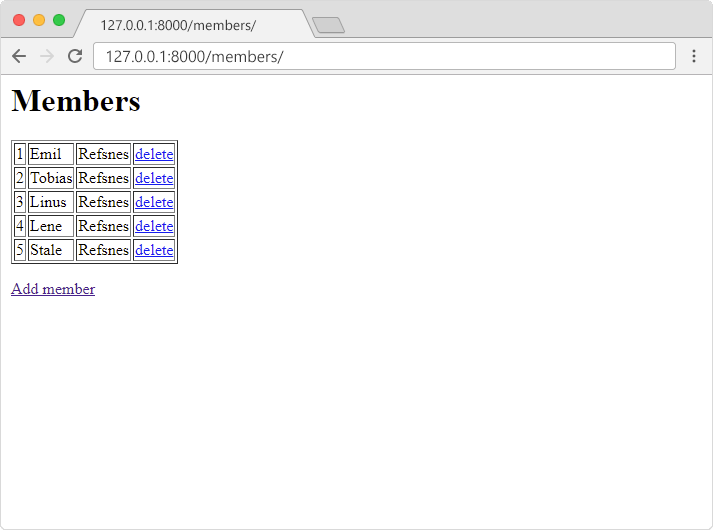
member.delete()

return HttpResponseRedirect(reverse('index'))

The delete view does the following:

* Gets the id as an argument.
* Uses the id to locate the correct record in the Members table.
* Deletes that record.
* Redirects the user back to the index view.

Click on the "delete" link for Jane Doe, and see the result:



Updating Records

To update a record, we need the ID of the record, and we need a template with an interface that let us change the values.

First we need to make some changes in the index.html template.

Modify Template

Start by adding a link for each member in the table:

members/templates/index.html:

<h1>Members</h1>

<table border="1">

{% for x in mymembers %}

<tr>

<td><a href="update/{{ x.id }}">{{ x.id }}</a></td>

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

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

<td><a href="delete/{{ x.id }}">delete</a>

</tr>

{% endfor %}

</table>

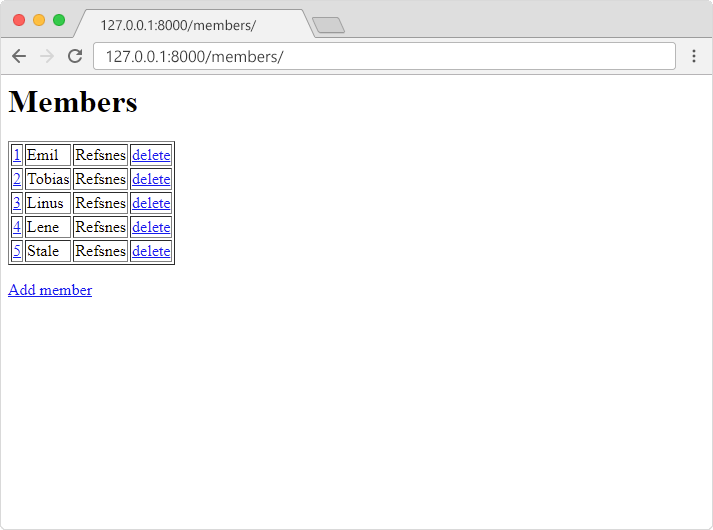
<p>

<a href="add/">Add member</a>

</p>

The link goes to a view called update with the ID of the current member.

The result will look like this:



View

Next, add the update view in the members/views.py file:

members/views.py:

from django.http import HttpResponse, HttpResponseRedirect

from django.template import loader

from django.urls import reverse

from .models import Members

def index(request):

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

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

context = {

'mymembers': mymembers

}

return HttpResponse(template.render(context, request))

def add(request):

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

return HttpResponse(template.render({}, request))

def addrecord(request):

first = request.POST['first']

last = request.POST['last']

member = Members(firstname=first, lastname=last)

member.save()

return HttpResponseRedirect(reverse('index'))

def delete(request, id):

member = Members.objects.get(id=id)

member.delete()

return HttpResponseRedirect(reverse('index'))

def update(request, id):

mymember = Members.objects.get(id=id)

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

context = {

'mymember': mymember,

}

return HttpResponse(template.render(context, request))

The update view does the following:

* Gets the id as an argument.
* Uses the id to locate the correct record in the Members table.
* loads a template called update.html.
* Creates an object containing the member.
* Sends the object to the template.
* Outputs the HTML that is rendered by the template.

New Template

Add a new template in the templates folder, named update.html:

members/templates/update.html:

<h1>Update member</h1>

<form action="updaterecord/{{ mymember.id }}" method="post">

{% csrf\_token %}

First Name:<br>

<input name="first" value="{{ mymember.firstname }}">

<br><br>

Last Name:<br>

<input name="last" value="{{ mymember.lastname }}">

<br><br>

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

</form>

The template contains an HTML form with the values from the selected member.

