

# 1 MIDDLEWARE

## 1.1 DEVELOPMENT SETUP

### 1.1.1 SOFTWARE REQUIREMENTS

This development setup guide enables the user to install the required software/packages onto their local machines/ virtual machines to kick start the development.

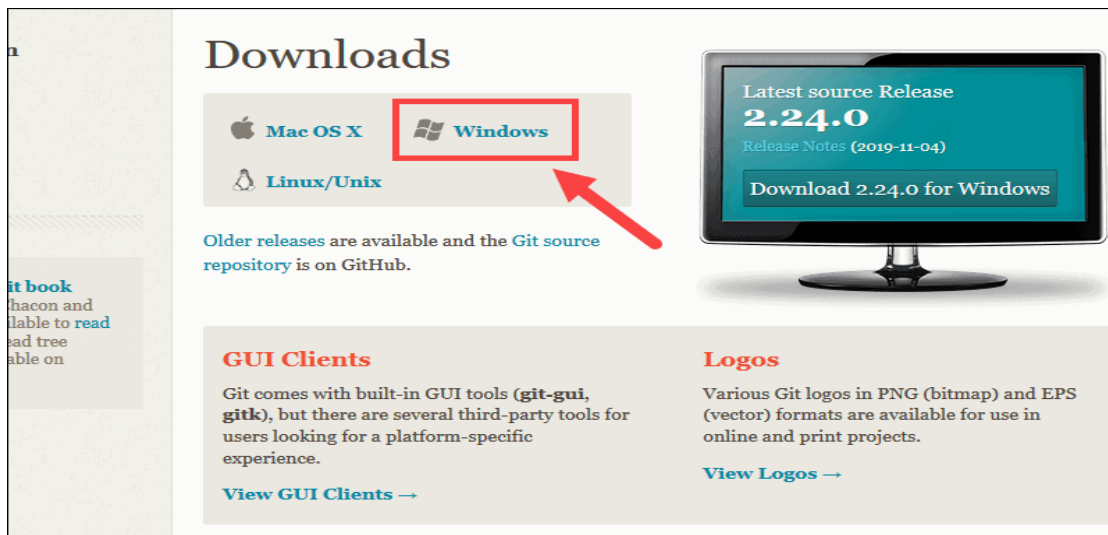
#### 1.1.1.1 GIT INSTALLATION

Git is a widely used open-source software tracking application used to **track projects** across different teams and revision levels.

##### 1.1.1.1.1 INSTALLING GIT FOR WINDOWS

Installing Git prompts you to select a text editor. If you don't have one, we strongly advise you to install prior to installing Git.

- Browse to the official Git website: <https://git-scm.com/downloads>
- Click the download link for Windows and allow the download to complete.

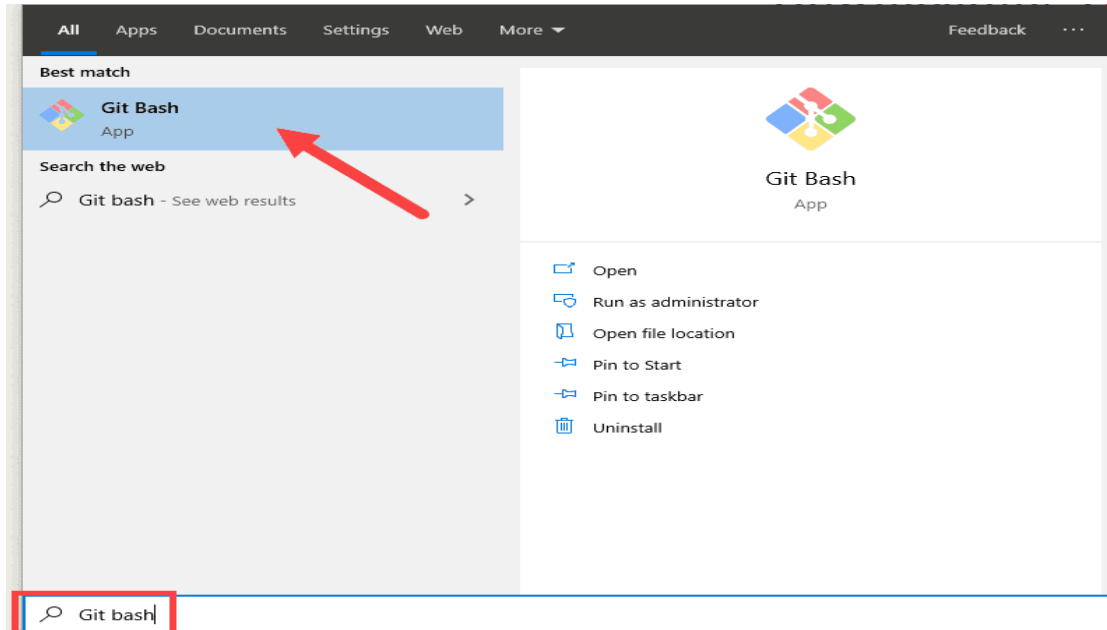


- Browse to the download location (or use the download shortcut in your browser). Double-click the file to extract and launch the installer.

#### 1.1.1.1.2 LAUNCH GIT IN WINDOWS

Git has two modes of use – a **bash scripting shell** (or command line) and a **graphical user interface (GUI)**.

- To launch **Git Bash**, open the **Windows Start** menu, type ***git bash*** and press **Enter** (or click the application icon)



#### 1.1.1.2 PYCHARM INSTALLATION

PyCharm is a cross-platform editor developed by JetBrains. Pycharm provides all the tools you need for **productive Python development**.

##### 1.1.1.2.1 INSTALLING PYCHARM FOR WINDOWS

- To download Pycharm visit the website <https://www.jetbrains.com/pycharm/download/> and Click on “DOWNLOAD” link under the community section.

# Download PyCharm

Windows macOS Linux

## Professional

Full-featured IDE for Python & Web development

DOWNLOAD

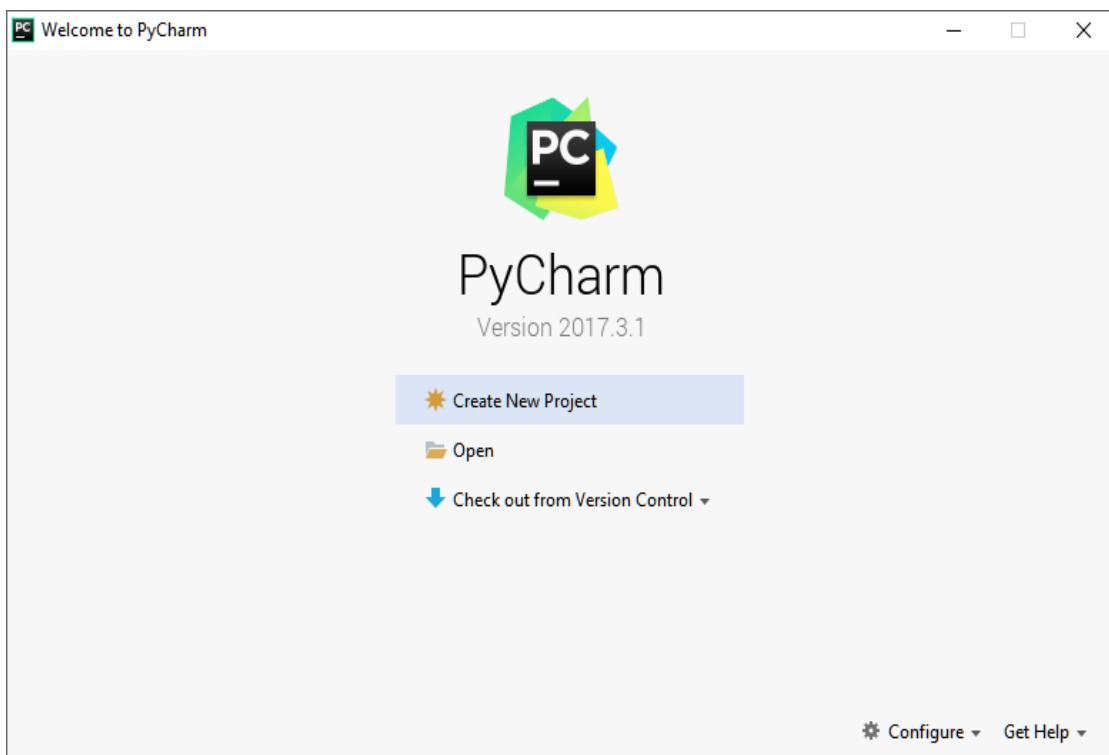
Free trial

## Community

Lightweight IDE for Python & Scientific development

DOWNLOAD

Free, open-source



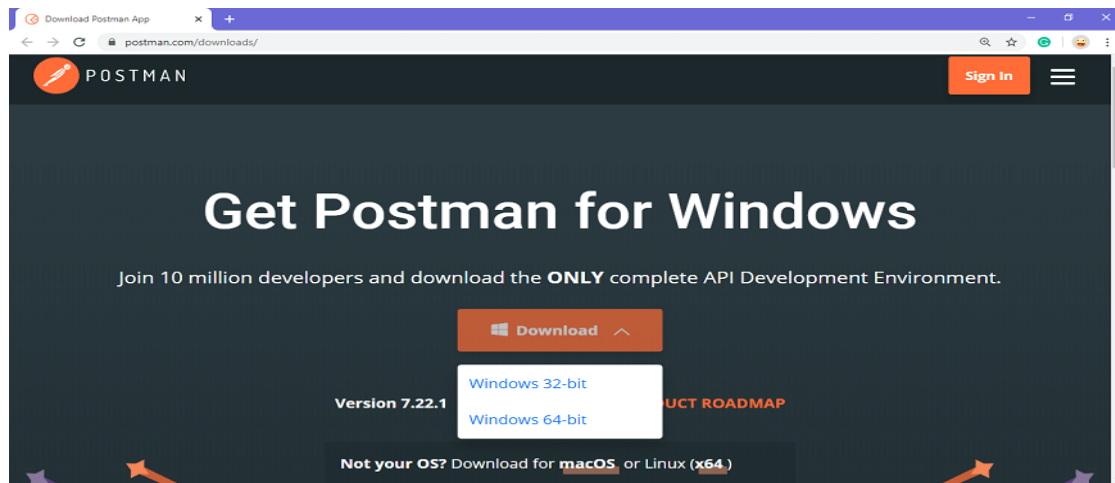
### 1.1.1.3 POSTMAN INSTALLATION

Postman is one of the most popular software testing tools which is used for **API testing**. With the help of this tool, developers can easily create, test, share, and document APIs.

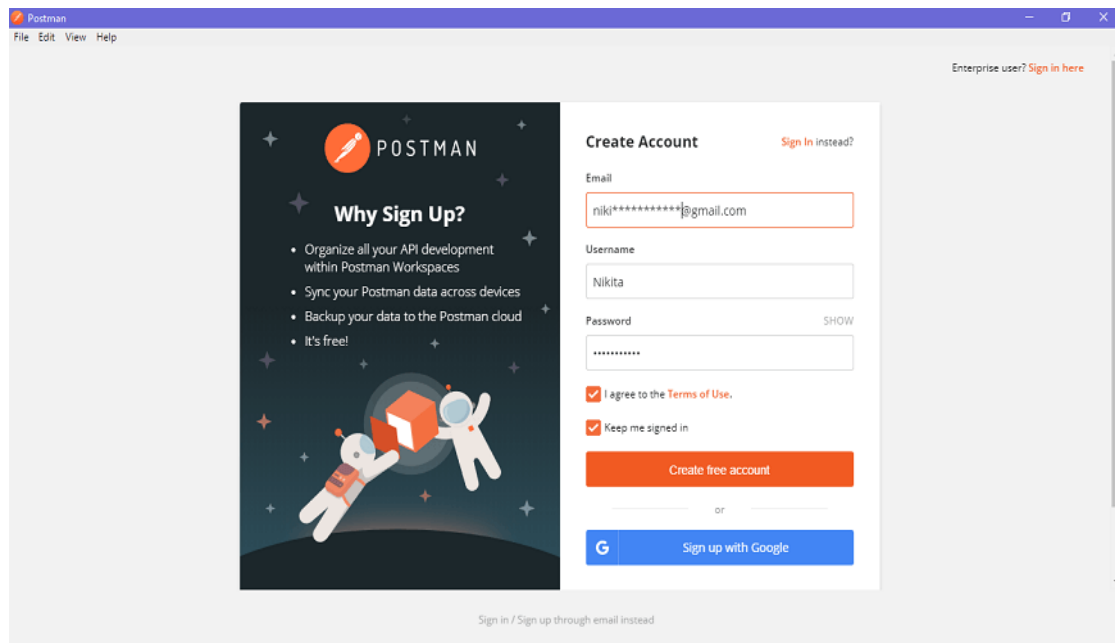
#### 1.1.1.3.1 POSTMAN FOR WINDOWS

- Go to the link <https://www.postman.com/downloads/> and click download for Mac or Windows or Linux based on your operating system.

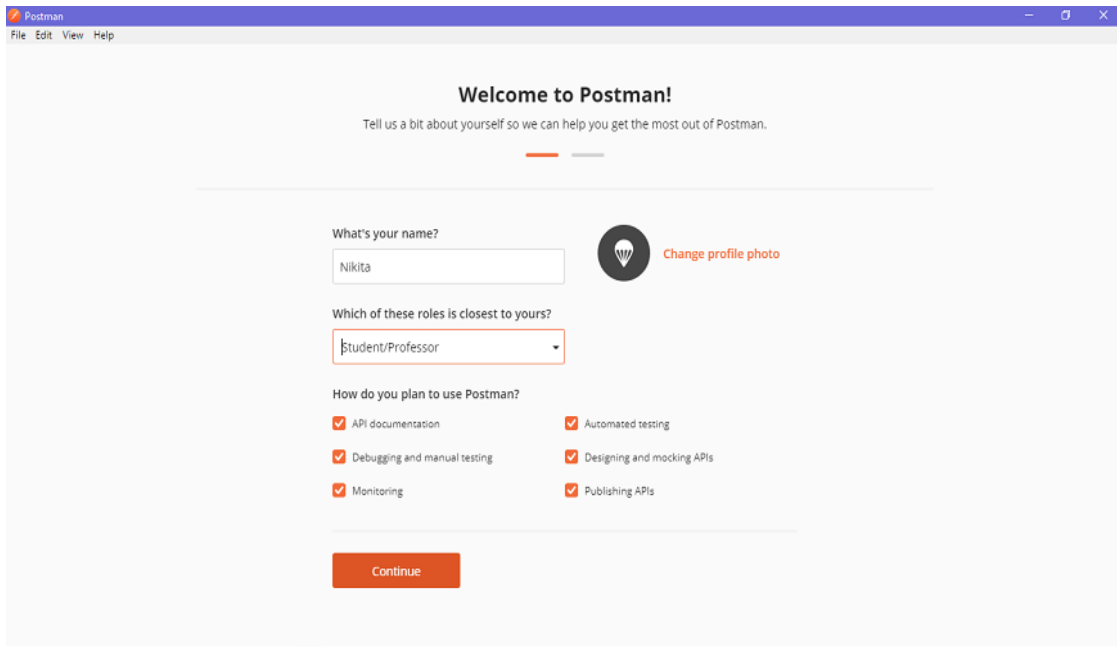
- For downloading the app for Windows, click on the download button and select the version. I opted for the 64-bit version. If you are using a 32-bit OS, you can choose the 32 bit.



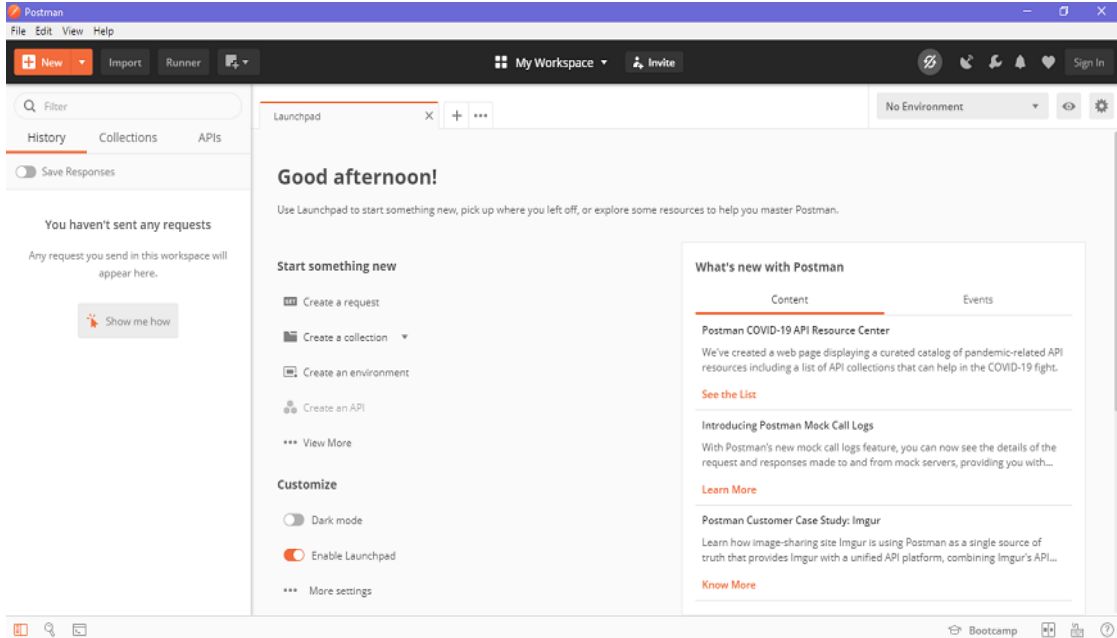
- Create your account with all the required details, or you can also signup with Google, as shown in the image.



- After signing in, select the workspace tools as per your requirement, and then click on, continue to get the start-up screen.



- You will see the following page, and then you are ready to use Postman.

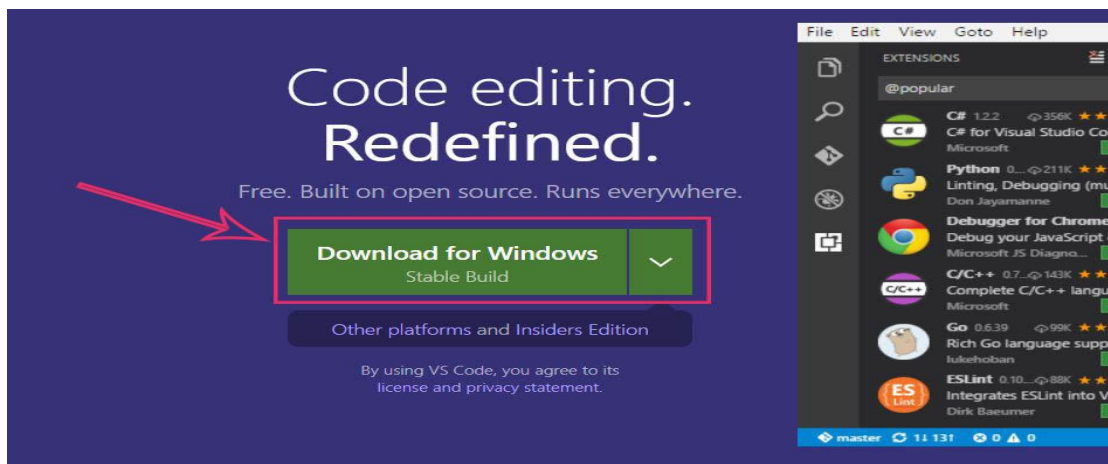


### 1.1.1.4 VS CODE INSTALLATION

Visual Studio Code is a freeware **source-code editor** made by Microsoft for Windows, Linux and macOS.

#### 1.1.1.4.1 INSTALLING VS CODE FOR WINDOWS

- VS code can be simply installed on Windows 10 with the setup file. So, we need to download the setup file from this official website.  
<https://code.visualstudio.com/>
- After opening the website, click on the **Download for Windows** button. This will download the VS code Setup Wizard on our system as an EXE file.



### 1.1.2 ENVIRONMENT SETUP

Before starting any Django project, we need to create a python virtual environment in the system. the main purpose of Python virtual environments is to create an isolated environment for Python projects. This means that each project can have its own dependencies, regardless of what dependencies every other project has. To simplify the virtual environment separates the packages, python version installed etc. from one project to another thereby ensuring that there are no conflicts. To create a python virtual environment in your system follow the steps below:

In your local system create a folder for the project.

- Install **anaconda** or **mini conda** in your system.
- Open the conda command prompt and navigate to the above created folder.
- Create a virtual environment for the project using the following commands:

In the terminal client enter the following where yourenvname is the name you want to call your environment, and replace x.x with the Python version you wish to use.

```
conda create -n yourenvname python=x.x anaconda
```

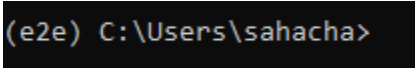
Press y to proceed. This will install the Python version and all the associated anaconda packaged libraries at “path\_to\_your\_anaconda\_location/anaconda/envs/yourenvname”

- Activate your virtual environment using:

```
conda activate yourenvname
```

Activating a conda environment modifies the **PATH** and **shell** variables to point to the specific isolated Python set-up you created. The command prompt will change to indicate which conda environment you are currently in by prepending (**yourenvname**).

Example:



```
(e2e) C:\Users\sahacha>
```

- Deactivate your virtual environment using:

```
conda deactivate
```

Now inside this virtual environment created for the project we can install all the necessary packages relevant to the project.

**In the project we have two environments:**

1. Dev
2. Pre-Production (staging)
3. Production

We create two environment files one for each that stores the necessary information. As shown below:

```

24 export KEY_VAULT_NAME=<"encoded key vault name">;
25 export AZURE_CLIENT_ID=<"client ID">;
26 export AZURE_CLIENT_SECRET=<"client secret">;
27 export AZURE_TENANT_ID=<"tenant ID">;
28
29 export APP_CLIENT_ID=<"application client ID">;
30
31 export SQL_DB_NAME=<"encoded SQL database name"> ;
32 export SQL_DB_SERVER=<"encoded SQL database server"> ;
33 export SQL_DB_USER=<"SQL database user"> ;
34
35 export SQL_PASSWORD=<"azure SQL password">;
36 export SCHEMA="dev";
37 export SCHEMA_ID=<schema ID>;
38
39 export DB_DRIVER=<"driver name for SQL server"> ;
40 DB_NAME=<'database name'> ; export DB_NAME
41 DB_USER=<'database user'> ; export DB_USER
42 DB_PASSWORD=<'database password'>; export DB_PASSWORD
43 DB_HOST=<'database host'>; export DB_HOST
44 DB_PORT=<'port number'> ; export DB_PORT

```

We name the files as:

- **.env\_<project\_name>\_dev** - used for **development** phase
- **.env\_<project\_name>\_staging** - used for **pre-production** phase

In the **settings.py** file as shown in the snippet below, the parameters like SCHEMA, SCHEMA\_ID etc. have different values in the **development** environment and in the **pre-production** environment.

The values of these parameters are fetched from the environment files as per the requirements and the working phase.

For example, if we are in the development environment:

```
SCHEMA = os.environ["SCHEMA"]
```

The variable **SCHEMA** will fetch the value from the **.env<project\_name>\_dev** file.

```

122
123 SCHEMA = os.environ["SCHEMA"]
124 SCHEMA_ID = os.environ["SCHEMA_ID"]
125 TENANT_ID = os.environ["AZURE_TENANT_ID"]
126 CLIENT_ID = os.environ["APP_CLIENT_ID"]
127 SQL_PASSWORD_TO_FETCH = os.environ['SQL_PASSWORD']

```



Similarly, the other parameters will take the values from the environment file depending on the environment the application is running on.

This is called as **parameterisation of credentials** using different environment files i.e., using different values for the parameters from different environment files based on the working environment.

*1.1.2.1*

## 1.2 HOSTING SETUP

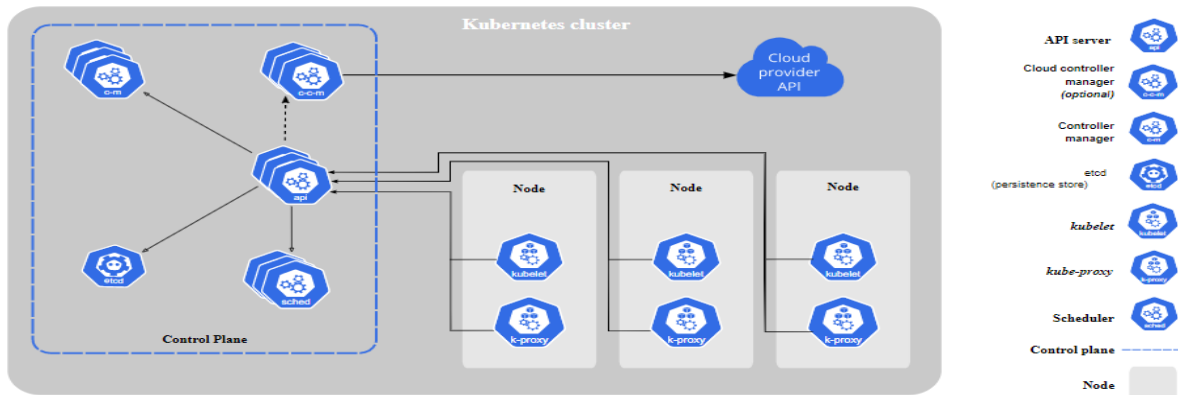
Hosting is required to host and deploy the services like middleware, frontend etc onto a virtual cloud container like Kubernetes

### 3.1.1 KUBERNETES SETUP

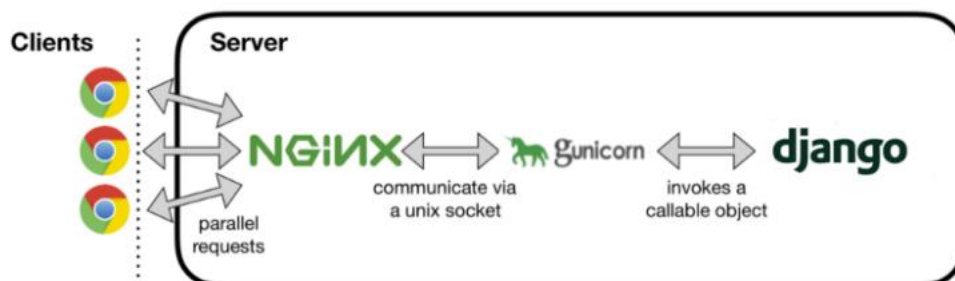
**Kubernetes** is a portable, extensible, open-source platform for managing **containerized workloads and services**, that facilitates both declarative configuration and automation. Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime.

When you **deploy Kubernetes**, you get a **cluster**. A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

Here's the diagram of a Kubernetes cluster with all the components tied together.



### 3.2.1.2 SETUP PRODUCTION ENVIRONMENT IN KUBERNETES:



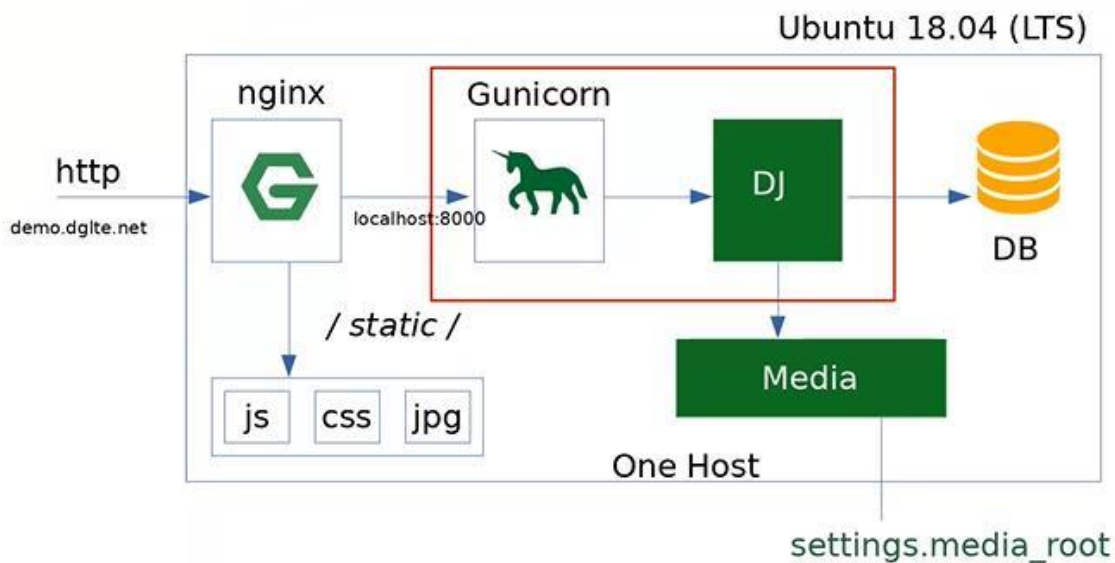
Running a local server of Django is not a recommended way in production because it is just a test server not a production ready server. So, to run Django in production is to run with Gunicorn and use Nginx as a reverse proxy so it gives more security to our application.

Install gunicorn dependency using pip command and also add to the requirements.txt file of the project:

*pip install gunicorn*

- In development phase we run our Django project using runserver command.
- In the settings.py file of the application we have a variable called "DEBUG" which is set to TRUE.

- During the production phase we set this DEBUG value to FALSE to make the application safe and for other security reasons.
- In production phase we need to make our application use the concept of multi-threading and execute the processes in parallel.
- We use nginx as a router and gunicorn as a webserver for the above process using the wsgi.py file in the project.
- Gunicorn will help to run our Django application with multiple workers. We can set the number of workers between 2 and 5.



- Nginx routes the traffic between gunicorn and static. This makes the server fast as it routes the non-static requests to gunicorn and static requests to the static location
- /static/ is the location for a static folder present in the project structure that serves the static files of our application such as images, thumbnails, css etc.
- For nginx to route the traffic we need to write a script in a file named nginx.default that is shown in the snippet below:

```

1 server {
2     location / {
3         proxy_pass http://0.0.0.0:8081;
4     }
5     location /admin/{
6         proxy_pass http://0.0.0.0:8081;
7     }
8     location /swagger_documentation/static/ {
9         root /home/LIA/middleware/django/lcc_performance_tracking/;
10    }
11    location /swagger_documentation/static/admin/css/ {
12        root /home/LIA/middleware/django/lcc_performance_tracking/;
13    }
14 }
  
```

- Suppose our Django application is hosted on 8081 port using gunicorn then all the non-static requests will be routed there as shown in lines 2 to 7 in the above snippet.

- Any request for the static files will be routed to the static folder path as shown in lines 8 to 13.
- This is how the nginx is configured to setup the routing of the requests to out application.

1 We need to make changes in the **.yaml** file for the Kubernetes object we want to create.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: middleware-e2e-dev-deployment
5    namespace: dev
6  spec:
7    selector:
8      matchLabels:
9        app: middleware-e2e-dev
10   template:
11     metadata:
12       labels:
13         app: middleware-e2e-dev
14     spec:
15       imagePullSecrets:
16         - name: regcred
17       containers:
18         - name: middleware-e2e-dev
19           image: <container_name>.azurecr.io/<image_name>:<image_tag>
20           command: ["/bin/bash", "-c"]
21           # args: ["-c", "while true; do echo hello; sleep 1000;done"]
22           args: ["apt-get -y update; apt-get -y install git; mkdir /home/temp; cd /home/temp; git clone
23             #
24             args: ["bash db_req.sh; pip install requirements.txt; source .env e2e_dev
25             nginx start && gunicorn <application_name>.wsgi --bind 0.0.0.0:8081 --workers 3"]
26           ports:
27             - containerPort: 80
28           resources:
29             requests:
30               cpu: 0.2
31               memory: 300Mi
32             limits:
33               cpu: 0.3
34               memory: 500Mi

```

- In line 24 above we see how gunicorn is binded to the Django application on the server port 8081 and the workers are set to 3 i.e., 3 parallel workers will be present.
- The requests by nginx are taken at the 8080 port and are routed to 8081 to gunicorn if the request is non-static else it routes to the static url.
- This reduces the load on the server and the performance of the application becomes better.
- We need to make some changes in the settings.py file of the Django application as shown below:

```

36  STATIC_URL = '/<traffic_path>/swagger_documentation/static/'
37
38  PROJECT_DIR = os.path.dirname(os.path.abspath(__file__))
39  STATICFILES_DIRS = [
40    os.path.join(BASE_DIR, 'swagger_documentation/static/')
41  ]

```

- Make sure to set the DEBUG value as FALSE.
- Set the STATIC\_URL with the traffic\_path as shown in the snippet above. This traffic\_path added in the STATIC\_URL helps the nginx to route the static requests to the correct location in the application. This traffic\_path varies with deployment.
- We also need to make some changes in the docker file as well as shown below:

```

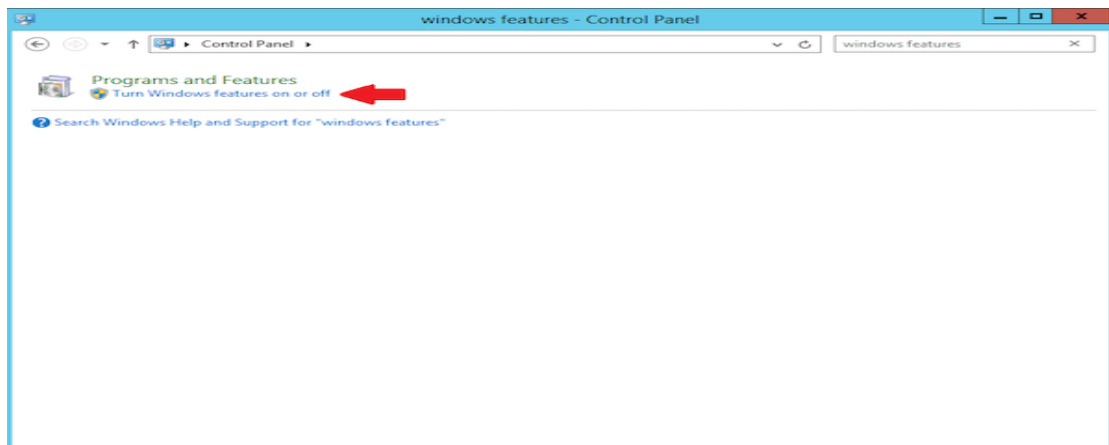
9 RUN apt-get -y update && apt-get install -y nginx
10
11 RUN rm /etc/nginx/sites-enabled/default
12
13 RUN touch /etc/nginx/sites-available/django_project
14
15 RUN ln -s /etc/nginx/sites-available/django_project /etc/nginx/sites-enabled/
16
17 COPY nginx.default /etc/nginx/sites-available/django_project
18
19 RUN sed -i -e 's/\r$//' db_req.sh
20
21 RUN bash db_req.sh

```

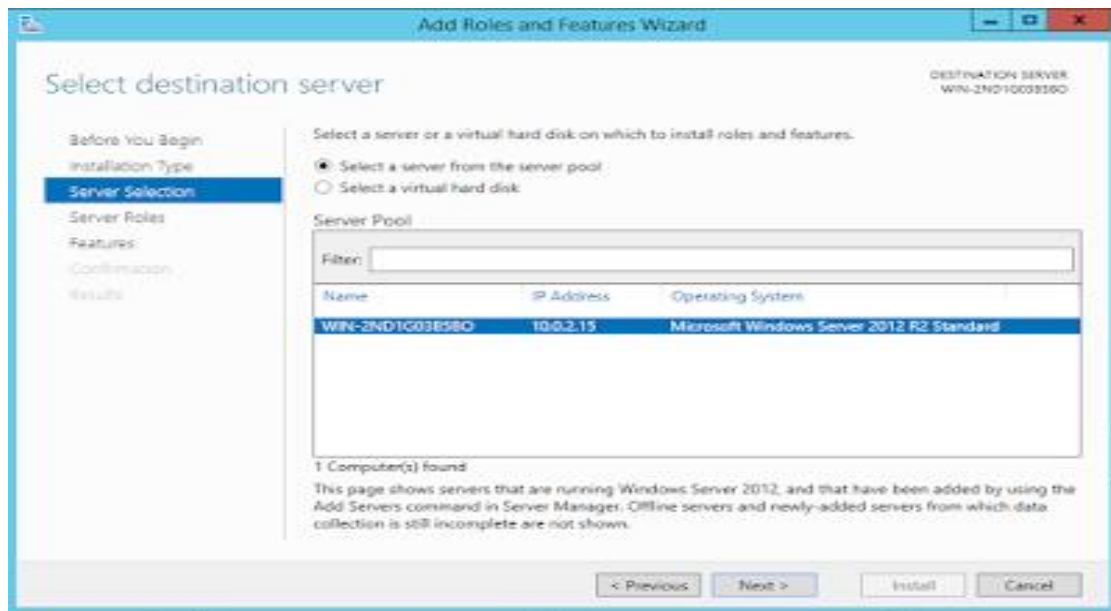
## 3.1.2 WINDOWS IIS (INTERNET INFORMATION SERVICES) SETUP

### 3.1.2.1 *ENABLING IIS:*

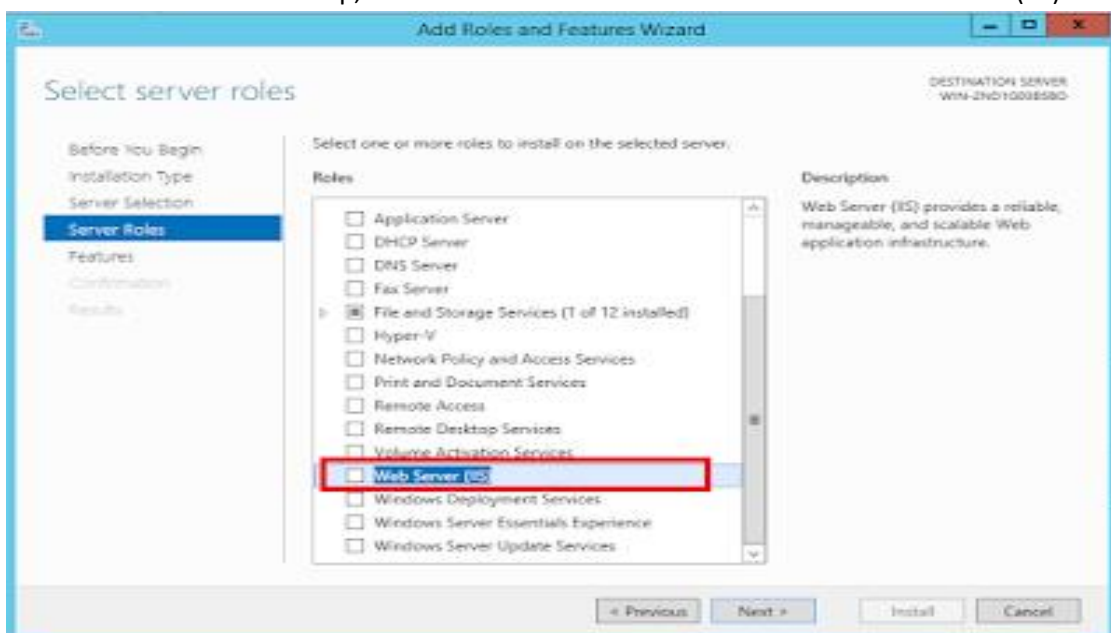
- Open the Control Panel
- In the search box in the top right, type “windows features” (without the quotes)
- In the search results under “Programs and Features” click “Turn Windows features on or off.”  
This launches the Add Roles and Features Wizard.



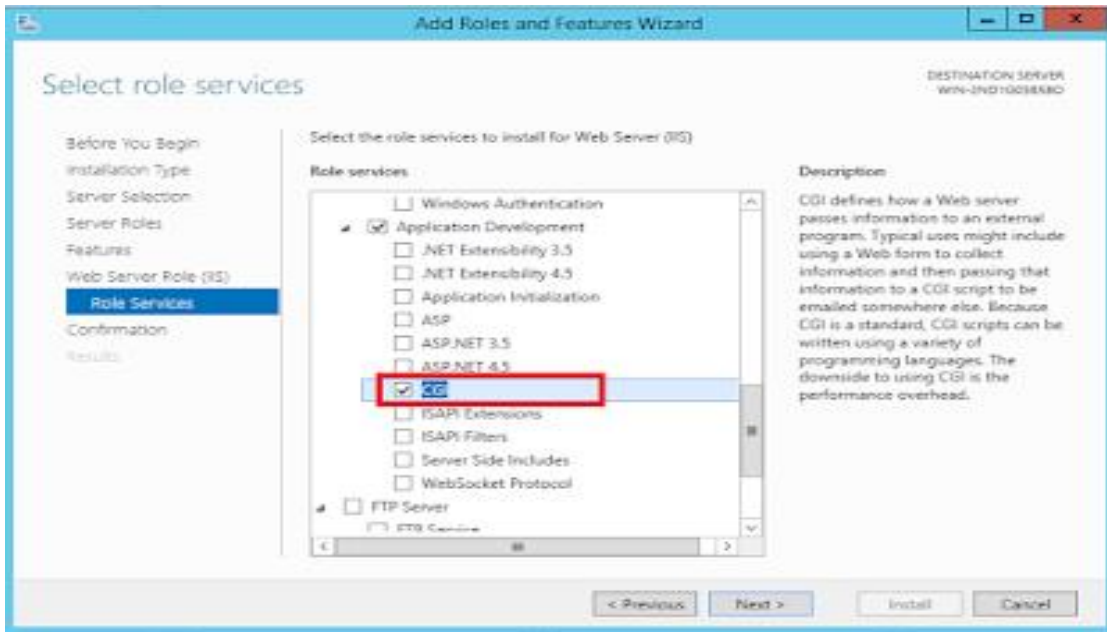
- On the “Before you begin” step, click “Next”
- On the “Installation Type” leave the “Role-based or feature-based installation” radio button selected and click “Next”
- On the “Server Selection” step, leave the current server highlighted and click “Next”



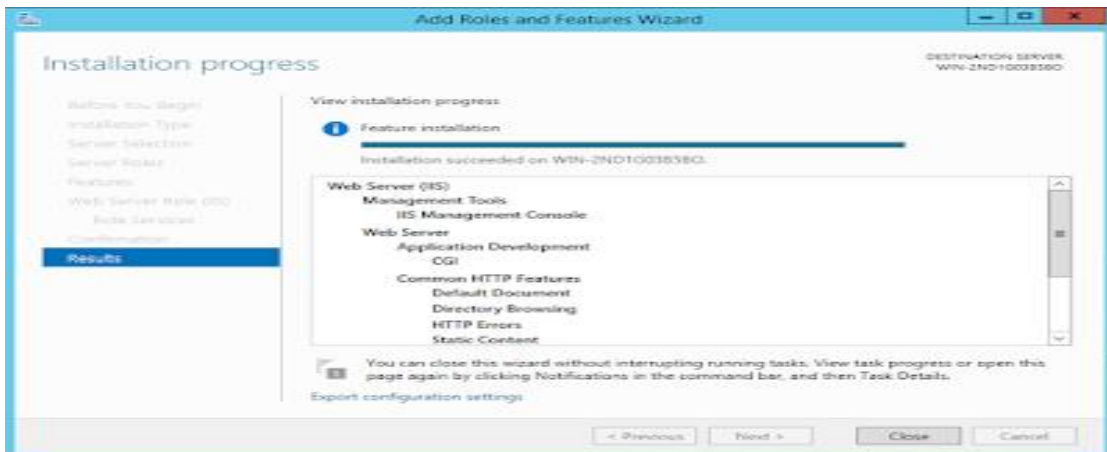
- On the “Server Roles” step, scroll to the bottom of the list and check “Web Server (IIS)”



- In the “Add features that are required for Web Server (IIS)?” dialog that appears, leave the “Include management tools (if applicable)” checkbox checked and click “Add Features”
- On the “Select server roles” step, now that “Web Server (IIS)” is checked, click “Next”
- On the “Select features” step, leave the defaults and click “Next”
- On the “Web Server Role (IIS)” step, click “Next”
- On the “Select role services” step, scroll down to “Application Development,” expand that section, and check the “CGI” box. This will also check the “Application Development” checkbox. With “CGI” checked, click “Next.”



- On the “Confirmation” step, click “Install”
- Once the installation completes, click “Close”



- Close Server Manager
- Close Control Panel

### 3.1.2.2 VERIFYING IIS INSTALLATION:

- Open a web browser on the server
- Enter <http://localhost> in the address bar and press Enter. You should see the default IIS page.



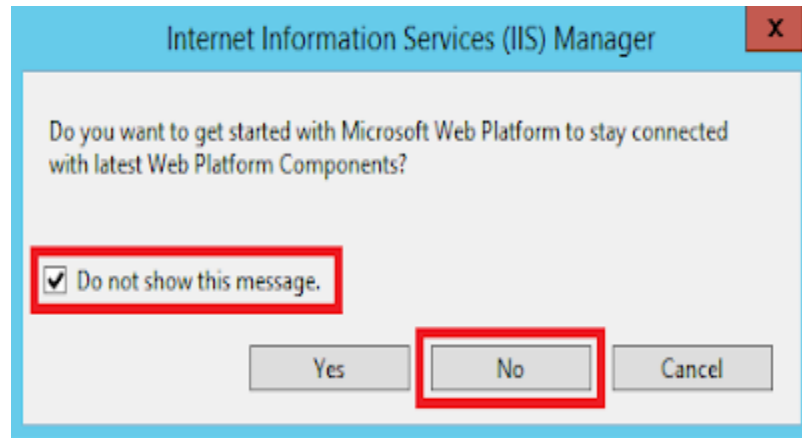
- If you do not see the default IIS page:
  - Open Control Panel
  - Type “services” in the search box
  - Under “Administrative Tools” click “View local services”
  - Scroll to the bottom of the list and ensure you see “World Wide Web Publishing Service” listed, and that the status is “Running”
  -

### 3.1.2.3 *CONFIGURE IIS TO SERVER DJANGO APPLICATIONS:*

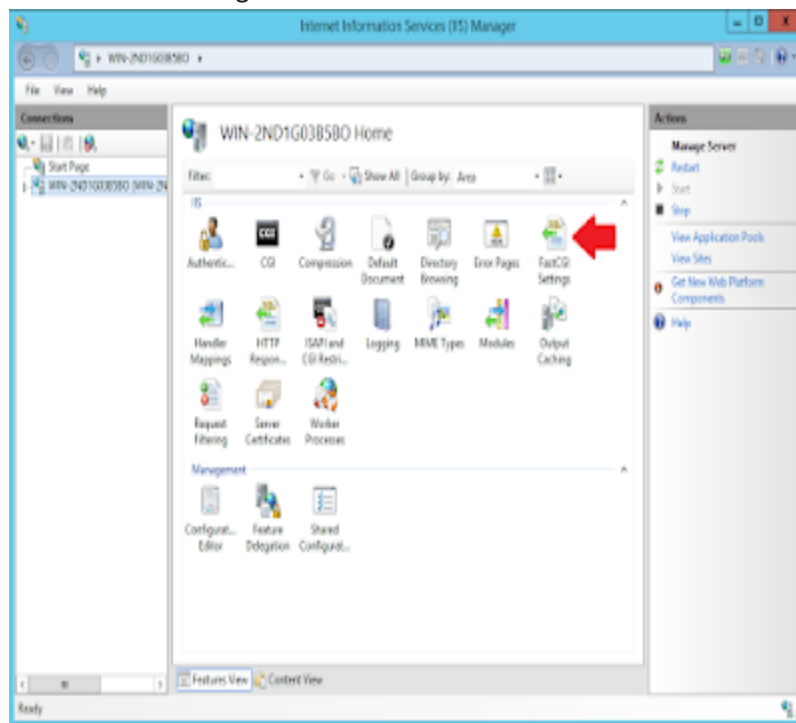
#### 1. **Configure FastCGI in IIS:**

- Open the IIS Manager
  - Click the Windows button
  - Click on Administrative Tools
  - Double-click Internet Information Services (IIS) Manager
- Click on the name of the server in the list on the left. If you see the following dialog box, check the box “Do not show this message” and click “No.” (You can always get to this later if necessary by clicking “Get New Web Platform Components” under “Actions” on the right-hand side of IIS Manager.)

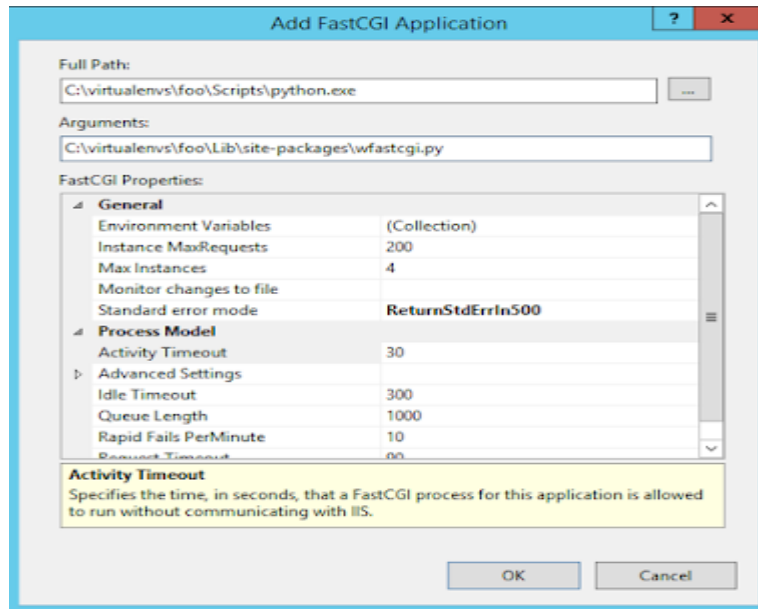




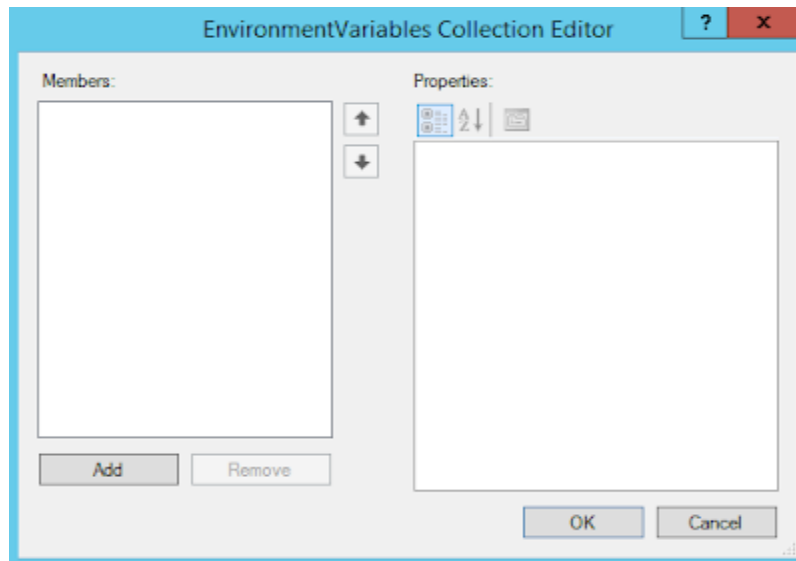
- Double-click the “FastCGI Settings” icon



- Under “Actions” on the right-hand side click “Add application ...”
- In the Add FastCGI Application dialog, in the “Full Path” box, type the path to the Python executable for the application’s virtual environment.
  - C:\virtualenvs\foo\Scripts\python.exe
- In the Arguments input box, type the path to the wfastcgi.py file in the application’s virtual environment.
  - C:\virtualenvs\foo\Lib\site-packages\wfastcgi.py
- At this point your settings in the Add FastCGI Application dialog should look like this:

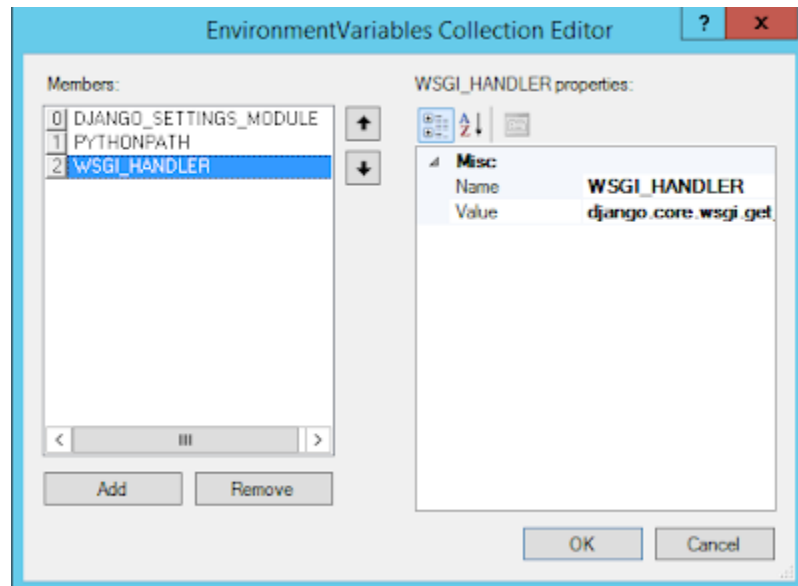


- With the Add FastCGI Application dialog box still open, under the “General” section, click on the “Environment Variables” line, then click the Gray “...” button that appears next to (Collection) on the right-hand side of the line. This opens the Environment Variables Collection Editor dialog.



- In the Environment Variables Collection Editor dialog, click “Add”
- In the “Name properties” section on the right, click the input box to the right of “Name,” remove the “Name” text that is already in the input box, and enter `DJANGO_SETTINGS_MODULE` (note that this MUST be entered in ALL CAPS)
- Click the input box to the right of “Value” and enter `donor_mgmt_system.settings`
- Click “Add” again and enter the following:
  - Name: `PYTHONPATH` Value: `C:\apps\donor_mgmt_system`

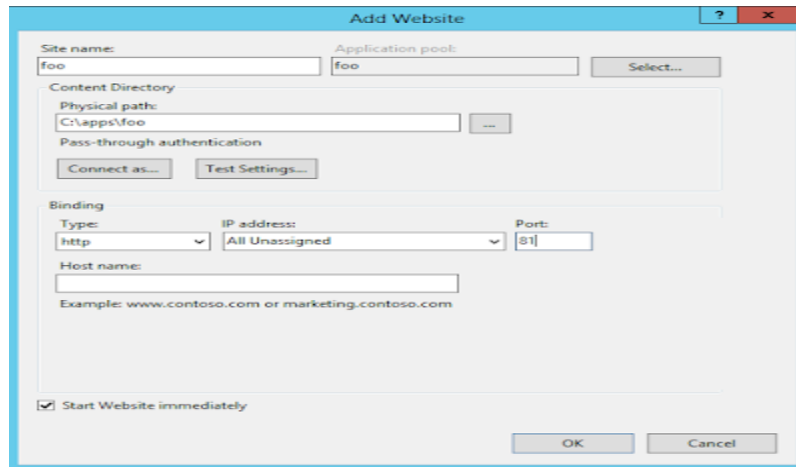
- Click “Add” again and enter the following:
  - Name: `WSGI_HANDLER` Value: `django.core.wsgi.get_wsgi_application()`
- At this point you will have three environment variables:



- NOTE: All these settings are CASE-SENSITIVE. They must be entered with exactly the case indicated here to work.
- Click “OK” to close the Environment Variables Collection Editor
- Click “OK” to close the Add FastCGI Application dialog

## 2. Create and configure new IIS Web Site:

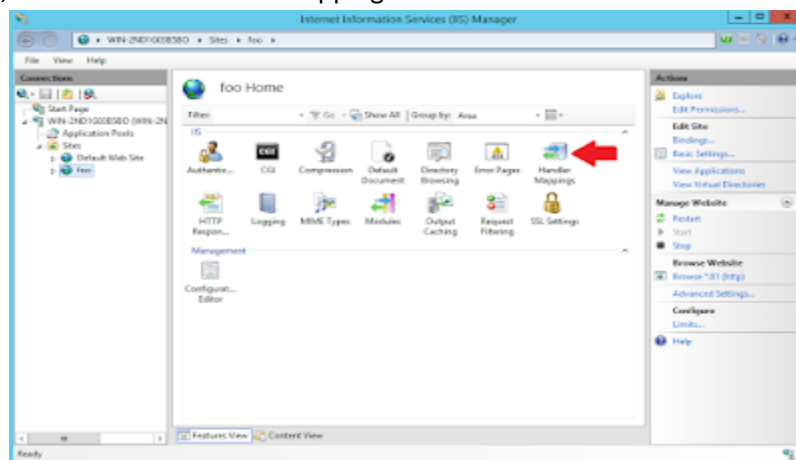
- Open IIS Manager
- On the left-hand side under Connections, expand the tree under the server name by clicking on the arrow to the left of the server name
- Right-click on the Sites folder and click “Add Website ...”
- For the site name enter foo
- For the physical path, type the following: `C:\apps\foo`
- For the purposes of this example configuration, change the Port to 81, since the Default site is running on port 80. For a real-world application you’ll likely want to use name-based virtual hosting by adding bindings and run the site on port 80.
- You may leave the “Host name” blank. At this point the Add Website dialog should look like this:



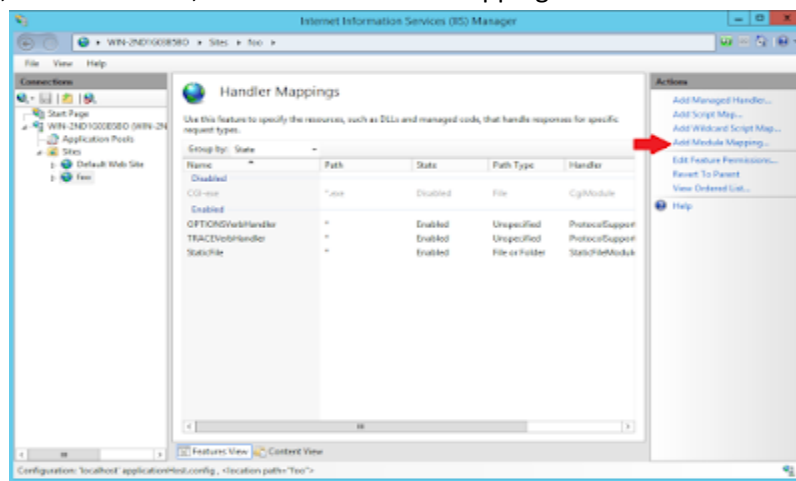
- Click “OK”

### 3. Add a FastCGI handler mapping to this site

- In IIS Manager, expand the Sites folder on the left-hand side and click on the foo site
- On the right, double-click “Handler Mappings”

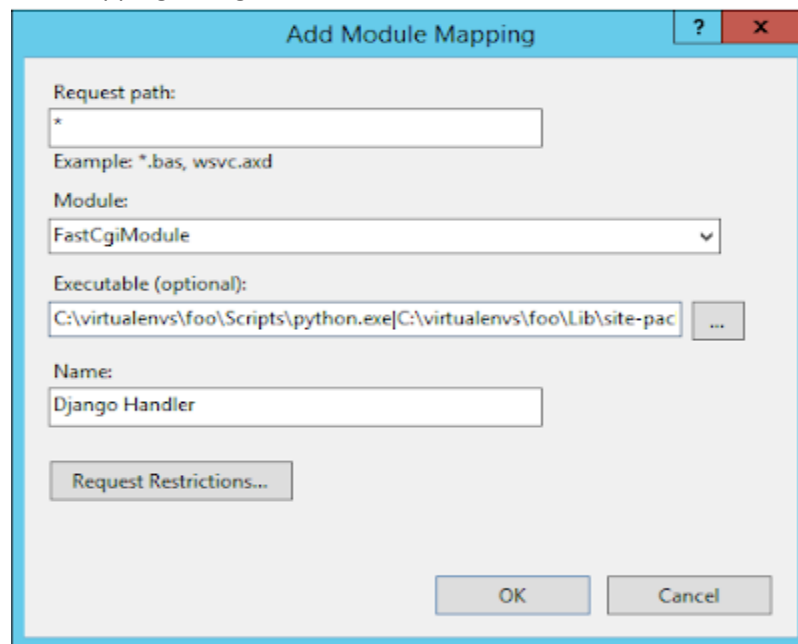


- On the right, under “Actions,” click “Add Module Mapping”

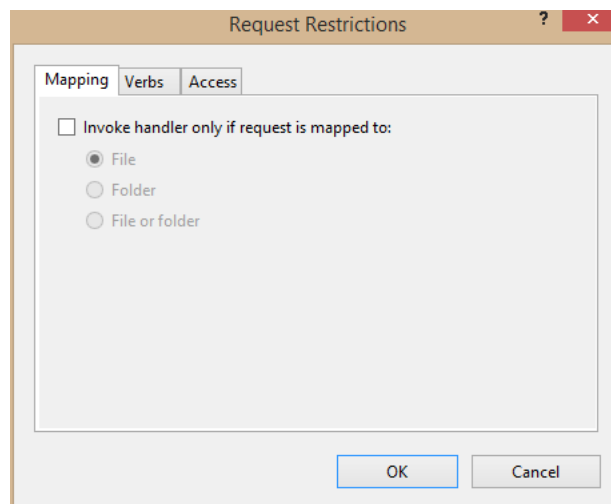


- In the “Request path” box enter an asterisk: \*

- Click the arrow on the right-hand side of the “Module” box and select “FastCGI Module”
- NOTE: Make sure to select **FastCGI Module**, *NOT* CGIModule
- In the “Executable” box, enter the following:
  - C:\virtualenvs\foo\Scripts\python.exe|C:\virtualenvs\foo\Lib\site-packages\wfastcgi.py
- Note that the character after python.exe is a pipe (|), which is entered by pressing Shift-\ on your keyboard
- In the “Name” box, enter Django Handler (you can call this whatever you want; it’s merely a friendly name for the module mapping)
- The Add Module Mapping dialog should now look like this:

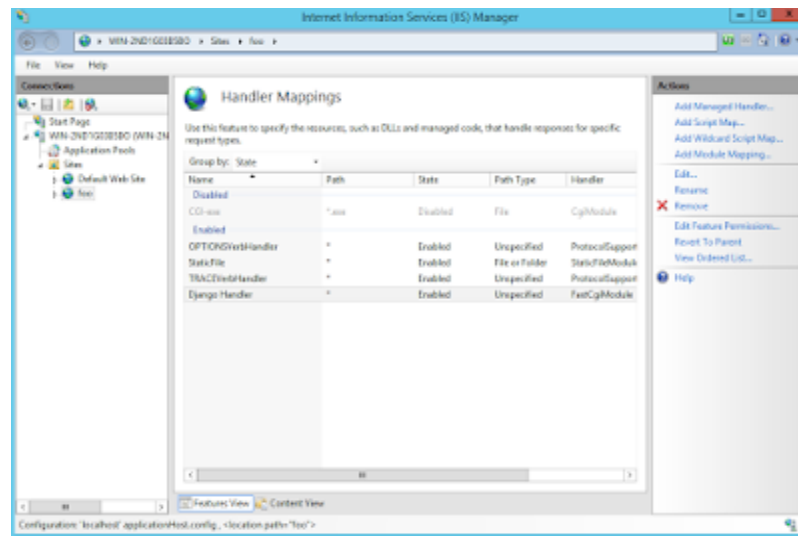


- Click the “Request Restrictions” button and uncheck the “Invoke handler only if request is mapped to:” checkbox.



- Click “OK” to close the Request Restrictions dialog.

- Click “OK” to close the Add Module Mapping dialog.
- When prompted “Do you want to create a FastCGI application for this executable?” click “No” since we created the application earlier.
- Note that you CAN have it create the FastCGI application for you from the module mapping, but the settings seem to be different and the end result isn’t fully editable. I also detailed how to create the FastCGI application to be as thorough as possible with all the various pieces involved.
