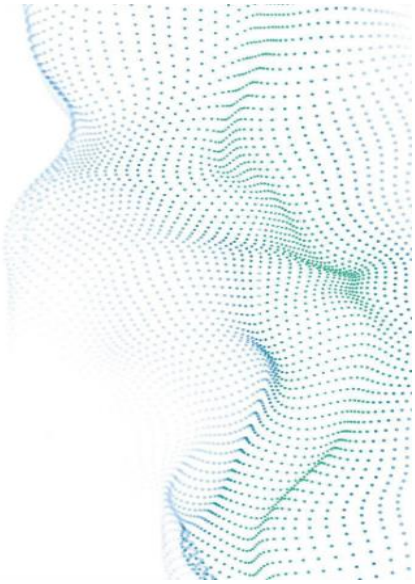
Module 1
Django Framework



Django

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 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).

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 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`.

When you have installed Django and created your 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 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
```

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)
```

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, which is included in Python.

The name of the virtual environment is your choice, in this tutorial we will call it `myworld`.

Type the following in the command prompt, remember to navigate to where you want to create your project:

Windows:

```
py  -m venv myworld
```

Unix/MacOS:

```
python -m venv myworld
```


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

```
myworld
  Include
  Lib
  Scripts
  pyvenv.cfg
```

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

Windows:

```
myworld\Scripts\activate.bat
```

Unix/MacOS:

```
source myworld/bin/activate
```


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

Windows:

```
(myworld) C:\Users\Your Name>
```

Unix/MacOS:

```
(myworld) ... $
```

Install Django

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

Django is installed using pip, with this command:

Windows:

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

Unix/MacOS:

```
(myworld) ... $ 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 22.3 is available.
You should consider upgrading via the 'C:\Users\Your Name\myworld\Scripts\python.exe -m pip install --upgrade pip'
command.
```


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
```

Check Django Version

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

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

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

```
4.1.2
```


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.

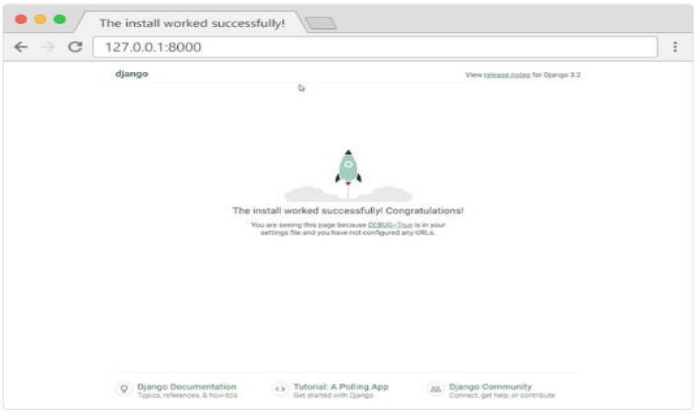
Navigate to the `/my_tennis_club` folder and execute this command in the command prompt:

```
py manage.py runserver
```

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.
October 27, 2022 - 13:03:14
Django version 4.1.2, using settings 'my_tennis_club.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

The result:



Create App

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
```


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

```
my_tennis_club
    manage.py
    my_tennis_club/
    members/
        migrations/
            __init__.py
        __init__.py
        admin.py
        apps.py
        models.py
        tests.py
        views.py
```

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

# Create your views here.
```

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!")
```

Note: The name of the view does not have to be the same as the application.

I call it `members` because I think it fits well in this context.

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

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:

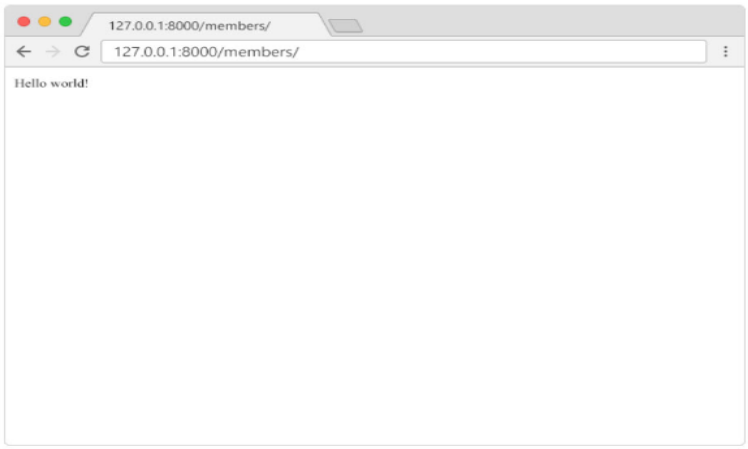
`my_tennis_club/my_tennis_club/urls.py` :

```
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
```


In the browser window, type `127.0.0.1:8000/members/` in the address bar.