Note: Django requires this line in the form:  
{% csrf\_token %}  
to handle Cross Site Request Forgeries in forms where the method is POST.

URLs

Add a path() function in the members/urls.py file, that points the url 127.0.0.1:8000/members/update/ to the right location, with the ID as a parameter:

members/urls.py:

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

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

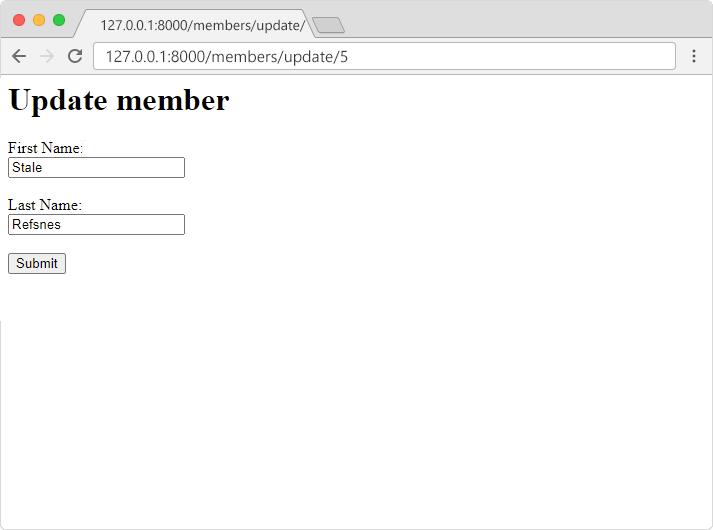
path('add/addrecord/', views.addrecord, name='addrecord'),

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

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

]

In the browser, click the ID of the member you want to change and the result should look like this:



What Happens on Submit?

Did you notice the action attribute in the HTML form? The action attribute specifies where to send the form data, in this case the form data will be sent to:  
updaterecord/{{ mymember.id }}, so we must add a path() function in the members/urls.py file that points to the right view:

members/urls.py:

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

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

path('add/addrecord/', views.addrecord, name='addrecord'),

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

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

path('update/updaterecord/<int:id>', views.updaterecord, name='updaterecord'),

]

Code for Updating Records

So far we have made the user interface, and we point the URL to the view called updaterecord, but we have not made the view yet.

Make sure you add the updaterecord view in the in the members/views.py file:

members/views.py:

from django.http import HttpResponse, HttpResponseRedirect

from django.template import loader

from django.urls import reverse

from .models import Members

def index(request):

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

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

context = {

'mymembers': mymembers,

}

return HttpResponse(template.render(context, request))

def add(request):

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

return HttpResponse(template.render({}, request))

def addrecord(request):

x = request.POST['first']

y = request.POST['last']

member = Members(firstname=x, lastname=y)

member.save()

return HttpResponseRedirect(reverse('index'))

def delete(request, id):

member = Members.objects.get(id=id)

member.delete()

return HttpResponseRedirect(reverse('index'))

def update(request, id):

mymember = Members.objects.get(id=id)

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

context = {

'mymember': mymember,

}

return HttpResponse(template.render(context, request))

def updaterecord(request, id):

first = request.POST['first']

last = request.POST['last']

member = Members.objects.get(id=id)

member.firstname = first

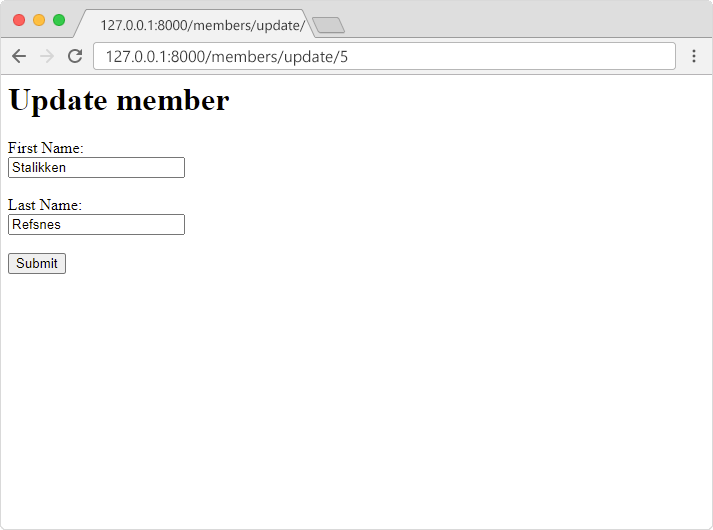
member.lastname = last

member.save()

return HttpResponseRedirect(reverse('index'))

The updaterecord function will update the record in the members table with the selected ID.

Try to update a record and see how it works:



If you press the submit button, the members table should have been updated:

