Web Application Development with Python (Django, Flask) and SQLite

Using SQLite instead of MySQL or PostgreSql or Oracle is simpler because SQLite is a serverless, self-contained, and zero-configuration database engine. Here’s how to modify the previous steps to use SQLite instead of MySQL for both Django and Flask.

**1. Setting Up SQLite**

SQLite does not require any setup or installation apart from having Python installed, as it is included with Python's standard library.

**2. Developing with Django and SQLite**

**Step 1: Create a Django Project**

1. Create a new Django project:

django-admin startproject myproject

cd myproject

**Step 2: Configure Database Settings**

Edit myproject/settings.py to configure SQLite:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

**Step 3: Create a Django App**

1. Create a new app:

python manage.py startapp myapp

1. Add the app to INSTALLED\_APPS in myproject/settings.py:

INSTALLED\_APPS = [

...

'myapp',

]

**Step 4: Define Models**

Edit myapp/models.py to define your models:

from django.db import models

class Item(models.Model):

name = models.CharField(max\_length=100)

description = models.TextField()

price = models.DecimalField(max\_digits=10, decimal\_places=2)

def \_\_str\_\_(self):

return self.name

**Step 5: Create and Apply Migrations**

1. Create migrations:

bash

Copy code

python manage.py makemigrations

1. Apply migrations:

bash

Copy code

python manage.py migrate

**Step 6: Create Views and Templates**

1. Define views in myapp/views.py:

python

Copy code

from django.shortcuts import render

from .models import Item

def item\_list(request):

items = Item.objects.all()

return render(request, 'item\_list.html', {'items': items})

1. Configure URLs in myapp/urls.py:

python

Copy code

from django.urls import path

from . import views

urlpatterns = [

path('', views.item\_list, name='item\_list'),

]

1. Include app URLs in myproject/urls.py:

python

Copy code

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

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

path('items/', include('myapp.urls')),

]

1. Create a template in myapp/templates/item\_list.html:

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>Item List</title>

</head>

<body>

<h1>Items</h1>

<ul>

{% for item in items %}

<li>{{ item.name }}: ${{ item.price }}</li>

{% endfor %}

</ul>

</body>

</html>

**Step 7: Run the Server**

bash

Copy code

python manage.py runserver

Access the application at http://localhost:8000/items/.

**3. Developing with Flask and SQLite**

**Step 1: Create a Flask Project**

1. Create a new directory for your project and navigate into it:

bash

Copy code

mkdir myflaskproject

cd myflaskproject

1. Create and activate a virtual environment (optional but recommended):

bash

Copy code

python -m venv venv

source venv/bin/activate # On Windows, use `venv\Scripts\activate`

**Step 2: Install Flask and SQLite Packages**

bash

Copy code

pip install flask

**Step 3: Create the Flask Application**

1. Create a file named app.py:

python

Copy code

from flask import Flask, render\_template, request, g

import sqlite3

app = Flask(\_\_name\_\_)

DATABASE = 'database.db'

def get\_db():

db = getattr(g, '\_database', None)

if db is None:

db = g.\_database = sqlite3.connect(DATABASE)

return db

@app.teardown\_appcontext

def close\_connection(exception):

db = getattr(g, '\_database', None)

if db is not None:

db.close()

@app.route('/')

def index():

cur = get\_db().cursor()

cur.execute('SELECT \* FROM items')

items = cur.fetchall()

return render\_template('index.html', items=items)

if \_\_name\_\_ == '\_\_main\_\_':

with app.app\_context():

db = get\_db()

db.execute('''CREATE TABLE IF NOT EXISTS items

(id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

description TEXT,

price REAL NOT NULL)''')

db.commit()

app.run(debug=True)

**Step 4: Create Templates**

1. Create a directory named templates in the project root.
2. Create a file named index.html in the templates directory:

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>Item List</title>

</head>

<body>

<h1>Items</h1>

<ul>

{% for item in items %}

<li>{{ item[1] }}: ${{ item[3] }}</li>

{% endfor %}

</ul>

</body>

</html>

**Step 5: Run the Flask Server**

bash

Copy code

python app.py

Access the application at http://localhost:5000/.

