Server Side Web Development using Django

https://www.djangoproject.com/

https://docs.djangoproject.com/en/5.0/intro/

Índice

1. Installation of Django

https://docs.docker.com/samples/django/

To create the web sever and a Postgress server we can follow the steps described on:

https://github.com/docker/awesome-compose/tree/master/official-documentation-samples/django/

To start with we will only need the web server and therefore we will follow the steps that correspond to this server:

1. Create the project folder: password_generator-project and go to that folder.
2. Create the Dockerfile.
3. Create the requirements.txt file.
4. Create the docker-compose.yml file.
5. Run a one time command on the web image:

   To create the Django project:

   docker compose run web django-admin startproject password_generator .

   To apply Django migrations:

   docker compose run web python manage.py migrate
6. Check that the directory tree and the file names are correct.
7. docker compose up
8. Check that the web server is running: http://localhost:8000
9. To stop the containers: docker compose down.

2. Create a new Django project

We have created a new project in the previous section, using:

    django-admin startproject password_generator

We have fired up our Django project using the docker-compose.yml file:

    python3 manage.py runserver

We have checked that it works on http://127.0.0.1:8000

3. Site 1 – Password Generator

Project Structure

We have the project created in the folder password_generator. We rename it to password_generator-project.

manage.py – We shouldn't modify it.

We can go to a terminal and execute it: python3 manage.py help --> It shows us all things it can do

db.sqlite3 is a database.

Inside our password_generator folder we have:

pycache This is cached code for when we run our project and it is generated by Python.

init.py allows us to add some features for when Python runs the project for the very first time.

asgi.py and **wsgi.py** are used when deploying a project, when making a website live on the Internet.

settings.py

BASE_DIR says where in our computer the project is living

SECRET_KEY is used when we have data flowing in and out of our website. We must not share this with others because if they get it they can do some serious manipulation of our data. It only applies when we launch our project on the Internet. We can change it if somebody gets it.

DEBUG indicates whether our web is in debug mode or not. When it is in debug mode, whenever there is a problem it shows us a lot of information to help us resolve the problem.

INSTALLED_APPS is a way to bring in different pieces of code into your project.s

MIDDLEWARE is some kind of built in django things that can help out our project.

ROOT_URLCONF tells where our URL file is.

TEMPLATES is related to the web page that will be shown on the web browser.

DATABASES indicates the databases in our project.

LANGUAGE_CODE, TIME_ZONE, etc. We can change and customize it.

urls.py id the starting point for any django project. When we type an url on the web browser django comes to this file and checks which code is associated to it and must be executed.

Apps

A Django project often consists of multiple modules that make up the project. Those modules are called apps. We store our application code in those apps.

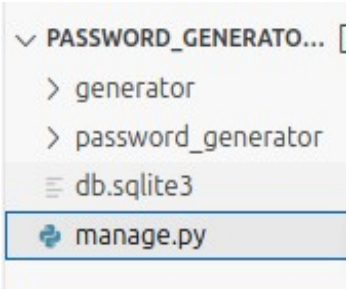
To add features to your entire project you add multiple apps that hold those features.

When we build a website with Django, we create a project for this and the entire project is our website.

e.g. if we create a shop we have a feature for displaying products and we have a feature for handling the cart and the checkout process. Those features would be represented by apps in this project. To split our code and logic by feature into these different apps.

Apps are the building blocks that form this overall project.

Going on with our example:



To create an app we type:

python3 manage.py startapp nameOfApp

In our example we will only have one app. To create it:

python3 manage.py startapp generator

We have to stop the dockers and run:

docker compose run --rm web python manage.py startapp generator

and start again the dockers.

It added a new folder to our project called generator with several files. In this project we'll only use views.py.

We have to **let our settings know that we just added an app.** We edit password_generator\ settings.py and add it:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'generator',
]
```

URLs

password_generator\urls.py is the file that is referenced any time someone types some sort of URL for our website.

By default we have the path admin defined. We can type localhost:8000/admin and check how it works.

We delete the admin lines because we're not going to use them.

We import the views file, that we're going to use.

We add a path with no url (that's the home page) and the file views.home.