- You will now see the Django Handler listed in the Handler Mappings for the foo website:



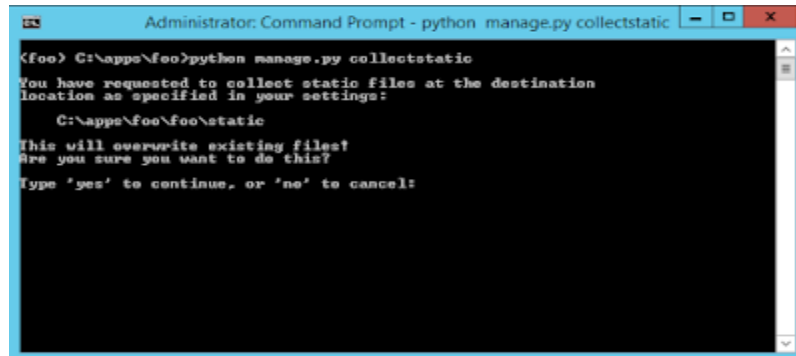
### 3.1.2.4 CONFIGURING STATIC RESOURCES AND MEDIA DIRECTORIES:

#### 1. Configure Django’s settings.py file to tell it where to put the static files

- Using a plain text editor such as Notepad, or Idle if that was installed when you installed Python, open the file `C:\apps\foo\foo\settings.py`
- Scroll to the bottom of the file, or use the find feature of your text editor, and find the `STATIC_URL` setting
- Above the `STATIC_URL` setting, add the following setting
  - `STATIC_ROOT = os.path.abspath(os.path.join(BASE_DIR, 'foo', 'static'))`
- The placement of the `STATIC_ROOT` setting in the settings.py file doesn’t matter but putting it right next to the `STATIC_URL` setting is typical and keeps all the settings related to static files in one place.
- The `STATIC_ROOT` setting we’re using as our example will put the static files in `C:\apps\foo\foo\static`
- Save the settings.py file

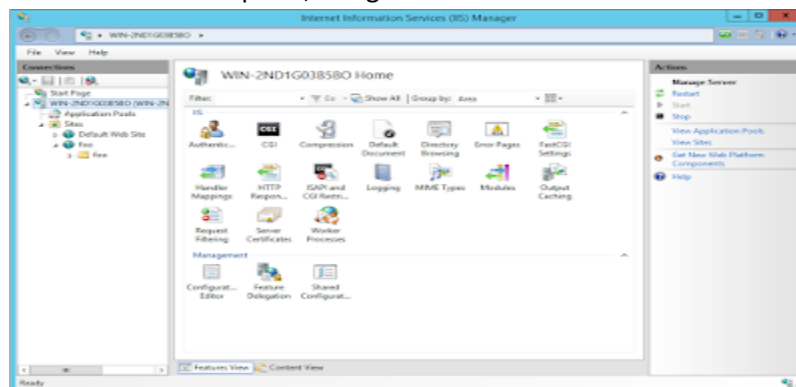
#### 2. Run the collect static Management Command

- Now that Django knows where to put the static files, we can run the `collectstatic` command. This command takes all the static files for your project, including the static files used by the Django admin and other packages in your virtual environment, and puts them under one static directory, the location of which is the `STATIC_ROOT` setting we added above, so they can be served by the web server.
- `python manage.py collectstatic`
- Type yes at the prompt where Django asks you to confirm copying the the files into the `STATIC_ROOT` directory.

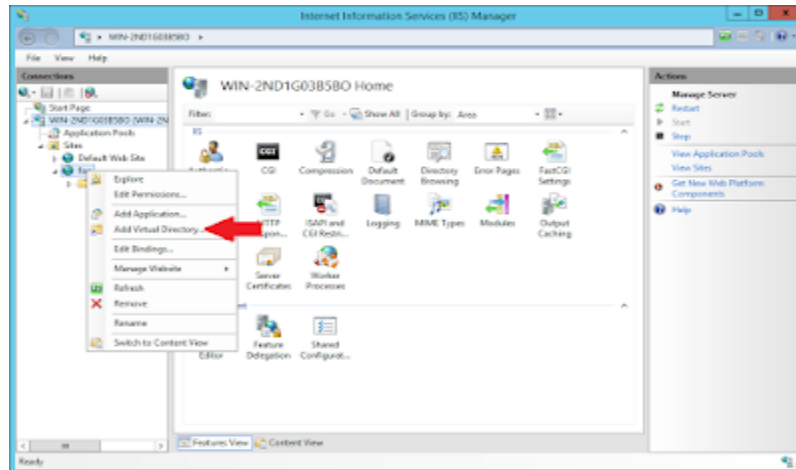


### 3. Add a static Virtual Directory to the IIS Web Site

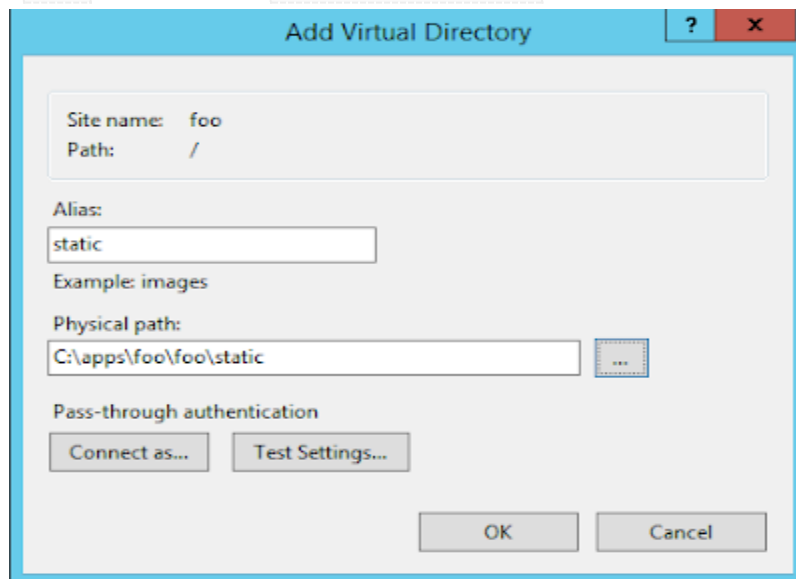
- Now we have all our static files in one place, but we still haven't told IIS where they're located. We'll accomplish this by adding a `static` virtual directory to the IIS web site for our application.
- Note that the name of the virtual directory in IIS *must match* the value of the `STATIC_URL` setting in the Django application's `settings.py` file, absent the beginning and trailing slashes. For our sample application we're using a value of `/static/` for the `STATIC_URL` setting, so the name of the virtual directory in IIS will be `static`. To create virtual directory.
- Open IIS Manager
- On the left-hand side under "Connections," expand the server's tree
- Expand the "Sites" folder
- Expand the foo web site. At this point, things should look like this:



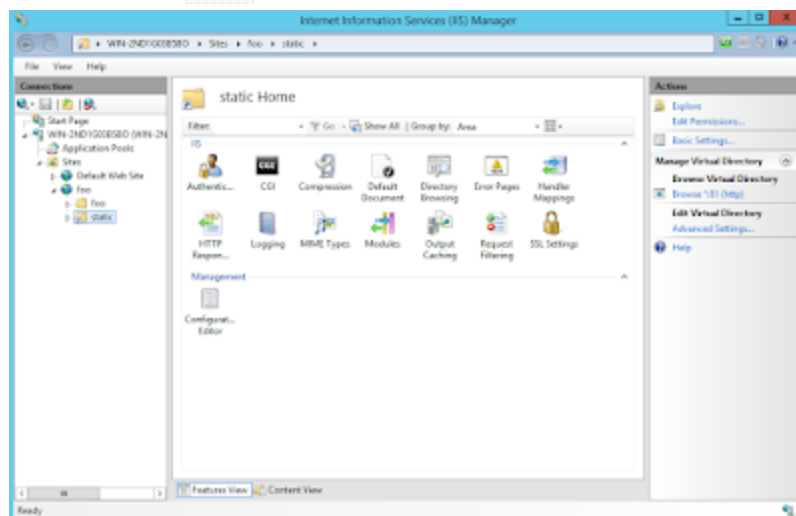
- Right-click the foo web site and click "Add Virtual Directory"



- In the Add Virtual Directory dialog, enter the following values
  - Alias: static Physical path: C:\apps\foo\foo\static



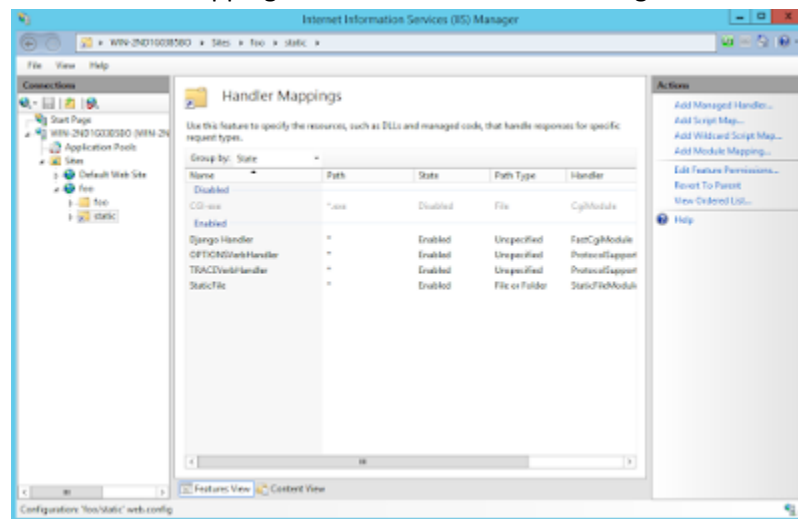
- Click "OK." You will see the static virtual directory appear under the web site.



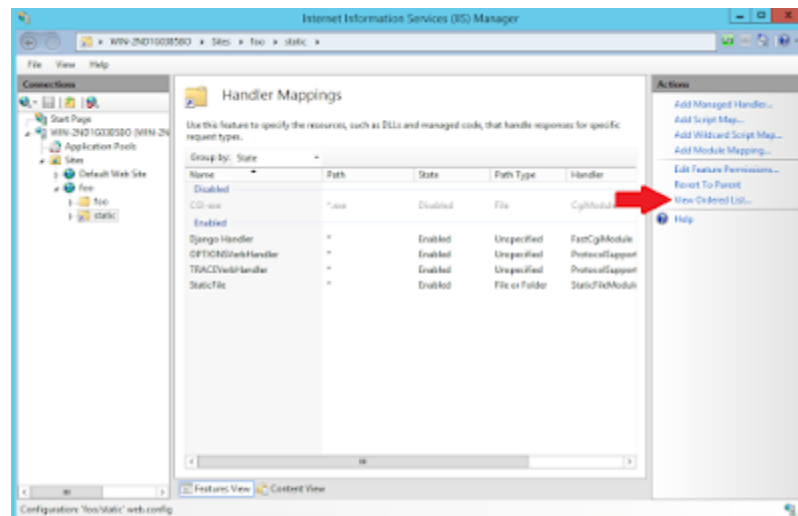


#### 4. Configure Handler Mappings in IIS for the static Virtual Directory

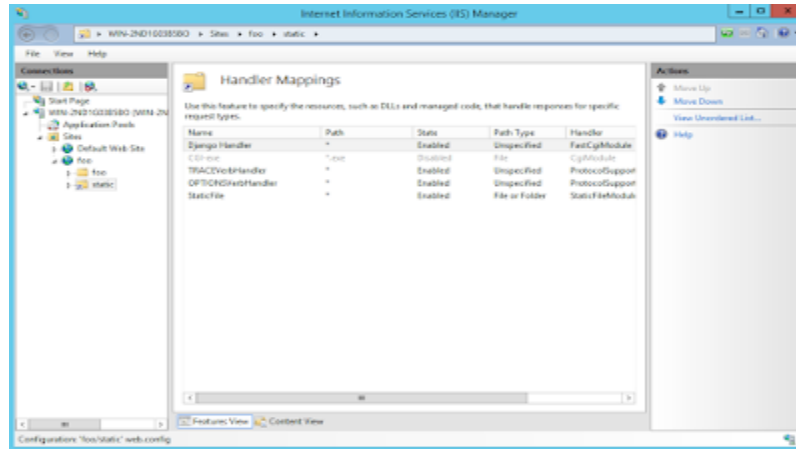
- At this point our Django Handler is set to serve all files for our application, even with the static virtual directory in place the Django Handler will still be attempting to serve these files.
- IIS already has a static file handler active in the web site, but it's down further in the list of handlers than the global Django Handler we configured, so to get IIS to serve the static files under the static virtual directory we'll move the static file handler to the top of the handler list on the static virtual directory.
- Open IIS Manager
- Expand the server tree
- Expand the foo web site
- Click on the static virtual directory
- Double-click the "Handler Mappings" icon. You'll see the following list of handler mappings:



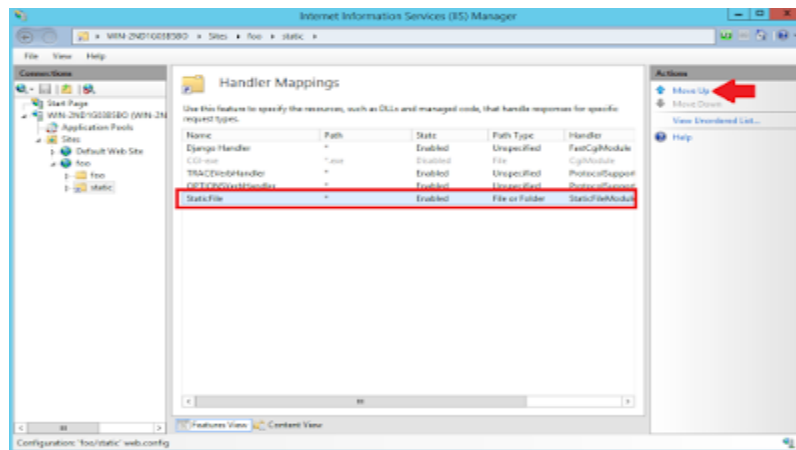
- On the right-hand side under "Actions" click on "View Ordered List ..."



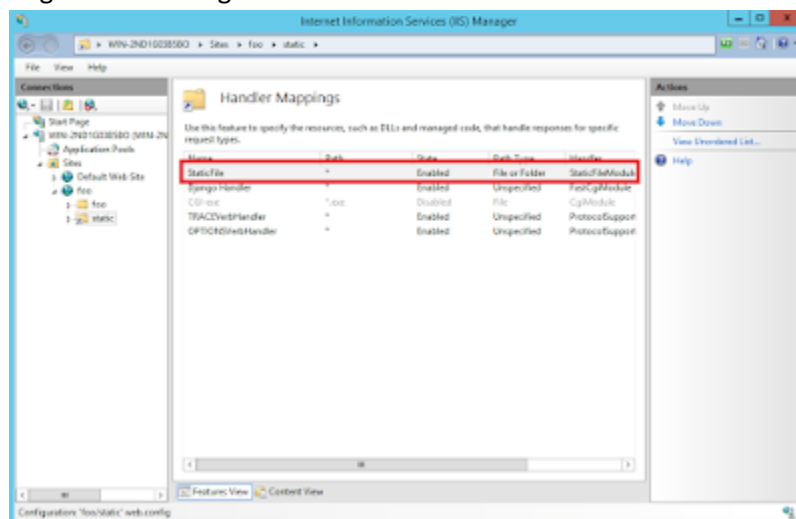
- You'll now see a list of the handler mappings in the order in which they're run, with the Django Handler at the top.



- Click on the StaticFile handler at the bottom of the list, then click “Move Up” under “Actions” on the right-hand side.

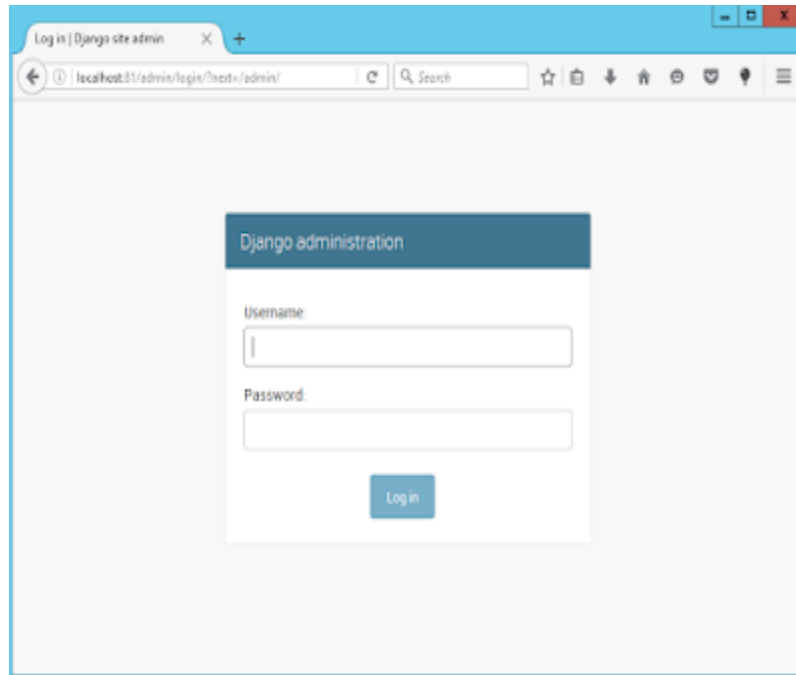


- Note at this point you may receive a warning that by moving handlers you are detaching this virtual directory from the inheritance of the parent’s handler mappings. This is what we want to do, so if you get this warning click “Yes”.



- Continue to click “Move Up” until the StaticFile handler is at the top of the list of Handler Mappings.

- Confirm that everything is working by going to <http://localhost:81/admin> in a browser. You should now see the CSS being applied to the page.



## 1.3 CONNECTION SETUP (MW TO BE)

To connect Middleware to Backend, install the following packages using pip in the virtual environment of the project:

1. pyodbc==4.0.30
2. django-mssql-backend==2.8.1
3. azure-identity==1.4.0
4. azure-keyvault-secrets==4.2.0

### 1.3.1 AZURE KEY VAULT

**AZURE Key Vault:** Set and retrieve a secret from Azure Key Vault using the Azure portal

**Prerequisites:** To access **Azure Key Vault**, you will need an Azure subscription. All access to secrets takes place through **Azure Key Vault**. For this **quickstart**, create a key vault using Azure portal, Azure CLI, or Azure PowerShell.

**Sign-in to Azure:** Sign-in to the Azure portal at <https://portal.azure.com>.

### **Add a secret to Key Vault:**

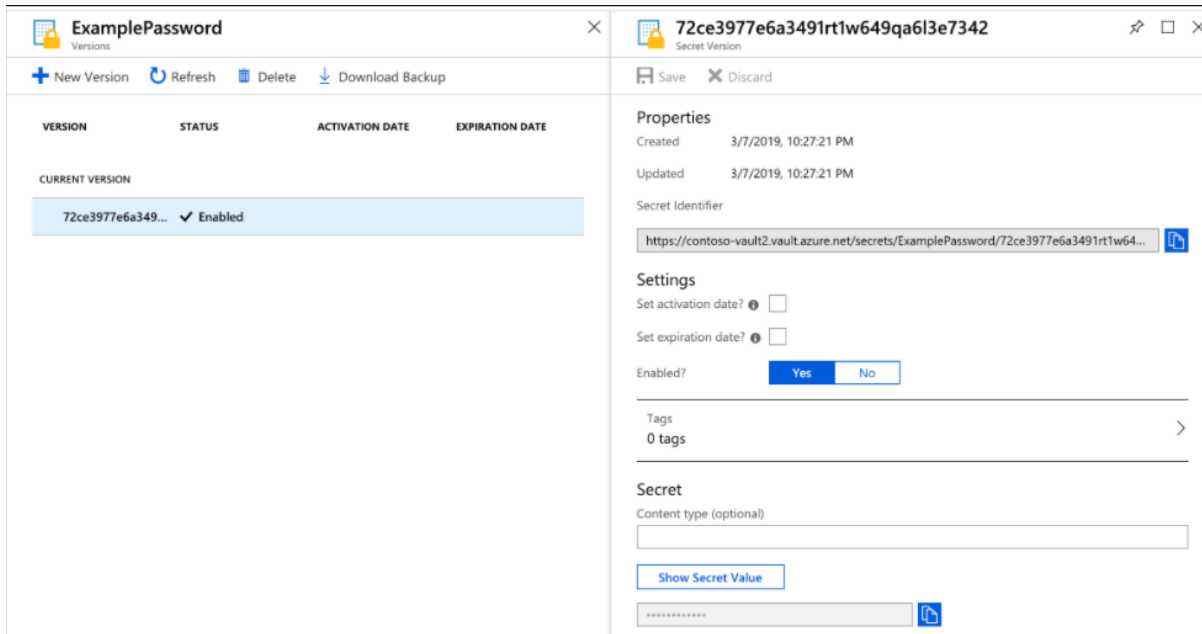
To add a secret to the vault, follow the steps:

1. Navigate to your new key vault in the Azure portal
2. On the Key Vault settings pages, select **Secrets**.
3. Click on **Generate/Import**.
4. On the **Create a secret screen** choose the following values:
  - a. **Upload options:** Manual.
  - b. **Name:** Type a name for the secret. The secret name must be unique within a **Key Vault**. The name must be a 1-127 character string, starting with a letter and containing only 0-9, a-z, A-Z, and -. For more information on naming, see Key Vault objects, identifiers, and versioning
  - c. **Value:** Type a value for the secret. Key Vault APIs accept and return secret values as strings.
  - d. Leave the other values to their defaults. **Click Create**.

Once that you receive the message that the secret has been successfully created, you may click on it on the list.

### **Retrieve a secret from Key Vault**

If you click on the current version, you can see the value you specified in the previous step.



By clicking "**Show Secret Value**" button in the right pane, you can see the hidden value.

## Secret

Content type (optional)

**Hide Secret Value**

hVFkk965BuUv



## Middleware Key Vault

We use a key vault which is a part of our resource group to store passwords for our SQL database.

The following table shows an example with the details:

Key name	Storage info	Used in
SQL-PASSWORD-DEV	This stores the password for the SQL Server hosted on dev Kubernetes cluster of Product Innovation Cluster.	Development version of project hosted over cluster part of product and innovation cluster. Contains dummy data.
SQL-PASSWORD-DEV-SQL	This stores the password for the Dev SQL Server hosted in Development RG of project.	Used in Pre-Production (hosted over PI cluster) and Internal-Dev(hosted over internal cluster) version of the project tool

These keys are accessed by middleware from the environment files specific to each version of the tool.

The below snippet shows the **settings.py** file of the project.