**Additional Steps (Optional)**

1. **Handling Forms and CRUD Operations**: Add routes and views to handle form submissions and CRUD operations.
2. **User Authentication**: Implement user authentication using libraries like Flask-Login (Flask) or the built-in authentication system (Django).
3. **Frontend Integration**: Use frontend frameworks/libraries like Bootstrap, React, or Vue.js for a better user interface.
4. **Deployment**: Deploy your application using platforms like Heroku, AWS, or DigitalOcean. Configure your web server (e.g., Gunicorn for Flask or Django).

By following these steps, you can set up and develop a web application using Python with Django or Flask and SQLite as your database. This setup is simpler and suitable for small to medium-sized applications or for development and testing environments.

**Standard Practice: Using settings.py**

By default, Django stores database configuration details in the settings.py file:

python

Copy code

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'mydatabase',

'USER': 'myuser',

'PASSWORD': 'mypassword',

'HOST': 'localhost',

'PORT': '5432',

}

}

This approach works for local development and small projects, but it exposes sensitive information and is not suitable for production environments.

**Best Practices**

**1. Using Environment Variables**

Store sensitive information in environment variables and retrieve them in settings.py. This approach keeps your credentials out of version control and makes it easier to manage different configurations for different environments (development, staging, production).

**Example:**

python

Copy code

import os

from pathlib import Path

# Set the base directory

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

# Use environment variables for database configuration

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': os.getenv('DB\_NAME'),

'USER': os.getenv('DB\_USER'),

'PASSWORD': os.getenv('DB\_PASSWORD'),

'HOST': os.getenv('DB\_HOST', 'localhost'),

'PORT': os.getenv('DB\_PORT', '5432'),

}

}

Then, set the environment variables in your operating system or in a .env file (used with tools like django-environ).

**2. Using a .env File with django-environ**

The django-environ package makes it easy to use environment variables. You can store your environment variables in a .env file and load them in your settings.py.

**Installation:**

bash

Copy code

pip install django-environ

**Example .env file:**

makefile

Copy code

DB\_NAME=mydatabase

DB\_USER=myuser

DB\_PASSWORD=mypassword

DB\_HOST=localhost

DB\_PORT=5432

**Example settings.py:**

python

Copy code

import environ

# Initialize environment variables

env = environ.Env()

environ.Env.read\_env(os.path.join(BASE\_DIR, '.env'))

# Database configuration

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': env('DB\_NAME'),

'USER': env('DB\_USER'),

'PASSWORD': env('DB\_PASSWORD'),

'HOST': env('DB\_HOST'),

'PORT': env('DB\_PORT'),

}

}

**3. Using Django Secrets Management**

For more advanced scenarios, consider using a secrets management service such as AWS Secrets Manager, Azure Key Vault, or Google Cloud Secret Manager. These services allow you to securely store and access sensitive information.

**Example using AWS Secrets Manager:**

python

Copy code

import boto3

from botocore.exceptions import NoCredentialsError, PartialCredentialsError

def get\_secret(secret\_name):

try:

client = boto3.client('secretsmanager')

response = client.get\_secret\_value(SecretId=secret\_name)

secret = response['SecretString']

return json.loads(secret)

except (NoCredentialsError, PartialCredentialsError) as e:

raise Exception("AWS credentials not found") from e

# Fetch secrets from AWS Secrets Manager

secrets = get\_secret('my\_secret\_name')

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': secrets['DB\_NAME'],

'USER': secrets['DB\_USER'],

'PASSWORD': secrets['DB\_PASSWORD'],

'HOST': secrets['DB\_HOST'],

'PORT': secrets['DB\_PORT'],

}

}

**Summary**

* **Standard Practice**: Place database details in settings.py. Suitable for local development.
* **Best Practice**: Use environment variables or a .env file for storing sensitive information.
* **Advanced Practice**: Use secrets management services for enhanced security in production environments.

These practices help keep sensitive information secure and make it easier to manage configurations for different environments.