```
from django.urls import path # type: ignore
from generator import views

urlpatterns = [
    path('', views.home),
    path('password/', views.password),
]
```

We add a path password. We use the '/' at the end so that it is posible to write both 'password' and 'password/' in the URL on the browser.

We create the corresponding views:

generator > views.py > home

```
1  from django.shortcuts import render
2  from django.http import HttpResponse
3
4  # Create your views here.
5
6  def home(request):
7      return HttpResponse('Hello there')
8
```

Templates

They define the visual part of our website.

We create a "templates/generator" folders inside our generator folder. Inside we create a home.html file.

In settings.py, in TEMPLATES we edit 'DIRS': ['templates']

settings.py ● urls.py views.py × home.html

generator > views.py > home

```python
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def home(request):
    # return HttpResponse('Hello there')
    return render(request, 'generator/home.html', {'password':'hui43fasdfsd'})
```

generator > templates > generator > home.html > ...

```
<h2></h2>This is the new home page.</h2>
{{ password }}
```

To add a hyperlink:

Home

And in the url.py file we assign a name to home:

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

Forms

In the 'action' of the form we can write the name of the path, e.g. 'password'.

We can add a name to the path:

```python
urlpatterns = [
    path('', views.home),
    path('password/', views.password, name='password'),
]
```

And use the name in the action of the form:

```
<form action="{% url 'password' %}">
```

In the **view** we write the code that is executed when submiting the form. We can access the form variables through the request object:

```
length = int(request.GET.get('length', 12))   # 12 is the default value in case length has no value
```

With checkboxes, when they are checked the value of the variable is 'true'.

BootStrap

In the tutorial he is using BootStrap 4.

We connect to the bootstrap web page and we copy the bootstrap link to include it in our web pages.

4. Git

Git is a free and open source distributed version control system.

Git allows you to have save points in your code so that if you ever really mess something up you can go back in time. It's also a great way to work with multiple people on a project.

We're going to add git to the project that we have just made.

First we check that we have it installed on our computer.

It it's not installed we look for information about how to install it in: https://git-scm.com/

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

We follow these steps:

- We open a terminal and go to the main folder of our project.
- We type 'git init' and now our project is already a git project. 'ls -a' or 'dir /a' to see the new folder '.git'
- git add -A (git status)
- git commit -m "Our first commit " (git status)
- If we need to go back to the previous commit: git stash
- If we need to go to a differente previous commits, we check the previous commits with "git log" and "git checkout commit_id"

Github

Create an account and sign in.

Click on "+" and "New repository" and create a private repository.

Generate a token so that you can push from the command console in your computer:

On the right click on the user icon and on "Settings". Then on the menu on the left bottom click on "Developer Settings" and then on "Tokens(classic)" and "Generate new token (classic)".

We push an existing repository form the command line:

```
git branch -M main
git remote add origin https://github.com/eploira/django3-password-generator.git
git push -u origin main
```

We reload the repository page on Github and we can see the code.

If we prefer git GUI we can use some apps as for example 'sourcetree'.

Some git commands

git remote -v → remote repository where we are working

git status → Stage area status

git log → To see all the previous commits.

Git show –name-only → Show a list of all the files that were commited.

Modify password

git config --global --unset user.password

Ao facer git pull origin main xa pide o password

Senón probar con: git config --global user.password “xxxx”

From local computer to the cloud

git add -A → Add all files

git commit -m “Primeira version”

git push origin main

From the cloud to local computer

git pull origin main

git fetch → Are there any changes in the remote directory to download?

pull = fetch + merge

Some other commands

git remove file_to_delete

git checkout hashCode --> to return to a previous state

git branch --> to see the branches

git branch name_of_new_branch --> to create a new branch

git checkout name_of_branch --> to go to that branch

5. Object Relational Mapping (ORM)

ORM is an acronym for the object-relational mapper.

The ORM's main goal is to transmit data between a relational database and an application model.

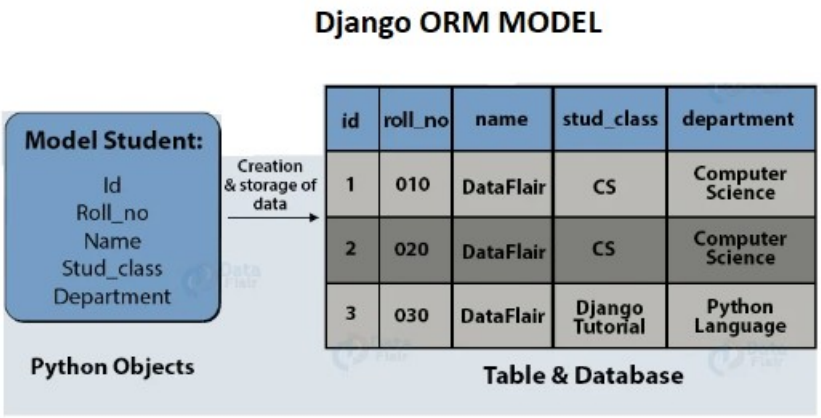
The ORM automates this transmission, such that the developer need not write any SQL.

The Django web framework includes a default object-relational mapping layer (ORM) that can be used to interact with data from various relational databases such as SQLite, PostgreSQL, MySQL, and Oracle.

Django allows us to add, delete, modify, and query objects, using an API called ORM.

An object-relational mapper provides an object-oriented layer between relational databases and object-oriented programming languages without having to write SQL queries.

It maps object attributes to respective table fields. It can also retrieve data in that manner. This makes the whole development process fast and error-free.



In the above image, we have some Python objects and a table with corresponding fields.

The object's attributes are stored in corresponding fields automatically. An ORM will automatically create and store your object data in the database. You don't have to write any SQL for the same.

Class = Table

class property = table field

Instance of a class (object) = row of data in a table

6. Site 2 – Personal Portfolio

We will organize this project in two apps: blog, personal_portfolio (list of all our projects).

Task: Create the apps.

6.1. Creating a Model

https://docs.djangoproject.com/en/5.0/topics/db/models/

To create models for our portfolio app we modify the file **models.py** within the portfolio folder.

We check the available types for the class properties.

If we want to add an **optional field** we use blank=True (by default it is false)

```
from django.db import models

class Project(models.Model):
    title = models.CharField(max_length=100)
    description = models.CharField(max_length=250)
    image = models.ImageField(upload_to='portfolio/images/')
    url = models.URLField(blank=True)
```

To make the system knows about our model (check all the models in the project and check if there is anything new there): **python manage.py makemigrations**

Every time we do changes on our model it is called a migration (migrating to the DB)

To apply those migrations: **python manage.py migrate**

6.2. Access to the models using /admin

To use the database web interface **localhost:8000/admin** we must create a user:

python manage.py createsuperuser (dwcs - abc123.)

python manage.py changepassword dwcs

Then we need to modify the admin.py file of the app that we want to see that app inside of admin:

```
from django.contrib import admin
from .models import Project

admin.site.register(Project)
```

In order to see the image fields of the objects correctly we must modify the personal_portfolio/settings.py so that we indicate the folder for the media files:

```
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

We must modify the urls.py too:

```
from django.contrib import admin
from django.urls import path
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
  path('admin/', admin.site.urls),
```

```
]
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

We can add a register in Projects from the Django Admin web and then see it, including the image.

It is also posible to check the SQLite database using a SQLite Viewer.

6.3. Displaying objects

In views.py we write the code that interacts with the database and the template.

To make a query of all the objects in the Project class:

```
from django.shortcuts import render
from .models import Project

def home(request):
    projects = Project.objects.all()
    return render(request, 'portfolio/home.html', {'projects':projects})
```

To get an query ordered by a specific field in descending order:

```
projects = Project.objects.order_by('-field') [:5]
```

In the template we can access the properties of these objects.

```
{% for project in projects %}
    <h2>{{ project.title }}</h2>
```

6.4. Another set of URLs

When we have several Apps in the project it is better to use several files urls.py.

In the urls.py of our project we add:

```
from django.urls import path, include
urlpatterns = [
    ...
    path('blog/', include('blog.urls')),
]
```

In the urls.py of the blog App we add a new urls.py file that will use the views.py within the blog directory:

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

app_name = 'blog'

urlpatterns = [
    path('', views.all_blogs, name='all_blogs'),
]
```

6.5. View the database

Using for example **dbeaver**.

