TASK

Introduction to Django

Visit our website

Introduction

WELCOME TO THE INTRODUCTION TO DJANGO TASK!

This task will introduce some foundational concepts in back-end Web Development with Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel.

Disclaimer: We've curated the most relevant bits of the official documentation for you and added some additional explanation to make Django as concise and accessible as possible.

WHY DJANGO?

If this is your first venture into Web development with the Python programming language, you're probably wondering why you should learn Django when there are approximately 20 Web development frameworks available in Python. Well, for the most part, there are high-level and low-level frameworks, and Django is the highest level framework, making it easier to build better Web apps more quickly and with less code.

INSTALLING PYTHON

Linux

For Linux use:

```
> sudo apt update

> sudo apt install python3
```

Windows

For Windows, you can get the installer here. Note that you should choose the option "Download Windows installer (64 bit)".

MacOS

For MacOS, you must ensure that you have brew installed. If not, you must first install brew.

First, you must install xcode (if you have not already):

```
> xcode-select --install
```

Then, download homebrew using wget:

```
> curl -O
https://raw.githubusercontent.com/Homebrew/install/master/install.sh
```

Then, either use the bash command to run the installer:

```
> bash install.sh
```

Or, alternatively, you can change the permissions of the **install.sh** file directly to make it runnable. This can be done using chmod:

```
> chmod +x install.sh
>./install.sh
```

Then, you can install Python using brew:

```
> brew install python3
```

SETTING UP VIRTUAL ENVIRONMENTS

If you don't already have Django installed, you can open your command prompt and use pip. Pip stands for 'pip installs packages'.

First, it is common best practice to use a **virtual environment** to help keep different systems separate. To make things easier, use the built-in pip module to install PIP. This is so that you can use pip normally when installing packages. The command for this is:

```
> python -m pip install pip
```

The `-m` option in Python means using an installed module as an executable. The next thing is to install the package to enable virtual environments using:

```
> pip install virtualenv
```

This enables the core virtual environment for pip. In addition, there is a wrapper available for this to enable some handy commands (such as `mkvirtualenv` and `workon`).

For Windows, this can be installed using:

```
> pip install virtualenvwrapper-win
```

For any other operating system (Linux or MacOS), this can be installed using:

```
> pip install virtualenvwrapper
```

Please note, if you get stuck trying to install virtual wrapper, try this:

```
> sudo pip install virtualenvwrapper
```

A NOTE FOR LINUX AND MAC

In order to get your commands working, there are certain commands that need to run when opening your terminal. In Linux, these commands are stored at **~/.bashrc** and in MacOS this is stored in **~/.zshrc** (or **~/.bash_profile** if you are using bash).

For Linux, put the following commands in **open ~/.bashrc**:

```
#Virtualenvwrapper settings:
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
export WORKON_HOME=$HOME/.virtualenvs
export
VIRTUALENVWRAPPER_VIRTUALENV=/home/<username>/.local/bin/virtualenv
source ~/.local/bin/virtualenvwrapper.sh
```

Note: you will need to replace <username> with your login name for the PC.

For MacOS, place the following in your **~/.zshrc**:

```
# Setting PATH for Python 3 installed by brew if it is not present
export PATH=/usr/local/share/python:$PATH

# Configuration for virtualenv
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
export VIRTUALENVWRAPPER_VIRTUALENV=/usr/local/bin/virtualenv
source /usr/local/bin/virtualenvwrapper.sh
```

After pasting these commands in, you will need to restart your computer for the changes to take place.

If the above does not work, try this:

The path that is given after the equal sign might not exist for you in this line:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
```

Run the command `which python3`, which should return your correct path. Paste that in.

You may need to go through the same process for the next two lines, except you will run: `which virtualenv` and `which virtualenvwrapper.sh`

Your code might end up looking like so:

```
# Setting PATH for Python 3 installed by brew if it is not present
export PATH=/usr/local/share/python:$PATH

# Configuration for virtualenv
export WORKON_HOME=$HOME/.virtualenvs
export
VIRTUALENVWRAPPER_PYTHON=/Library/Frameworks/Python.framework/Versions/3
.8/bin/python3
export
VIRTUALENVWRAPPER_VIRTUALENV=/Library/Frameworks/Python.framework/Versio
ns/3.8/bin/virtualenv
source
/Library/Frameworks/Python.framework/Versions/3.8/bin/virtualenvwrapper.
sh
```

Remember to restart your computer for the changes to take place.

INSTALLING DJANGO

Now that we have virtual environments and handy commands set up, we can set up the virtual environment. Let's set up a virtual environment called *my_django* using:

