Django Framework

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It was designed to help developers take applications from concept to completion as swiftly as possible.

**Key Features of Django**

**1. MTV (Model-Template-View) Architecture**

Django follows the Model-Template-View (MTV) design pattern, which is similar to the Model-View-Controller (MVC) pattern used by other frameworks.

* **Model**: Represents the data structure. It is defined by classes in the models.py file and is typically a representation of database tables.
* **Template**: The HTML that Django renders and serves to the user. Templates are written using the Django Template Language (DTL).
* **View**: The business logic layer. Views retrieve data from models and pass it to templates. Views are typically defined in the views.py file.

**2. ORM (Object-Relational Mapping)**

Django includes a powerful ORM that allows developers to interact with databases using Python code instead of SQL. This makes database operations more intuitive and less error-prone.

**3. Admin Interface**

Django automatically generates a professional-looking and customizable admin interface from your models. This interface can be used to manage the website's data.

**4. Form Handling**

Django provides a robust form handling system. It includes form validation, form rendering, and protection against common web attacks like Cross-Site Request Forgery (CSRF).

**5. Security**

Django includes many security features by default, such as protection against SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and clickjacking.

**6. Scalability**

Django is designed to handle high-traffic sites and can scale to handle increased demand with minimal changes to your code.

**7. Community and Documentation**

Django has a large, active community and comprehensive documentation, making it easier to find solutions and get support.

**Detailed Explanation of Django Components**

**1. Project Structure**

A typical Django project is organized as follows:

myproject/

manage.py

myproject/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

myapp/

\_\_init\_\_.py

admin.py

apps.py

models.py

tests.py

views.py

migrations/

\_\_init\_\_.py

**2. Creating a Django Project and App**

* To create a new Django project:

django-admin startproject myproject

cd myproject

* To create a new app within the project:

python manage.py startapp myapp

**3. Settings Configuration**

settings.py contains all the configuration for your Django project. Key settings include:

* **DEBUG**: Turn debugging on or off.
* **DATABASES**: Database configuration.
* **INSTALLED\_APPS**: A list of all Django applications that are activated in this project.
* **MIDDLEWARE**: A list of middleware classes to use.
* **TEMPLATES**: Configuration for the template engine.
* **STATIC\_URL**: URL to use when referring to static files.

**4. Models**

Models are Python classes that define the structure of your database tables.

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

**5. Create Forms**

Create forms for creating and updating items using Django’s forms framework.

# myapp/forms.py

from django import forms

from .models import MyModel

class MyModelForm(forms.ModelForm):

class Meta:

model = MyModel

fields = ['name', 'description']

**6. Update Views**

Update your views to handle CRUD operations.

# myapp/views.py

from django.shortcuts import render, redirect, get\_object\_or\_404

from .models import MyModel

from .forms import MyModelForm

def item\_list(request):

items = MyModel.objects.all()

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

def item\_detail(request, pk):

item = get\_object\_or\_404(MyModel, pk=pk)

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

def item\_create(request):

if request.method == 'POST':

form = MyModelForm(request.POST)

if form.is\_valid():

form.save()

return redirect('item\_list')

else:

form = MyModelForm()

return render(request, 'item\_form.html', {'form': form})

def item\_update(request, pk):

item = get\_object\_or\_404(MyModel, pk=pk)

if request.method == 'POST':

form = MyModelForm(request.POST, instance=item)

if form.is\_valid():

form.save()

return redirect('item\_list')

else:

form = MyModelForm(instance=item)

return render(request, 'item\_form.html', {'form': form})

def item\_delete(request, pk):

item = get\_object\_or\_404(MyModel, pk=pk)

if request.method == 'POST':

item.delete()

return redirect('item\_list')

return render(request, 'item\_confirm\_delete.html', {'item': item})

**7. Update URLs**

Add URL patterns for the new views.

# myapp/urls.py

from django.urls import path

from . import views

urlpatterns = [

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

path('item/<int:pk>/', views.item\_detail, name='item\_detail'),

path('item/new/', views.item\_create, name='item\_create'),

path('item/<int:pk>/edit/', views.item\_update, name='item\_update'),

path('item/<int:pk>/delete/', views.item\_delete, name='item\_delete'),

]

**8. Templates**

Templates are used to render HTML. They can include dynamic content provided by the views.