Templates

In the Django Intro 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 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
```

Which will produce this output:

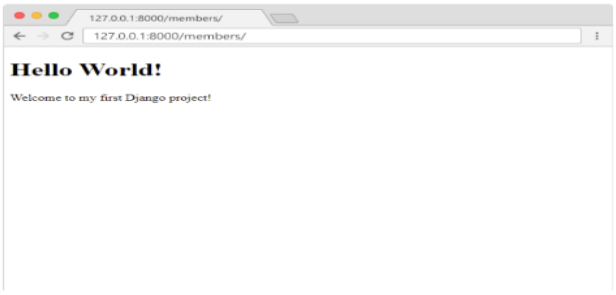
```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... 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

(myworld) C:\Users\Your Name\myworld\my_tennis_club>
```


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

```
py manage.py runserver
```

In the browser window, type `127.0.0.1:8000/members/` in the address bar.

The result should look like this:

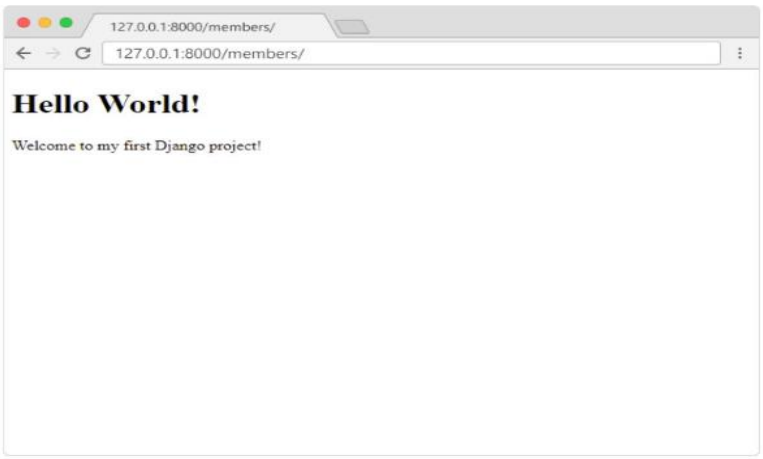


Start the server by navigating to the `/my_tennis_club` folder and execute this command:

```
py manage.py runserver
```

In the browser window, type `127.0.0.1:8000/members/` in the address bar.

The result should look like this:



Django Models

Up until now in this tutorial, output has been static data from Python or HTML templates.

Now we will see how Django allows us to work with data, without having to change or upload files in the process.

In Django, data is created in objects, called Models, and is actually tables in a database.

Create Table (Model)

To create a model, navigate to the `models.py` file in the `/members/` folder.

Open it, and add a `Member` table by creating a `Member class`, and describe the table fields in it:

`my_tennis_club/members/models.py` :

```
from django.db import models

class Member(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 member's last name.

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

Migrate

Now when we have described a Model in the `models.py` file, we must run a command to actually create the table in the database.

Navigate to the `/my_tennis_club/` 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 Member

(myworld) C:\Users\Your Name\myworld\my_tennis_club>
```


my_tennis_club/members/migrations/0001_initial.py :

```
# Generated by Django 4.1.2 on 2022-10-27 11:14

from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Member',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('firstname', models.CharField(max_length=255)),
                ('lastname', models.CharField(max_length=255)),
            ],
        ),
    ]
```


The table is not created yet, you will have to run one more command, then 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

(myworld) C:\Users\Your Name\myworld\my_tennis_club>
```

Now you have a `Member` table in you database!

View SQL

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
```

Which will result in this output:

```
BEGIN;
--
-- Create model Member
--
CREATE TABLE "members_member" ("id" integer NOT NULL PRIMARY KEY
AUTOINCREMENT, "firstname" varchar(255) NOT NULL, "lastname"
varchar(255) NOT NULL); COMMIT;
```


Add Records

The Members table created in the previous chapter is empty.

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.

Read more about QuerySets in the Django QuerySet chapter.

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:

```
>>> x.firstname
```


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` will now represent the member at index 5, which is "Jane Doe", but to make sure, let us see if that is correct:

```
>>> x.firstname
```


This should give you this result:

```
'Jane'
```

Now we can delete the record:

```
>>> 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'}]>
```