```
> mkvirtualenv my_django
```

This sets up your virtual environment and automatically activates it. This can be seen with `(my_django)` appearing at the beginning of your command line. In future, to activate the virtual environment, simply type in:

```
> workon my_django
```

All pip packages will be installed to this virtual environment, and Python will use packages installed in the virtual environment. Type the following command in your command line and hit enter:

```
> pip install django
```

This should work if you have Python successfully installed, however, if it doesn't work, please feel free to contact a mentor for additional support. Alternatively, you can visit: **https://djangoproject.com/download/** to discover how you can download Django.

Once you have Django installed, you're ready to start your project. However, you'll first have to auto-generate some code that establishes a Django project – a collection of settings for an instance of Django, including database configuration, Django-specific options, and application-specific settings.

Navigate to a directory where you would like to start your web development and open a command window in that directory.

To start your project, type the following command in the command window you just opened:

```
> django-admin startproject hyperion
```

The command will set up a new project called **hyperion** for you. You can name your project however you see fit, but you'll need to avoid naming projects after built-in Python or Django components. In particular, this means you should avoid using names like "django" (which will conflict with Django itself) or "test" (which conflicts with a built-in Python package).

If you have a look at the project you just generated, you will see that it has the following scaffolding:

- **hyperion/:** The outer hyperion/ root directory is just a container for your project. Its name doesn't matter to Django; you can rename it if you want.
- **manage.py:** a command-line utility that lets you interact with this Django project in various ways.

- **hyperion/:** the inner hyperion/ directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. hyperion.urls).
  - **__init__.py:** an empty file that tells Python that this directory should be considered a Python package.
  - **settings.py:** settings/configuration for this Django project. Perhaps one of the most important elements inside this directory is "INSTALLED APPS". (We will get to this shortly). Another thing to note here is your SECRET_KEY - keep this a secret! This is used for session encryption. If someone gains access to this key, they will be able to decrypt the session and act as an admin on your site.
  - **urls.py:** controls what is served, based on url patterns (regular expressions).
  - **wsgi.py:** a Python script used to help run your development server and deploy your project to a production environment.
  - **asgi.py:** this Python script is similar to wsgi.py except it processes requests asynchronously. This enables a more efficient process.

INITIAL PROJECT TEST

Now that you have generated your project, you may want to run an initial test to make sure that it is in working order. Open a command window inside the directory where your manage.py file is located and run the following command:

```
> python manage.py runserver
```

This starts up the Django development server, which is a lightweight web server built entirely in Python. It is included in Django for testing purposes. This means that development is rapid and doesn't involve configuring and restarting the server as most changes are made, until you are ready for production.

Don't worry if you receive a warning about unapplied migrations - it's a helpful alert that will not impact setting up your application at this stage.

Go to **http://127.0.0.1:8000/** via your web browser and you will see a "Welcome to Django" page, which confirms that your installation was successful. If you want the site to run on a different port, then you are welcome to change the runserver command, e.g. for **port 8080** we run:

```
> python manage.py runserver 8080
```

Take note:

To quit the server, you can simply enter CTRL-BREAK (`CTRL + C`) or exit the command window.

CREATING YOUR FIRST WEB APP

A Django project is a collection of configurations and applications that have been put together to make a website. A Django application exists to perform a particular function. A project can contain multiple applications and an application can be in multiple projects. To create an app, we run the following command from within the same directory that our `manage.py` is in:

```
> python manage.py startapp webapp
```

The above command will generate an app called webapp for you. Remember not to choose an app name that may conflict with your project name or with a Python package.

Your project scaffolding should now be adjusted to the following:

- **hyperion/**
    - **manage.py**
    - **hyperion/**
        - **asgi.py**
        - **settings.py**
        - **urls.py**
        - **wsgi.py**
        - **__init__.py**
    - **webapp/**
        - **admin.py:** stores the representation of a model in the admin interface.
        - **apps.py:** contains a registry of installed applications that stores configuration and provides introspection. It also maintains a list of available models.
        - **migrations/**

        - **__init__.py:** this file indicates to the Python interpreter that the directory is a Python package.
    - **models.py:** describes database structures and metadata.
    - **tests.py:** a Python script used to help run your development server and deploy your project to a production environment.
    - **views.py:** handles what the end-user "views" or interacts With.
    - **__init__.py:** this file indicates to the Python interpreter that the directory is a Python package.

Once you have generated a new application, you have to install it. This is easily done by adding the name of your application (`webapp`) to the `INSTALLED_APPS` list in your settings.py file. Once you've installed your application, your `INSTALLED_APPS` list in your **settings.py** file should look like this:

```
# Application definition

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

Notice that `'webapp'` is added at the beginning of the list. It doesn't really matter where in the list you append your app name, just as long as you follow the correct Python syntax.

