Django Workshop

In this workshop, we will be using Django to create the backend for a single page to-do web app.

What is Django? 🤔

Django is a web application framework written in Python. It helps speed up the development of websites that interact with databases.

Let’s go ahead and set everything up first.

For Mac/Linux:

```
# Open Terminal
# Navigate to the directory where you would like to create the project directory
cd path/to/directory
# Create the project directory
mkdir todo-app
# Navigate to the project directory
cd todo-app
# Install virtualenv
pip install virtualenv
# Create a new virtual environment called djangoenv
python -m venv djangoenv
# Acivate the virtual environment
source djangoenv/bin/activate
# Install Django inside the virtual environment
python -m pip install Django
# Create a subdirectory and navigate to it.
mkdir todo-be
cd todo-be
# Create a new Django Project called myproject
django-admin startproject myproject
# Navigate to the Django project
cd myproject
# Start running the development server.
python manage.py runserver
```

For Windows:

```
# Open Terminal
# Navigate to the directory where you would like to create the project directory
cd path/to/directory
# Create the project directory
mkdir todo-app
# Navigate to the project directory
cd todo-app
# Install virtualenv
pip install virtualenv
# Create a new virtual environment called djangoenv
python -m venv djangoenv
# Acivate the virtual environment in PowerShell
.\djangoenv\Scripts\activate
# Install Django inside the virtual environment
python -m pip install Django
# Create a subdirectory and navigate to it.
mkdir todo-be
cd todo-be
# Create a new Django Project called myproject
django-admin startproject myproject
# Navigate to the Django project
cd myproject
# Start running the development server.
python manage.py runserver
```

Now let’s create an app. ✍️

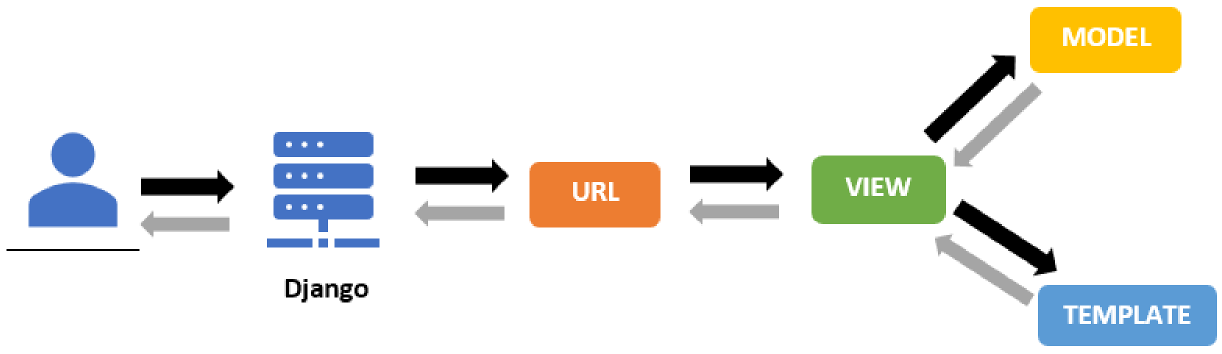
```
python manage.py startapp myapp
```

The terminology may seem confusing. In Django, “project” refers to the entire application, while an “app” is a building block in your project that handles a specific task. We will see this clarified more when we take a look at the architecture of Django.

Note that a single project can contain multiple apps and an app can be used in different projects.

Awesome! With everything set up, let’s take a look at how Django is organized.

Django follows a Model-View-Template (MVT) Architecture.



Templates define what we see - you can think of this as the frontend.

Models act as the interface for maintaining data. They define the structure to store data in our database, with each one mapping to a single database table.

- Each attribute represents a database field
- Note that by default, Django uses SQLite as a database.

Views interact with the model to process data requested by user and returns a response. This could result in rendering templates to display all or a portion of the data to the user.

How does this all come together?

As the user, we use Django framework to write code that handles how we want the templates, models and view to work together. That is, we define how we want to receive the data and handle the data.

Now that we have a clearer understanding of the architecture of Django, let's start creating the backend of our to-do app.

Open a separate terminal window and navigate to your project directory. Make sure the virtual environment is activated.

1. Add `myapp` to `INSTALLED_APPS` In `myproject/settings.py`

   This lets Django know that we are creating a new app.

2. Let's create a model to handle how we want to define tasks in our to-do app.

   - Go to `myapp/models.py`

```
class Task(models.Model):
    title = models.CharField(max_length=120, help_text="Enter task title")
    description = models.TextField(help_text="Enter task description")
    completed = models.BooleanField(default=False)

    def __str__(self):
        return self.title
```

3. In terminal, make migrations to confirm the changes to our database.

```
python manage.py makemigrations myapp
python manage.py migrate myapp
```

4. Register the model. This lets Django know that we are making changes to our models i.e. the structure of our database is changing.
   - Go to `myapp/admin.py`

```
from .models import Task

class TaskAdmin(admin.ModelAdmin):
    list_display = ('title', 'description', 'completed')

admin.site.register(Task, TaskAdmin)
```

5. In terminal, create a superuser to access admin interface and see the database.

```
python manage.py superuser
```

6. Restart server and go to http://localhost:8000/admin
   - Log in and try creating a task. In just a few more steps, we will build and use our API to see that this task is in our database.

7. Since we will be creating APIs later, let's install two packages in terminal. These will allow us to build Web APIs that use REST framework and interact with different web browsers.

```
pip install djangorestframework django-cors-headers
```

Add `rest_framework` and `corsheaders` to `INSTALLED_APPS` in `myproject/settings.py`

In the same file under `MIDDLEWARE`, add `corsheaders.middleware.CorsMiddleware`

8. Create a serializer to convert model instances to JSON. In general, serializers allow us to handle data of different formats.

    - Create `serializer.py` file in `myapp`

```
from rest_framework import serializers
from .models import Task

class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = ('id', 'title', 'description', 'completed')
```

Let's create our first API! 🧑‍💻

```
from rest_framework.response import Response
from rest_framework.decorators import api_view
from rest_framework import status
from .serializers import TaskSerializer
from .models import Task

# Create your views here.
@api_view(['GET', 'POST'])
def tasks(request):
    # Get all tasks
    if request.method == 'GET':
        data = Task.objects.all()
        serializer = TaskSerializer(data, context={'request': request}, many=True)
        return Response(serializer.data)
    # Create a new task
    elif request.method == 'POST':
        serializer = TaskSerializer(
            data = request.data
        )
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        else:
            return Response(status=status.HTTP_400_BAD_REQUEST, data={
                "errors": serializer.errors})


@api_view(['GET', 'DELETE', 'PATCH'])
def task_details(request, id):
    try:
        task = Task.objects.get(id=id)
    except Task.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    # Get a specific task
    if request.method == 'GET':
        serializer = TaskSerializer(task, context={"request": request})
        return Response(serializer.data)
    # Delete a task
    elif request.method == 'DELETE':
        task.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
    elif request.method == 'PATCH':
        # Update fields
        updated_data = request.data.copy()
        serializer = TaskSerializer(
            task,
            data=updated_data,
            partial=True)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        else:
            return Response(status=status.HTTP_400_BAD_REQUEST, data={
                "errors": serializer.errors})
```

In order to access the API, let's specify the URL path in `myapp/urls.py`

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

urlpatterns = [
    path('', views.tasks, name='tasks'),
    path('<int:id>', views.task_details, name='task-detail'),
]
```

If we restart the server using `python manage.py runserver` in terminal and navigate to http://localhost:8000 in the web browser we should be able to see our tasks.

Congrats! You’ve just coded the backend for a single page to-do app! 😊