6.6. Connecting our App to a different database

In settings.py, in DATABASES, we indicate the details of the database.

6.7. Static files

We've termed **media files** as things that users can upload to our system. They are dynamic contents in our pages and they can change if they are changed in the database.

Static files are the files inserted in our pages when we design them. They are always the same. If we add a custom CSS it will be a static file. The pdf with the resumé is a static file.

In settings.py we already have the **STATIC_URL** variable defined.

We can define the **STATIC_ROOT**:

```
STATIC_URL = 'static/'
STATIC_ROOT = BASE_DIR / 'static'
```

We create the 'static' folder in the App folder. Within that folder I create another one with the name of my App. It's the same way as we had done it with 'templates'.

We copy the static files to that directory.

To show those files in a template we must add a 'load static' command at the top of the file somewhere and then we'll be able to use the static files:

```
{% load static %}
<a href="{% static 'app/resume.pdf' %}">Resume</a>
<img src="{% static 'app/file.jpg' %}">
<link rel="stylesheet" href="{% static 'app/custom.css' %}">
```

6.8. Details Page

To be able to show a specific register of the database (select by id).

We add in urls.py:

```
app_name = 'blog'

urlpatterns = [
        path('<int:blog_id>/', views.detail, name='detail'),
```

this way, if somebody types an int, it will be passed to views.detail as the value of the blog_id variable. In views.detail we indicate blog_id as a parameter to the function detail:

```
from django.shortcuts import render, get_object_or_404
def detail(request, blog_id):
        blog = get_object_or_404(Blog, pk=blog_id)
        return render(request, 'blog/detail.html', {'blog':blog})
```

A 404 page indicates that it does not exist.

In the template:

```
<a href="{% url 'blog:detail' blog.id %}">{{ blog.title }}</a>
```

6.9. Extending Base Templates

We can create a base template so that the other templates inherit from it. It allows us to define common parts of the pages of our web as for example navigation bars and footers.

In a base templates we write 'blocks' in the places where the templates that inherit from them will have some contents:

{% block content %}{% endblock %}

In the other templates we indicate that we use a base template:

{% extends 'portfolio/base.html' %}

{% block content %}

All the content of the block

{% endblock %}

6.10. Django Templates Reference

https://docs.djangoproject.com/en/5.1/topics/templates/

Variables

A variable outputs a value from the context, which is a dict-like object mapping keys to values.

Variables are surrounded by {{ and }}

Tags

Tags provide arbitrary logic in the rendering process.

This definition is deliberately vague. For example, a tag can output content, serve as a control structure e.g. an “if” statement or a “for” loop, grab content from a database, or even enable access to other template tags.

Tags are surrounded by {% and %}

Filters

Filters transform the values of variables and tag arguments.

They look like this:

{{ django|title }}

e.g.

```
{{ blog.date|date:'F jS Y' }}
{{ blog.description|safe }}
{{ blog.description|striptags|truncatechars:100 }}
{{ blog.date|date:'M d Y'|upper }}
```

Comments

Comments look like this: {# this won't be rendered #}

A {% comment %} tag provides multi-line comments.

6.11. Some tips

Lorem Ipsum generator: https://www.webfx.com/tools/lorem-ipsum-generator/

Google Fonts: https://fonts.google.com/

Starter template from BootStrap: https://getbootstrap.com/docs/5.0/getting-started/introduction/

W3 Schools: https://www.w3schools.com/bootstrap5/bootstrap_get_started.php

7. Deployment – Pushing your Site Live

http://zappycode.com/pa

pythonanywhere.com

Your website being hosted, deployed, is having it running somewhere. In this case it's going to run on Python Anywhere servers.

It is very easy to do it with git.

We create a new repository in GitHub. We keep it public. We upload our project there.

In PythonAnywhere (PA) we open a bash console.

From GitHub we copy the "Clone with HTTPS" URL. In the PA console we execute "git clone" using that URL. Now we have all the code from our GitHub repository in PA.

We go to the project directory.

We're going to use a Virtual Environment. A Virtual Environment allows us to create a space for different projects that may have different requirements. For example, in a given virtual environment we can specify the version of Python that we want to use, all the packages we want from PIP installed there.