* **Item List (item\_list.html)**:

<!-- templates/item\_list.html -->

<h1>Item List</h1>

<a href="{% url 'item\_create' %}">Create New Item</a>

<ul>

{% for item in items %}

<li>

<a href="{% url 'item\_detail' item.pk %}">{{ item.name }}</a>

<a href="{% url 'item\_update' item.pk %}">Edit</a>

<a href="{% url 'item\_delete' item.pk %}" onclick="return confirm('Are you sure?');">Delete</a>

</li>

{% endfor %}

</ul>

* **Item Detail (item\_detail.html)**:

<!-- templates/item\_detail.html -->

<h1>{{ item.name }}</h1>

<p>{{ item.description }}</p>

<a href="{% url 'item\_update' item.pk %}">Edit</a>

<a href="{% url 'item\_delete' item.pk %}">Delete</a>

<a href="{% url 'item\_list' %}">Back to List</a>

* **Item Form (item\_form.html)**:

<!-- templates/item\_form.html -->

<h1>{% if form.instance.pk %}Edit{% else %}Create{% endif %} Item</h1>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Save</button>

</form>

<a href="{% url 'item\_list' %}">Cancel</a>

* **Item Delete Confirmation (item\_confirm\_delete.html)**:

<!-- templates/item\_confirm\_delete.html -->

<h1>Confirm Delete</h1>

<p>Are you sure you want to delete "{{ item.name }}"?</p>

<form method="post">

{% csrf\_token %}

<button type="submit">Confirm</button>

</form>

<a href="{% url 'item\_list' %}">Cancel</a>

**9. Run Migrations**

Don’t forget to run migrations if you’ve made changes to the models:

python manage.py makemigrations

python manage.py migrate

**10. Run the Server**

Start the development server:

python manage.py runserver

With these steps, you’ll have a fully functional Django application with Create, Read, Update, and Delete (CRUD) capabilities, along with a form to manage items.

**9. Admin Interface**

The admin interface allows you to manage your data through a web-based interface. To register your models with the admin interface:

from django.contrib import admin

from .models import Item

admin.site.register(Item)

**10. Middleware**

Middleware is a way to process requests globally before they reach the view or after the view has processed them. Common uses for middleware include authentication, logging, and session management.

**11. Testing**

Django provides a testing framework for writing unit tests.

from django.test import TestCase

from .models import Item

class ItemModelTest(TestCase):

def test\_string\_representation(self):

item = Item(name="Sample Item")

self.assertEqual(str(item), item.name)

1. **Create superuser id for admin site**

If no login id was previously created, a new superuser could be created by running the following command by adding email and password.

python manage.py createsuperuser

1. **Run the development server:**

python manage.py runserver

By following these steps, you can set up a basic Django project, create and manage models, handle user requests through views, and render HTML using templates. This structure makes Django a powerful and flexible framework for building web applications.

# Using PostgreSQL as database server

Using PostgreSQL with Django involves several steps, including installing the necessary packages, configuring your Django project to use PostgreSQL, and running migrations to set up your database schema. Here's a step-by-step guide to help you get started:

**1. Install PostgreSQL**

First, you need to install PostgreSQL. You can download and install PostgreSQL from the [official website](https://www.postgresql.org/download/) or use a package manager suitable for your operating system.

**2. Install psycopg2**

psycopg2 is the PostgreSQL adapter for Python that Django uses to connect to a PostgreSQL database.

Install it using pip:

pip install psycopg2-binary

**3. Configure Django to Use PostgreSQL**

Update your Django project's settings to use PostgreSQL as the database backend. Open your settings.py file and configure the DATABASES setting:

# settings.py

DATABASES = {

'default': {

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

'NAME': 'your\_database\_name',

'USER': 'your\_database\_user',

'PASSWORD': 'your\_database\_password',

'HOST': 'localhost', # Set to 'localhost' if your database server is on the same machine

'PORT': '5432', # Default port for PostgreSQL

}

}

Replace 'your\_database\_name', 'your\_database\_user', and 'your\_database\_password' with your actual database name, user, and password.

**4. Create the PostgreSQL Database and User**

You need to create the database and the user in PostgreSQL if you haven't already. You can do this using the PostgreSQL interactive terminal (psql) or a GUI tool like pgAdmin.

Here are the basic commands to create a database and a user:

1. **Open psql**:

sudo -u postgres psql

1. **Create a new database**:

CREATE DATABASE your\_database\_name;

1. **Create a new user**:

CREATE USER your\_database\_user WITH PASSWORD 'your\_database\_password';

1. **Grant privileges to the user**:

GRANT ALL PRIVILEGES ON DATABASE your\_database\_name TO your\_database\_user;

**5. Run Migrations**

After configuring your settings and creating the database, you need to run Django's migrations to set up your database schema:

python manage.py makemigrations

python manage.py migrate

**6. Test the Configuration**

Start the Django development server and ensure everything is working correctly:

python manage.py runserver

Visit the server (usually at http://127.0.0.1:8000/) and make sure your application can connect to the PostgreSQL database.

**7. Optional: Use Environment Variables for Security**

For better security, you can use environment variables to store your database credentials. This prevents sensitive information from being hardcoded in your settings.py file.

1. **Install python-decouple**:

pip install python-decouple

1. **Create a .env file** at the root of your project:

DB\_NAME=your\_database\_name

DB\_USER=your\_database\_user

DB\_PASSWORD=your\_database\_password

DB\_HOST=localhost

DB\_PORT=5432

1. **Update settings.py** to use these environment variables:

# settings.py

from decouple import config

DATABASES = {

'default': {

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

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

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

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

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

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

}

}

By following these steps, you should be able to set up and use PostgreSQL with your Django project successfully.