After installing your application, the next task is to add a URL that will point to your app. To do this, open **urls.py**. This is in the same directory as your settings.py file. Once you have that opened, you will see the following:

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

To the urlpatterns list, append the following item:

```
path('webapp/', include('webapp.urls'))
```

Your urls.py file should now have the following code:

```
from django.contrib import admin
from django.urls import path, include
```

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

Note that `include` was imported from django.urls at the top (you may need to add `include` to your file). The changes we made to **settings.py** and **urls.py** in our configuration directory (hyperion) must be made every time we create a new app.

To summarise:

1. Install the app in your INSTALLED_APPS list in **settings.py**
2. Append a url that will point to your application in **urls.py**

In the configuration directory urls.py file we have referenced another urls.py in **webapp/**. However, if you have a look at the files inside your webapp directory, you will see that a **urls.py** file doesn't exist there. This means that we have to create a new Python file called **urls.py** inside your **webapp/** directory. Once you have created the new urls.py file, copy and paste the following code into it:

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

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

The above code in **webapp/ urls.py** references a function called `index` from **views.py**. This function doesn't exist yet, so let's open **webapp/views.py** and define this function.

```
def index(request):
    return HttpResponse("<h2>Hello World</h2>")
```

You will also need to import `HttpResponse` as follows in the views.py file:

```
from django.http import HttpResponse
```

Our web application now has a view. Start up your development server to see the output:

```
> python manage.py runserver
```

Go to **http://127.0.0.1:8000/webapp/**. You should see the text "Hello World!" as an h2 sized heading.

RENDERING HTML

HTML code can be placed as a parameter of the `HttpResponse` function, however, most of the time our web page will have a little more than just the "Hello World" text, therefore, we have to figure a way in which we can load HTML files into our function. Edit the function in **views.py**:

```
def index(request):

    return render(request, "index.html")
```

Remember to import `render` as follows:

```
from django.shortcuts import render
```

Inside the webapp directory, create a new folder called **templates**. Insert one of the HTML files that you created in the previous tasks in to the **templates** folder and rename it to **index.html**. All HTML templates for your webapp will be stored in the **templates/** directory. Your webapp directory should now have the following scaffolding:

- **webapp/**
    - **migrations/**
    - **templates/**
        - **index.html**
    - **_init_.py**
    - **admin.py**
    - **apps.py**
    - **models.py**
    - **tests.py**
    - **views.py**

Note: the home page of any website is generally named index. However, if you want to load any of the other HTML files you created make sure that you adjust your function accordingly.

Take note:

The home page of any website is generally named index. However, if you want to load any of the other HTML files you created make sure that you adjust your function accordingly.

LOADING STATIC FILES

If you have rendered your HTML files correctly and started up your development server, you will notice that your web pages exclude any stylesheets, images etc.

You will need to load these in by creating a new folder in your webapp directory called **static/**. You then have to place the following code right at the beginning of your relevant HTML file (e.g. index.html) in the **templates/** directory:

```
{% load static %}
```

Also, you have to reference your items accordingly (for example an image tag would be):

```
<img src="{% static 'assets/image.jpg' %}">
```

The **image.jpg** is in another folder in static called assets.

MTV ARCHITECTURE IN DJANGO

Model-template-view (MTV) is also known as model-view-template(MVT) is a software design pattern we use for developing web applications. Each component of the pattern refers to a specific process within the architecture:

- **Model:** refers to the reference source for the database fields and associated behaviour for the data we consume in our application.
- **Template:** refers to the representation of the requested information or data. Templates display as normal HTML, but it also includes DTL (Django Template Language). DTL provides dynamically displayed data.
- **View:** contains the logic that governs the user's request and determines the appropriate response. View doesn't necessarily mean 'display'. View relates our Model and Template together through URLs. The View component helps to determine which information data in our Model should be communicated via our Template.