```
123 SCHEMA = os.environ["SCHEMA"]
124 SCHEMA_ID = os.environ["SCHEMA_ID"]
125 TENANT_ID = os.environ["AZURE_TENANT_ID"]
126 CLIENT_ID = os.environ["APP_CLIENT_ID"]
127 SQL_PASSWORD_TO_FETCH = os.environ['SQL_PASSWORD']
128 keyVaultName = get_env_value("KEY_VAULT_NAME")
129 KVUri = f"https://{keyVaultName}.vault.azure.net"
130
131 credential = DefaultAzureCredential()
132 client = SecretClient(vault_url=KVUri, credential=credential)
133 DATABASES = {
134     'default': {
135         'ENGINE': 'sql_server.pyodbc',
136         'NAME': get_env_value("SQL_DB_NAME"),
137         'USER': get_env_value("SQL_DB_USER")+"@"+get_env_value("SQL_DB_SERVER").split(".")[0],
138         'PASSWORD': client.get_secret(SQL_PASSWORD_TO_FETCH).value,
139         'HOST': get_env_value("SQL_DB_SERVER"),
140         'PORT': '1433',
141         'TEST': {
142             'NAME': 'test_e2e',
143             'MIRROR': 'default'},
144         'OPTIONS': {
145             'driver': os.environ.get("DB_DRIVER"),
146         },
147     },
148 }
149
```

1. Depending upon the **working environment** the values of the **parameters** are fetched from the environment files(**dev/staging**).
2. SCHEMA, SCHEMA\_ID, TENANT\_ID, CLIENT\_ID, SQL\_PASSWORD\_TO\_FETCH and KEY\_VAULT\_NAME are fetched from the environment file of the working environment.
3. Since the **KEY\_VAULT\_NAME** is in the encoded format we need to decode it using the **get\_env\_value()** function.
4. The code snippet for the **get\_env\_value()** function is shown below:

```
32 def get_env_value(env_variable):
33     try:
34         encoded_var = os.environ[env_variable]
35         decoded_b64_var = base64.b64decode(encoded_var)
36         decoded_var = decoded_b64_var.decode('ascii')
37         return decoded_var
38     except KeyError:
39         error_msg = 'Set the {} environment variable'.format(env_variable)
40         raise ImproperlyConfigured(error_msg)
41
```

5. The encoded **KEY\_VAULT\_NAME** value fetched from the environment file is passed into this function and this function helps to decode the value as shown above and returns it back.
6. The **keyVaultName** variable store the value of the decoded **KEY\_VAULT\_NAME** received from the **get\_env\_value** function and is used to generate the connection **URL** for the azure vault and is stored in the **client** variable.
7. This **client** variable is used to fetch the password from the key vault.
8. Inside the **DATABASES** object we define the details as shown and the values for the parameters like NAME, USER etc. is fetched from the environment file of the working environment.
9. This helps to setup the connection of the middleware to the backend using azure key vault.

The below snippet shows an example of the environment file of the project.

```
24 export KEY_VAULT_NAME=<"encoded key vault name">;
25 export AZURE_CLIENT_ID=<"client ID">;
26 export AZURE_CLIENT_SECRET=<"client secret">;
27 export AZURE_TENANT_ID=<"tenant ID">;
28
29 export APP_CLIENT_ID=<"application client ID">;
30
31 export SQL_DB_NAME=<"encoded SQL database name"> ;
32 export SQL_DB_SERVER=<"encoded SQL database server"> ;
33 export SQL_DB_USER=<"SQL database user"> ;
34
35 export SQL_PASSWORD=<"azure SQL password">;
36 export SCHEMA="dev";
37 export SCHEMA_ID=<schema ID>;
38
39 export DB_DRIVER=<"driver name for SQL server"> ;
40 DB_NAME=<'database name'> ; export DB_NAME
41 DB_USER=<'database user'> ; export DB_USER
42 DB_PASSWORD=<'database password'>; export DB_PASSWORD
43 DB_HOST=<'database host'>; export DB_HOST
44 DB_PORT=<'port number'> ; export DB_PORT
```

- The values for the parameters in the settings file are fetched from this file.
- Set the **KEY\_VAULT\_NAME** to the encoded key vault name that we receive from the **Azure Key Vault**.
- Similarly set the **AZURE\_CLIENT\_ID**, **AZURE\_CLIENT\_SECRET** etc. with the correct values.
- Give the **DB\_DRIVER** value as the driver name for the SQL server for example: ODBC Driver 17 for SQL Server

### 1.3.2 ODBC DRIVER CONNECTION FOR SQL SERVER

Open Database Connectivity (ODBC) is a standard application programming interface (API). It is used to connect to database management systems (DBMS). ODBC aim is to make it independent of database systems and operating systems.

To install the Microsoft ODBC driver 17 from the bash shell. The following steps are followed:

1. Install curl, python-dev, build-essential and gnupg
  - curl is the command line tool and library that is used for transferring data with URL syntax.
  - Python-dev contains the header files you need to build Python extensions. On Linux typically, the binary libraries and header files of packages like Python are separate. When you want to build extensions, you need to install the corresponding -dev package.
  - **Build-essential** package contains an informational list of packages which are considered essential for building Debian packages
  - **GNU privacy guard (gnupg)** - is GNU's tool for secure communication and data storage. It is used to encrypt data and to create digital signatures. It includes an advanced key management facility and is compliant with the proposed OpenPGP Internet standard as described in RFC 4880.

```
1 apt-get -y install curl
2 apt-get -y install python3-dev
3 apt-get -y install build-essential
4 apt-get -y install gnupg
```

## 2. Install Microsoft SQL SERVER ODBC Driver

```
curl https://packages.microsoft.com/config/debian/10/prod.list > /etc/apt/sources.list.d/mssql-release.list
curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add -
```

Now we must update our Ubuntu operating system. Type the following:

```
apt-get update
```

We can now install the ODBC driver. To do this type:

```
ACCEPT_EULA=Y apt-get -y install msodbcsql17
```

We now have our ODBC driver installed for Microsoft SQL server.

## 3. Install SQL Server tools

These include sqlcmd and bcp. sqlcmd includes a command prompt to enter Transact-SQL statements. BCP copies data between Microsoft SQL server and file formats of your choice.

The first command installs the Microsoft SQL Server tools.

```
ACCEPT_EULA=Y apt-get -y install mssql-tools
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bash_profile
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bashrc
source ~/.bashrc
```

## 4. Install the ODBC development headers

```
apt-get -y install unixodbc-dev
```

## 5. Install Kerberos runtime libraries - krb5 GSS-API

Kerberos is a system for authenticating users and services on a network. Kerberos is a trusted third-party service. That means that there is a third party (the Kerberos server) that is trusted by all the entities on the network (users and services, usually called "principals").

This package contains the runtime library for the MIT Kerberos implementation of GSS-API used by applications and Kerberos clients.

```
apt-get -y install libgssapi-krb5-2
```

## 1.3.3 REQUIREMENTS

A requirements.txt file is a file that lists all the modules needed for the Django project to work. It helps everyone working in the project to have the same versions of all the modules/packages and prevents running into any problem while setting up the project.

- How to create a requirements.txt file



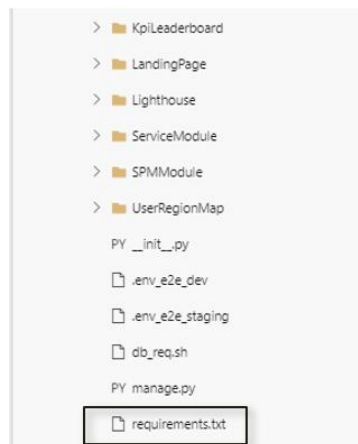
Inside the virtual environment of our project in the terminal we can see all the packages/modules installed for the project with the help of the following command:

*pip freeze*

This command will list all the modules. Now to create a requirements.txt file that contains all the listed packages run the following command:

*pip freeze > requirements.txt*

So now we have a requirements.txt file created that lists all the modules that you installed for the project. This file is created inside the project folder at last after the application folders and environment files as shown below:



- How to use requirements.txt file

The requirement.txt file created for the project should look like the snippet below:

```
1 Django==3.0.6
2 django-cors-headers==3.3.0
3 django-pandas==0.6.1
4 djangorestframework==3.11.0
5 numpy==1.18.4
6 pandas==1.0.3
7 pip==20.2
8 pyodbc==4.0.30
9 django-mssql-backend==2.8.1
10 sentry-sdk==0.15.1
11 django-rest-auth==0.9.5
12 djangorestframework-jwt==1.11.0
13 python-memcached==1.59
14 django-nose==1.4.6
15 coverage==5.2.1
16 PyJWT==1.7.1
```

While setting up the project in our system we need to install all the modules/packages related to that project. For this we use the requirements.txt file and install the packages/modules using the command below:

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

This will install all the packages/modules and the project will be up and ready to run.

# 1.4 UNIT TESTING

Unit test is a test that tests a single unit of code in isolation. A unit could be an entire module, a single class or function, or almost anything in between.

## 1.4.1 HOW TO STRUCTURE A UNIT TEST

1. A good test is divided into 3 parts

- **Given** – Make sure your system is in the right state for testing. For example, create objects, initialize parameters, and define constants. Define the expected results of the test.
- **When** – The key actions of the test. Typically, one or two lines of code
- **Then** – compare the outcomes of the key actions to that of the expected ones. Set of assertions regarding the new state of your system

2. Test simple but general cases

3. Test special cases and boundary conditions

- Code often breaks in boundary cases – empty list, none, Nan, 0.0, list with repeated elements, non-existent files.
- Respond to these cases with special behavior or raise meaningful exception.

## 1.4.2 STEPS FOR WRITING A UNIT TEST

- Write tests in tests.py for each app
- Check if tests are running by using *python manage.py test <appName> --parallel 4*  
*--parallel* can run test in parallel to gain a speed up on multi-core hardware.
- Status OK means the tests have passed
- Generate a unit test report and look at optimising the test coverage. To Generate the test results and coverage reports we use **django-nose**

- Install the following packages

***pip install django-nose***

***pip install coverage***

- Add the following to **settings.py**

```
INSTALLED_APPS = (
```

```
# ...
```

```
'django_nose',
```

```
)
```

***# Use nose to run all tests***

```
TEST_RUNNER = 'django_nose.NoseTestSuiteRunner'
```

***# Tell nose to measure coverage on the apps mentioned in cover-package***

```
NOSE_ARGS = [
```

```
'--with-xunit',
```

```
'--xunit-file=unit_test.xml',
```

```
'--with-coverage',
```

```

        '--cover-package = UserRegionMap.views, ServiceModule.views, DemandModule.views,
        KPIOverview.views', <replace your modules here>
        '--cover-xml',
        '--cover-xml-file=coverage.xml',
        '--cover-html',
        '--cover-erase',
    ]

```

- Now run the tests using `python manage.py test` to generate the reports.

### 1.4.3 TESTING WITH PYTHON: UNITTEST

**Unittest** is part of the Python standard library. Each test case is a subclass of `unittest.TestCase`. Each test unit is a method of a class, who name starts with **'test'**. Each test unit checks one aspect of your code and raises an exception if it does not work as expected.

A unit test consists of one or more **assertions** (statements that assert that some property of the code being tested is true). Each test should test a single, specific property of the code and be named accordingly.

**Example:** Testing an application

- Import the necessary **packages** and **function names** from application's views as shown.

```

8 from django.urls import reverse, resolve
9 from django.test import RequestFactory, TestCase, SimpleTestCase, Client, modify_settings
10 from django.contrib.auth.models import User
11 from GlobalOverview.views import get_global_overview_filter_options, get_global_overview_chart_data,
   get_leader_board_data
12 import json
13 from django.conf import settings
14 from django.db import connection
15

```

- Below snippet shows how to write a test for testing a function in an application.
  - **SetUp()** initializes the test.
  - Line 43 contains the json data that is sent to hit the **URL** of the API. This json data should be same as request body that is sent to the function present in the **views.py** file of the application.