To create a new virtual environment called "portfoliovenv":

mkvirtualenv --python=/usr/bin/python3.10 portfoliovenv

When we are inside this virtual environment we will see (portfoliovenv) at the beginning of the commant prompt.

pip install django

To go out of our virtual environment: **deactivate**

When we go back to a new bash console, in order to go to our virtual environment we will have to type: **workon portfoliovenv**

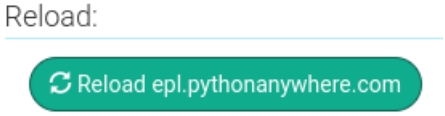
To see all the virtual environments that we have we go to de directory .virtualenvs and ls

We install our libraries: **pip install django pillow**

We need to go to the directory of our project. The name of the project is the name of the directory that conatins the settings.py

On a new tab of the browser we open up Python Anywhere. We click on "**Web**" on the menu bar on top. We "**Add a new web app**". It gives us a free domain. On the next screen we select "**Manual Configuration**". We select the Python version. Then it goes to the dashboard for our site.

Everytime we make changes we click on Reload.



We must set the path to our virtual environment. We just need to write the name of it.

We change the source code to the right path (we copy it from the console, pwd).

URL	Directory
/static/	/home/epl/personalPortfolio/static
/media/	/home/epl/personalPortfolio/media

We also update the working directory with the same information.

We click on the WSGI file. We delete everything but the Django section. We uncomment it. We indicate the correct path. We substitute mysite with the name of the directory that has the settings file. We save this file. Go to the previous page and "Reload".

Clicking on "Go to directory" we can access the files of our app.

Modifying the settings.py file:

- We add to ALLOWED_HOSTS the name of our site.
- We change the parameter "DEBUG" to false.
- Copy the contents of MEDIA_ROOT parameter creating the STATIC_ROOT = BASE_DIR / 'static'. In a deployed app we need to have all the static images. in one central location . We'll make a folder in the base directory called 'static'.

  python manage.py collectstatic --> It collects all the static files and folders and puts them in one single location, in the colation specified in the settings.py file.
- In PythonAnywhere in "Static Files" we need to specify the URL: /static/ with the path.

"Reload" and check the web

In PythonAnywhere

gitignore.io to copy some code and create a .gitignore on the project

We create the .gitignore file in our project directory with the contents:

```
*.log
*.pot
*.pyc
__pycache__/
local_settings.py
/static/
```

We execute:

```
git rm -r --cached .
git add .
git commit -m "Added .gitignore"
git push origin
```

In our computer we need to check that there aren't any changes in our project, we can execute '*git stash*' to delete them. Then we execute: '*git pull origin*'

Deploying changes

In our computer:

We add personal_portfolio/local_settings.py so that we can indicate specific settings for our project in development. In the settings.py file we add some code so that it uses local_settings.py if this file exists.

```
try:
    from .local_settings import *
except ImportError:
    print("Looks like no local file. You must be on production")
```

And we add the changes that we like in our local_settings file:

```
DEBUG = True
ALLOWED_HOSTS = []
```

To check whether we've made some changes via pip:

```
pip freeze > requirements.txt
```

git add . git commit ... git push origin

In PythonAnywhere:

```
git pull origin
```

To install all the python libraries that are installed in our local computer:

```
pip install -r requirements.txt
```

To execute posible migrations:

```
python manage.py migrate
```

To collect any new static files:

```
python manage.py collectstatic
```

Go to the web tab and "Reload www..."

Your Custom Domain

We need to buy a domain, e.g. a Google domain.

In PythonAnywhere we need to get a paid account, e.g. the "Hacker account" which is the cheapest one.

If you have a domain called e.g. domain.info you can indicate www.domain.info for your web site. We need to create it in the site where we have bought our domain, e.g. Google Domains.

8. References.

Udemy Course: Django 3 – Full Stack Websites with Python Web Development, Nick Walter

https://docs.docker.com/samples/django/

https://github.com/docker/awesome-compose/tree/master/official-documentation-samples/django

https://docs.djangoproject.com/en/5.0/

https://docs.djangoproject.com/en/5.0/topics/db/models/

https://certisured.com/blogs/importance-of-orm-in-django-frameworks