Some people prefer to think of MTV as **layers**. You could think of each component of the MTV model as a cog in an interlinked process. See below for a clearer picture:

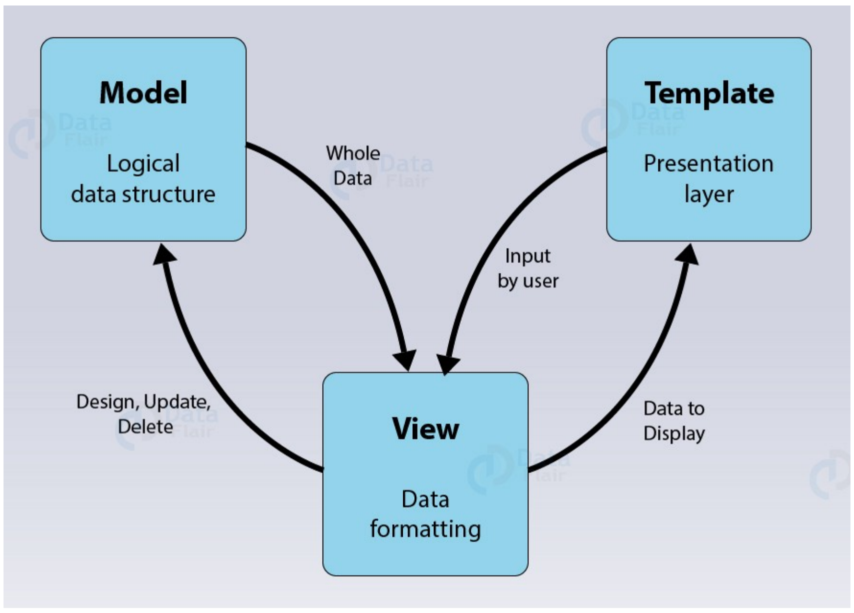

Image source:
https://towardsdatascience.com/working-structure-of-django-mtv-architecture-a741c8c64082

Are things starting to make a little more sense? Let's break it down a bit further:

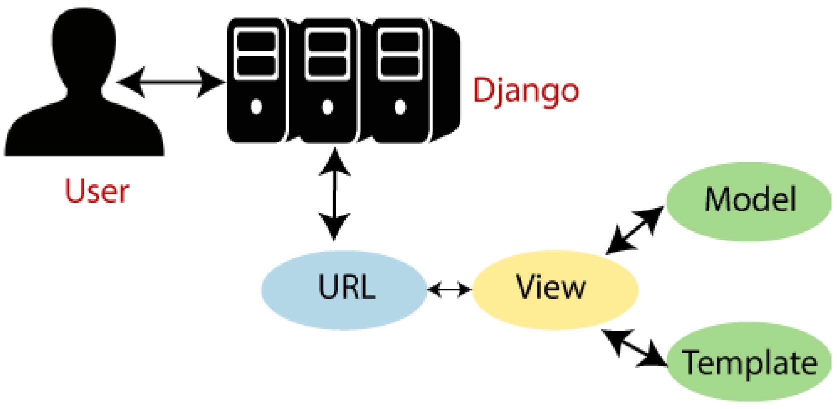

Image source: https://www.tutorialandexample.com/django-mvt

Let's imagine that you, the user, make a request ...

- The Django framework then inspects the content of your URL request. Notice the light blue oval above that reads 'URL' – that's the **URL dispatcher** and it's interrogating your URL request for certain patterns. If it finds a view that matches, it then shows the relevant view.
- Our view then queries the data in our model to decide what is relevant to your URL request.
- View also simultaneously speaks to the template. The template displayed will depend on the content of your URL request. Generally, relevant responses will likely contain some HTML, CSS, or JavaScript, and some Python template helpers.
- Finally views renders the corresponding template as a response to the user.

MTV != MVC

It should be noted that the MTV model is not the same as the Model-View-**Controller** (MVC) model, which you may have encountered previously. The main difference between the two models is that in MVC, the controller (the software responsible for managing the interplay between the template and model) is **not** handled by the framework.

Compulsory Task

Follow these steps:

- Create a new project called mySite.
- Start an application called personal. Our end-goal will be to set up an "about you" dynamic website.
- Create a template for your app that will display some information about you.
- Render your template and map a URL to it.
- Render two additional HTML files that you have previously created and set up URLs for them.
- Install your app, and set up a URL for it.

Completed the task(s)?

Ask an expert code reviewer to review your work!

Review work

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCES

Shadhin, F. (2022, January 6). *The MVT Design Pattern of Django - Python in Plain English*.

https://python.plainenglish.io/the-mvt-design-pattern-of-django-8fd47c61f582

Django documentation. (n.d.). Django Software Foundation. Retrieved October 18, 2022,

from **https://docs.djangoproject.com/en/4.1/**