```

30 class Test_Global_Chart(TestCase):
31
32     @modify_settings(MIDDLEWARE={'remove': 'UserRegionMap.regionmiddleware.CustomAuthMiddleware',})
33
34     def setUp(self):
35         self.factory = RequestFactory()
36
37     def test_submission_status1(self):
38         segment = ['MW', 'PET', 'FOOD']
39         region = ['All', 'North America']
40
41         for seg in segment :
42             for r in region:
43                 data1=[{"segment" : seg, "region": r}]
44                 request1 = self.factory.post(
45                     'getGlobalOverviewChartData/', data1, content_type='application/json')
46
47                 response1 = get_global_overview_chart_data(request1)
48                 self.assertEqual(response1.status_code, 200)
49

```

**TestCase.assertsomething**

**TestCase** defines utility methods to check that some conditions are met and raises an exception otherwise. Some examples are:

- Check that statement is **True/False**.
- Check that two objects are **equal**.
- Check that two numbers are **equal** up to a **given precision**.
- Check that **exception** is **raised**.

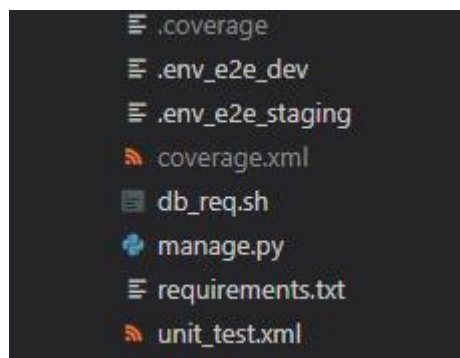
Many more assert methods – complete list can be found here:  
<<https://docs.python.org/2/library/unittest.html>>

**Note:**

- Most of the assert methods accept an optional msg argument that overwrites the default one.
- Most of the assert methods have a negated equivalent.

## 1.4.4 RUNNING TESTS AND GENERATING REPORTS

- Run the test using **python manage.py test** to run tests present in the **tests.py** file of the applications.
- This will generate three files **.coverage** , **coverage.xml** and **unit\_test.xml**.



- A cover folder is created that contains the coverage report of all the tests and the index.html file,
- To view the coverage report open **index.html** in your browser by right clicking on index.html and selecting copy path and pasting it in the browser.
- The below snippet shows the coverage report.

Coverage report: 91%				
Module ↓	statements	missing	excluded	coverage
ServiceModule\views.py	855	77	0	91%
<b>Total</b>	<b>855</b>	<b>77</b>	<b>0</b>	<b>91%</b>
coverage.py v5.2.1, created at 2021-05-17 17:21 +0530				

Coverage for **ServiceModule\views.py** : 91%

855 statements   778 run   77 missing   0 excluded

```
1  """
2  *
3  * Title - Service Module
4  * Author - Suathil Ganesan
5  * Date - 26-Feb-2021
6  *
7  """
8  from ServiceModule.models import DateMaster
9  from django.http import HttpResponse, JsonResponse
10 from django_pandas.io import read_frame
11 import json
12 import pandas as pd
13 from datetime import datetime
14 import numpy as np
15 import copy
16 import random
17 import sys
18 import math
19 from django.db import connection
20 from django.core.serializers.json import DjangoJSONEncoder
21 import warnings
22 from pandas.core.common import SettingWithCopyWarning
23 from django.conf import settings
24 from CommonUtils.common_functions import dictfetchall, calcWindow13
25
26 warnings.simplefilter(action="ignore", category=SettingWithCopyWarning)
27
28
29
30 def get_service_filter_options(request):
31     """This function returns the filter options for the Service Factory Page
32
33     :param request: {"segment" : segment}
34     :type request: object
35     :return: filter options for market, businessSegment, year filters along w lastUpdate date for service table
36     :rtype: JSON Object in the format {"demandFilter": {"market": {<markets>}, "businessSegment": {<market>: {<business segments>}}}, "year": {<market>: {<years>}}}]
37     """
```

## 1.5 AUTOMATION TESTING

**Automation Testing** is a software testing technique that performs using special automated testing software tools to execute test cases.

We use **Selenium** which is a popular open-source web-based automation tool in **Python** Programming Language.

### 1.5.1 PREREQUISITES

There are certain prerequisites which are needed before starting the Automation Testing.

- All the potential Bugs must be fixed which were encountered while performing manual tests
- Application should be Stable
- Manual Testing should be completely done before Automation testing starts
- Development team must communicate to testing team whenever some new features are pushed, or some functionality/feature is brought down
- Automation Testing tasks will always be lagging the Development phase

### 1.5.2 AUTOMATION FRAMEWORK

**POM (Page Object Model)** is a design pattern or a framework that we use in *Selenium* using which one can create an object repository of the different web elements across the application.

This Object Repository is divided into 3 parts:

**Locators** – we create a class file for each web page. This class file consists of different web elements present on the web page.

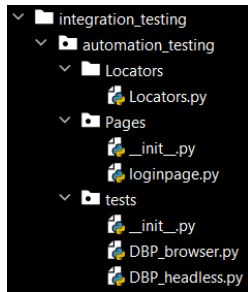
**Pages** – We then create test scripts which use these elements to perform different actions.

**Tests** – We then create user defined functions which are group of different test scripts which represent a particular functionality or a feature and are added sequentially to the file in order to mimic the actions of a real person using the application.

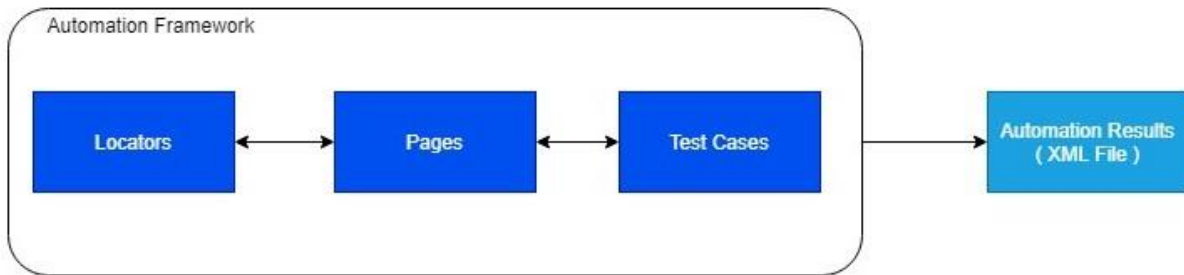
Types of Automation tests required:

- **Web Automation test:** Executing the test cases on browser
- **Headless Automation test:** Executing the test cases on CI Systems

Following example shows the file structure of framework:



**Test Results** – After the test is complete, the test results are stored in an XML file.



## 1.5.3 STEPS FOR WRITING TESTS

### Installing Packages

- *Pip install selenium*
- *Pip install xmlrunner*
- *Pip install unittest*
- *Other required Packages like 'os', 'sys' and 'time' are already installed with python*

Download a selenium webdriver which is specific to the browser version (Chrome recommended) used for Automation from here: <https://chromedriver.chromium.org/downloads>

### Add web elements to class file

```
Locators.py x
1 class Locators:
2     dev_URL = 'https://dbp.eastus.cloudapp.azure.com/dev/'
3     email_id = 'global.dbp@effem.com'
4     psw = 'o6PiQZ.38bsu'
```

### Writing Test Scripts in 'Pages' Directory

- Importing Packages



```

import xmlrunner
from selenium import webdriver
import time
import unittest
import sys
import os

```

- This snippet shows how to write a test script
  - A class object is created where different web elements are called to test single or multiple feature(s)

```

class login_page():
    def __init__(self, driver):
        self.driver = driver
        self.username_textbox_ID = Locators.username_textbox_ID
        self.password_textbox_ID = Locators.password_textbox_ID

    def SS0_integration(self, EffmID, password):
        self.driver.find_element_by_xpath(Locators.effm_id).send_keys(EffmID)
        time.sleep(6)
        self.driver.find_element_by_xpath(Locators.next_to_login).click()
        time.sleep(35)
        self.driver.find_element_by_id(Locators.mars_effm_id).send_keys(EffmID)
        self.driver.find_element_by_id(Locators.mars_password_id).send_keys(password, Keys.RETURN)
        time.sleep(25)
        self.driver.find_element_by_xpath(Locators.popup).click()
        time.sleep(5)
        self.driver.find_element_by_xpath(Locators.SS0).click()
        time.sleep(6)

```

## Grouping test scripts in 'tests' Directory

- This snippet shows how to add a decorator and connection with webdriver
  - Adding a decorator `@classmethod` so you can create class methods that pass the actual class object within the function call
  - Executable path is the webdriver location through which webdriver will remain connected when automation script is running
  - Below snippet is for web automation testing

```

@classmethod
def setUpClass(cls):
    cls.driver = webdriver.Chrome(executable_path='C:\driver\chromedriver_win32\chromedriver.exe')
    chromedriver_autoinstaller.install()
    cls.driver.maximize_window()

```

- Below snippet is for headless automation testing. This test is being run when integrated with CI Systems

```

chromedriver_autoinstaller.install()
#cls.driver.maximize_window()
# Headless
options = Options()
options.add_argument("--headless")
options.add_argument("--nogpu")
options.add_argument("--window-size=1920,1080")
cls.driver = webdriver.Chrome(options=options)
cls.driver.implicitly_wait(10)

```

- Grouping test cases
  - we group test scripts in a function
  - Naming convention plays an important role here as we want to call tests one after other, not in a random pattern.
  - We start first test name with “test\_A\_<name of test>” and so on...
  - In this way, all the tests are covered sequentially

```

def test_A_login_page(self):
    driver = self.driver
    driver.get(Locators.dev_URL)
    login = login_page(driver)
    login.SSO_integration(Locators.email_id, Locators.psw)

```

- Generating test results
  - Below script shows how test results are generated in xml file using ‘unittest’ and ‘xmlrunner’ library

```

if __name__ == '__main__':
    with open('./automation_results.xml', 'w+') as output:
        unittest.main(
            testRunner=xmlrunner.XMLTestRunner(output=output),
            failfast=False, buffer=False, catchbreak=False)

```

## 1.5.4 RUNNING TESTS AND GENERATING REPORTS

- In POM Framework, every file is treated as a module, test files are in ‘tests’ directory
  - To run browser test: `python -m tests.browser.py`
  - To run headless test: `python -m tests.headless.py`
- This will generate test results file “automation\_results.xml”
  - Below snippet shows the test results report

```
<?xml version="1.0" ?>
<testsuite errors="0" failures="0" name="DBPTest-20210413173516" tests="21" time="888.724">
  <testcase classname="DBPTest" name="test_A_login_page" time="101.499"/>
  <testcase classname="DBPTest" name="test_B_Overview_Page" time="10.533"/>
  <testcase classname="DBPTest" name="test_C_Download_and_Upload_CSV" time="33.390"/>
  <testcase classname="DBPTest" name="test_D_Submission_data_page" time="75.409"/>
  <testcase classname="DBPTest" name="test_E_Re_Submission_data_page" time="53.206"/>
  <testcase classname="DBPTest" name="test_F_Submission_logs" time="45.851"/>
  <testcase classname="DBPTest" name="test_G_View_plan_NSV" time="46.509"/>
  <testcase classname="DBPTest" name="test_H_View_Actual_NSV" time="48.394"/>
  <testcase classname="DBPTest" name="test_I_Admin_Actions_Valid_days" time="47.109"/>
  <testcase classname="DBPTest" name="test_J_Thresholds_and_Segments" time="57.909"/>
  <testcase classname="DBPTest" name="test_K_Submission_Management" time="24.007"/>
  <testcase classname="DBPTest" name="test_L_Manage_M0E" time="26.292"/>
  <testcase classname="DBPTest" name="test_M_NSV_Reconciliation" time="12.756"/>
  <testcase classname="DBPTest" name="test_N_user_management" time="38.343"/>
  <testcase classname="DBPTest" name="test_O_Data_Quality" time="103.306"/>
  <testcase classname="DBPTest" name="test_P_Manage_plan_rates" time="21.682"/>
  <testcase classname="DBPTest" name="test_Q_Add_Remove_Admin" time="21.477"/>
  <testcase classname="DBPTest" name="test_R_Add_remove_User" time="27.769"/>
  <testcase classname="DBPTest" name="test_S_Submission_Monitoring" time="53.229"/>
  <testcase classname="DBPTest" name="test_Y_data_refresh" time="31.109"/>
  <testcase classname="DBPTest" name="test_Z_logout" time="8.946"/>
</system-out>
```

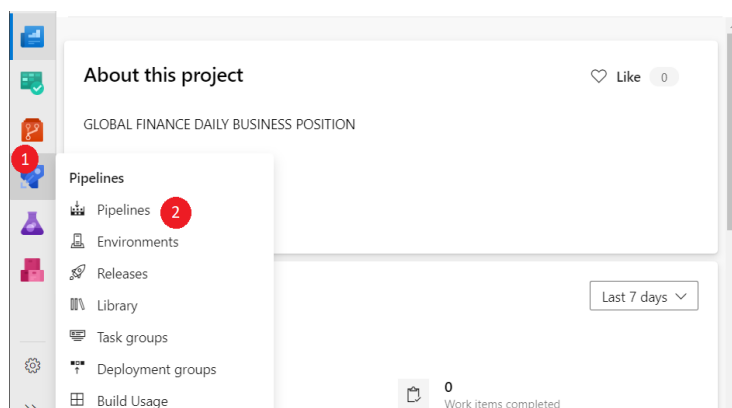
## 1.5.5 INTEGRATION WITH CI PIPELINES

### Prerequisites:

- An organization in **Azure DevOps**. If you do not have one, you can sign up for one for free. Each organization includes free, unlimited private Git repositories.
- You must have the Pushed your framework files to the git repository.

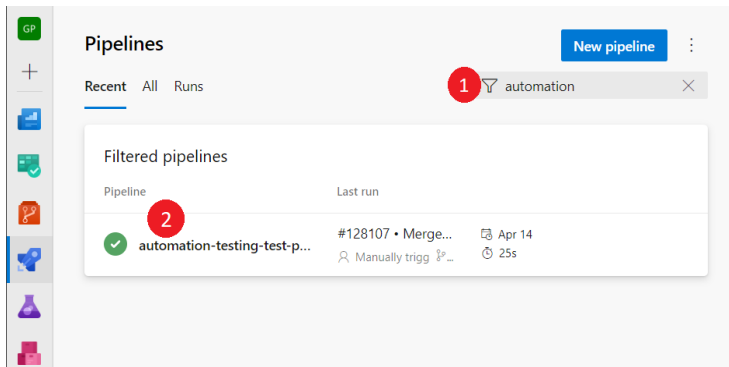
### Navigate to Automation test Pipeline

- Get to your Project in Azure DevOps organization and hover cursor on Pipelines from left side navigation drawer.
- From the dropdown list, select 'Pipelines' option.

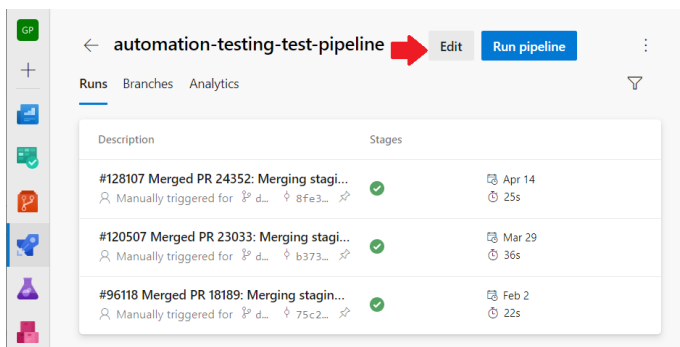


### Click on Edit Button

- In the Pipelines window, enter "automation" in the search bar.
- Select the Automation testing pipeline result

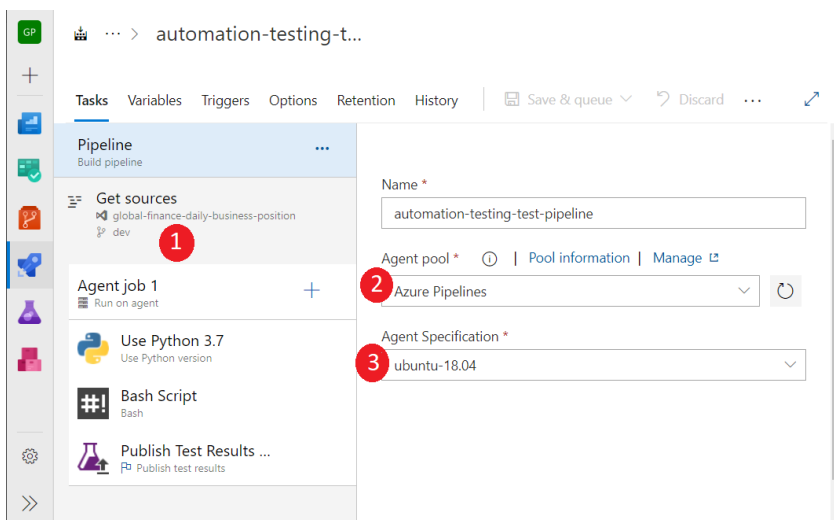


- If needed, click on Edit button for editing the pipeline tasks



## Pipeline Tasks

- Make sure that sources for the automation task are being provided from the dev branch of repository not a feature branch
- The mandatory specification: agent Pool will be “Azure Pipelines” and Agent Specification will be “ubuntu-18.04”



- In the Agent Job of this pipeline task,

- o Mention the Python version which will be used for carrying out the test

The screenshot shows the Jenkins Pipeline configuration page. On the left, under 'Agent job 1', the task 'Use Python 3.7' is selected. On the right, the configuration for this task is shown. The 'Task version' is set to '0.\*'. The 'Display name' is 'Use Python 3.7', with a red arrow pointing to it. The 'Version spec' is '3.7'. The 'Add to PATH' checkbox is checked. The 'Advanced' section is collapsed.


- o In the Bash Script, mention the task version, Display name and Type of script.
- o Enter all the requirements in Script which are needed when automation test is being run.

The screenshot shows the Jenkins Pipeline configuration page with the 'Bash Script' task selected. The 'Task version' is '3.\*'. The 'Display name' is 'Bash Script'. The 'Type' is set to 'Inline'. The 'Script' field contains the following commands: `google-chrome --product-version`, `pip install -r requirements.txt`, `#python -m tests.login`, `python -m tests.DBP headless.py`, `cd Reports`, and `cat automation_results.xml`. The 'Advanced' section is collapsed.

- o In Publish test results, mention all the specifications required to store the generated test results.

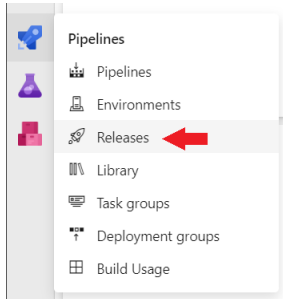
The screenshot shows the Jenkins Pipeline configuration page with the 'Publish Test Results' task selected. The 'Task version' is '1.\*'. The 'Display name' is 'Publish Test Results src/integration\_testing/automation\_testing/automation...'. The 'Test Result Format' is 'JUnit'. The 'Test Results Files' field contains 'src/integration\_testing/automation\_testing/automation\_results.xml'. The 'Merge Test Results' checkbox is unchecked. The 'Test Run Title' is 'Publish test report'.

- o If you have made any changes in the pipeline tasks, “Save and Queue” Button will activate and clicking on it will save the recent changes.

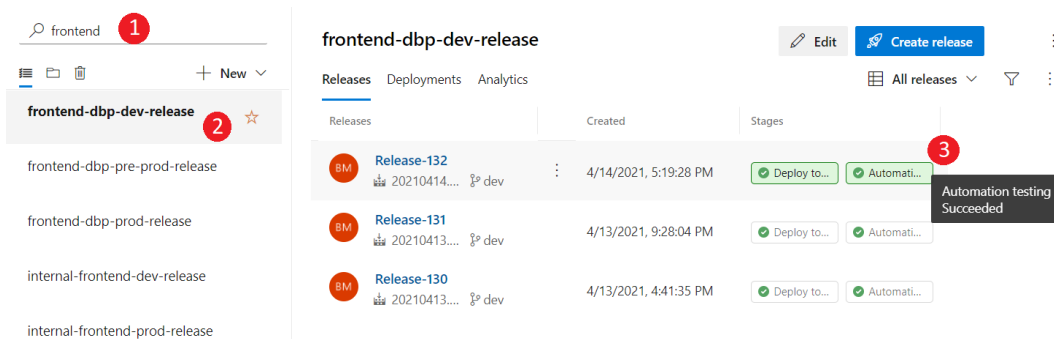
 Save & queue ▾

## Pipeline Release

- Now navigate to pipelines and click on “release” option from the dropdown list.



- Search for “frontend” in the search bar in release window
- Select a result which is “frontend dev release” among the search results. You will find “Automation Testing Succeeded” in the 2nd stage of that release.
- Pipeline Release is configured in such a way that whenever a push from frontend or automation testing happens, the pipeline runs and generates test results



- Clicking on it will show us list of logs for the automation Agent Job

Deployment process  
Succeeded

Agent job  
Succeeded · 1 warning

### Agent job

Pool: [Azure Pipelines](#) · Agent: Azure Pipelines 2

- Initialize job · succeeded
- Download artifact - \_frontend-dbp-dev - Code Coverage Report\_127915 · succeeded
- Download artifact - \_frontend-dbp-dev - frontend · succeeded
- Download artifact - \_frontend-dbp-dev - drop · succeeded
- Download artifact - \_frontend-dbp-dev - testdrop · succeeded
- Use Python 3.7 · succeeded
- Bash Script · succeeded
- Publish Test Results D:\a\r1\dev\\_frontend-dbp-dev/testdrop/Reports/automation\_results.xml · succeeded
- Finalize Job · succeeded

## Navigate to Automation test Pipeline

- Again, navigate to pipelines and click on “pipelines” option from the dropdown list.
- Inside the pipelines there are pipeline runs performed at different time. Choose the latest among them

← automation-testing-test-pipeline

Edit
Run pipeline

Runs
Branches
Analytics

Description	Stages	
<b>#128107 Merged PR 24352: Merging staging into dev</b> Manually triggered for dev 8fe30e2		Apr 14 25s
<b>#120507 Merged PR 23033: Merging staging into dev</b> Manually triggered for dev b373391		Mar 29 36s
<b>#96118 Merged PR 18189: Merging staging in to dev</b> Manually triggered for dev 75c2556		Feb 2 22s

- When inside the latest run pipeline, select “Tests” Tab
- There you will see all the tests and results.

Summary
Tests

Summary

1 Run(s) Completed ( 1 Passed, 0 Failed )

21  
Total tests  
+7

21 Passed  
0 Failed  
0 Others

100%  
Pass percentage

14m 48s  
Run duration  
↑ +7m 33s

0  
Tests not reported

- Make sure that only this test result has to be shared with clients on their teams channel or via mail when clients ask for automation test result.

# 1.6 DOCUMENTATION

## Setting up Sphinx with Python

1. To **install sphinx** with python first **create a virtual environment** in the command prompt.
2. **Activate** the **virtual environment** and move to the path that has your project folder
3. In the command prompt type the following command to **install Sphinx**:

```
pip install sphinx
```

this installs Sphinx in the virtual environment.

4. With the **virtual environment** still activated, run **sphinx-quickstart**, which creates a starting project for a Sphinx documentation project.

```
sphinx-quickstart
```

5. Answer all the questions from the prompts. You can choose enter to pick all the defaults and get a working project in the **current directory** (.).
6. Once you have the basics answered, the script creates the necessary files and you can edit those to your liking.
7. Create a couple of **.rst** files and add skeleton information for starters.
8. Edit those new **.rst** files in your text editor.
9. Now, you can build the docs to see the changes locally. Run this command in the directory with the **conf.py** file:

```
make html
```

10. In your browser, open the **build/html/index.html** file to look at your Sphinx site locally.
11. Edit the **source/index.rst** file to include links to the additional pages. Here is an example:

```
.. toctree::  
    :maxdepth: 2  
    :caption: Contents:  
  
    about.rst  
    prerequisites.rst
```

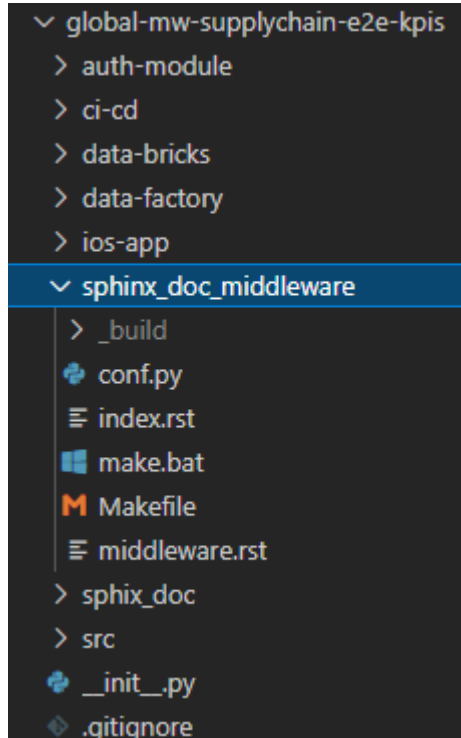
12. Build again to see these changes locally:



*make html*

13. Make sure you **commit** your changes to the **Git repository**.

The structure of the **Sphinx folder** in the project should be as follows:



The Sphinx doc for middleware is shown in the snippet.

sphinx\_doc\_middleware folder should be inline with the src folder. This structure should be followed for the successful implementation.

The sphinx\_doc\_middleware folder contains files like conf.py, index.rst, middleware.rst etc.

These files need to be properly configured to create the necessary documentation.

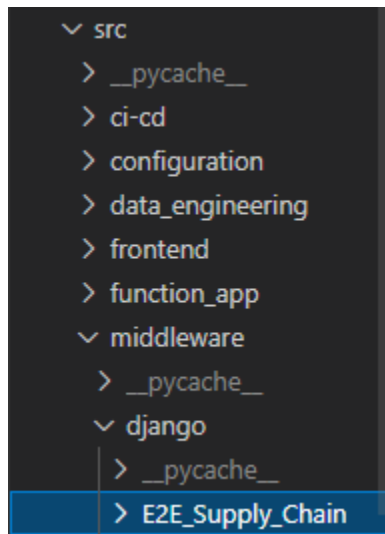
The snip of the **conf.py** file is shown below

- In the default **conf.py** file some necessary lines of code are needed to be added which is shown as below:

```
13
14 import sys, os, django
15 sys.path.insert(0, os.path.abspath('.'))
16 sys.path.append('../src/middleware/django/E2E_Supply_Chain')
17
18 os.environ.setdefault("DJANGO_SETTINGS_MODULE", "E2E_Supply_Chain.settings_local")
19 django.setup()
20
21
22
23
24 # -- Project information -----
25
26 project = 'End To End'
27 copyright = '2020, Mu-sigma'
28 author = 'Mu-sigma'
29
30 # The full version, including alpha/beta/rc tags
31 release = '1.0'
```

- In the above image line 14 to 19 is added.

- Line 16 `sys.path.append('../src/middleware/django/E2E_Supply_Chain')` helps to configure the path to the project folder in this case 'E2E\_Supply\_Chain'

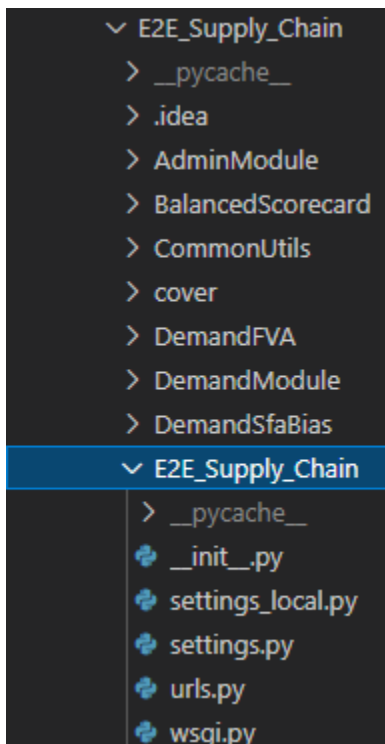


- Same method must be followed to configure it correctly.
- Line 18

`os.environ.setdefault("DJANGO_SETTINGS_MODULE", "E2E_Supply_Chain.settings_local")`

helps to set the Project settings file for the necessary environment configuration with necessary details present in the settings file of the project.

The file can be in the below structure.



The Image below shows the snip of **index.rst** file

```
global-mw-supplychain-e2e-kpis > sphinx_docmiddleware > index.rst
1  End To End Global KPI Reporting Tool's documentation!
2  =====
3
4  **Project Overview**
5
6  A single platform which would provide visibility into Supply Chain performance metrics across markets will
   help in performance monitoring and management of the end to end Supply Chain.
7
8
9  .. toctree::
10     :maxdepth: 2
11     :caption: Index:
12
13     middleware
14
15
16
17 |
```

The index.rst file should be configured for the index of the documentation as shown above.

The below snippet shows the **middleware.rst** file

- Create a **middleware.rst** file inside the **sphinx\_docmiddleware** folder.
- This folder contains the information to create the documentation for all the application views present in the project
- Configure the middleware.rst file for all the views present in the project. Refer the example snippet below:

```
8
9  Balanced Scorecard
10  =====
11
12  .. automodule:: BalancedScorecard.views
13     :members:
14     :undoc-members:
15
16  Demand Module Page
17  =====
18
19  .. automodule:: DemandModule.views
20     :members:
21     :undoc-members:
22
23  Demand Sfa & Bias Page
24  =====
25
26  .. automodule:: DemandSfaBias.views
27     :members:
28     :undoc-members:
29
```

- Create **documentation strings** inside each application view with the help of which the documentation of the project will be created.
- Refer the snippet below to see the example of doc strings in views.

```
def get_global_overview_chart_data(request):
    """This function returns the chart and tooltip data for the world chart in the Global Overview Page.
    Logic for map color : (SFA_SCORE+ BIAS_SCORE+ ATS_SCORE+ RFT_SCORE) > 3 then 1 when (SFA_SCORE+
    BIAS_SCORE+ ATS_SCORE+ RFT_SCORE) >=2 then 2 else 0

    :param request: {"segment" : segment}
    :type request: object
    :return: data for the world map and tooltip information
    :rtype: JSON Object in the format {"tooltip": [{"id": <country code>, "map":<map code>,"region":
    <region name>,"COLOR": 0,1 or 2 ,"data":[{"kpiName: <"CASEFILL", "FILLRATE", "FROT", "OTIF", "SFA",
    "BIAS", "ATS", "RFT">, "latestPeriod": <latestPeriod Value>, "Target": <target value>, "Status":
    <status value>, "adjFlag": <adjusted flag value - 0 or 1>}]}], "yearPeriod": corresponding yearPeriod}
    """
    try:
        if request.method == 'POST':
            post_data = json.loads(request.body)
            segment = post_data.get("segment")
            region_group = post_data.get("region")

            cursor = connection.cursor()
```

- The text inside the `""" """` is for the doc strings.
- The **doc strings** explain the functionality of the API and has different parameters like:
  - **Param request:** the parameters requested by the API
  - **Type request:** type of request
  - **Return:** what the API returns as output
  - **Rtype:** return type of the output which is generally in **JSON** format and the structure of the **JSON** output.
- These doc strings are used by the **middleware.rst** file to create the documentation.

## 1.7 ENCRYPTING REQUEST AND RESPONSES

**JSON Web tokens (jwt)** are used for **encrypting** requests and responses in our project.

**JSON Web Token (jwt)** is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with **HMAC algorithm**) or a public/private key pair using **RSA**.

### 1.7.1 STRUCTURE OF JSON WEB TOKENS (JWT)

### JSON Web Token example:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ0b3B0YWwuyY29tIiwiaWZlZXhwIjoxNDIyNDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaWlzL2l2ZX2FkbWluIjp0cnVlLCJjb2lwYW55IjoieGVzZGlzcmlzaW9udC9tZWZSI6dHJlZX0.yRQYnWzskCZUXPwaQuPWkiUZKELZ49eM7oWXaQK ZXw
```

There are 3 parts separated by a ., each section is created differently. The 3 parts are:

- header
- payload
- signature

```
<base64-encoded header>. <base64-encoded payload>. <base64-encoded signature>
```

**Header:**

The JWT Header declares that the encoded object is a **JSON Web Token** (JWT) and the **JWT** is a JWS that is MACed using the **HMAC SHA-256** algorithm. For example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

**"alg"** is a string and specifies the algorithm used to sign the token.

**"typ"** is a string for the token, defaulted to "JWT". Specifies that this is a JWT token.

**Payload(claims):**

A claim or a payload can be defined as a statement about an entity that contains security information as well as additional meta data about the token itself.

**Example:**

```
{User Id: xyz
First Name: 'ABC'
Last Name: 'PQR'
isAdmin: Yes
}
```

**Signature:**

**JSON Web Signature** specifications are followed to generate the final signed token. JWT Header, the encoded claims are combined, and an encryption algorithm, such as **HMAC SHA-256** is applied. The signature's secret key is held by the server so it will be able to verify existing tokens.

## 1.7.2 CREATING JWT IN PYTHON

- **Install** pyJWT package as  
*pip install pyJWT.*
- Once installed *import jwt*
- **Encode a request.**
  - We receive the token from the request body and decode it using **verify\_token()**

```
35     if request.method == 'POST':
36         post_data = json.loads(request.body)
37
38         token = post_data.get("token")
39         decoded_token = verify_token(token)
40         # print(decoded_token)
41
42         if decoded_token:
43             username = decoded_token['unique_name']
44             username = username.split('@')[0]
```

- Then using **decoded token**, we create username, then we create user credential object as **user\_cred** as payload which contains userID, first\_name, etc.

```
61         if user:
62             date_QS = DateMaster.objects.all()
63             date_DF = read_frame(date_QS)
64
65             user_cred = {}
66             user_cred["userID"] = username
67             user_cred["first_name"] = decoded_token['given_name']
68             user_cred["last_name"] = decoded_token['family_name']
69             user_cred["isAdmin"] = UserRegion.objects.filter(
70                 |   userid=username).values_list("isAdmin", flat=True).last()
71
```

- Now we encode the payload using **jwt.encode()** which contains payload, secret key, algorithm as parameters.
- Here, **JWT\_SECRET** key is stored in the settings.py file.
- **. decode("utf-8")** helps to decode the text in utf-8 format.

```
86
87         token_obj = jwt.encode(
88             |   user_cred, settings.JWT_SECRET).decode("utf-8")
89
```

- **Decoding the response**
  - We extract the **authorization header** and if it exists then we decode the request body in utf-8 format.

```

22     auth_header = request.META.get('HTTP_AUTHORIZATION', '')
23
24     if not auth_header:
25         if request.body:
26             body_unicode = request.body.decode('utf-8')
27             body_data = json.loads(body_unicode)
28             if "token" not in body_data:
29                 print("no auth header and no token in body")
30                 return HttpResponse("Unauthorised", status=401)
31             else:
32                 print("userRegionMap call")
33         else:
34             print("no auth header and no body")
35             return HttpResponse("Unauthorised", status=401)
36     else:
37         if request.body:
38             body_unicode = request.body.decode('utf-8')
39             body_data = json.loads(body_unicode)

```

- We decode the token using `verify_token()` using `idToken` parameter of the request body.

```

40         if "idToken" not in body_data:
41             print("no idToken as part of POST body")
42             return HttpResponse("Unauthorised", status=401)
43         else:
44             decoded_token = verify_token(body_data['idToken'])

```

- We split the **auth header** into **token type** and **credentials** and decode using `jwt.decode()`
- We decode it with the same secret and encoding algorithm as it was created.

```

45         if decoded_token:
46             token_type, _, credentials = auth_header.partition(' ')
47             if token_type == "Bearer":
48                 print("reached decoding part")
49                 decoded = jwt.decode(
50                     credentials, settings.JWT_SECRET, algorithms=['HS256'])

```

- If token expired or decoding error occurs, return response with error message. Otherwise, user is valid.

## 1.8 DJANGO BEST PRACTICES

Since Django is a Python based web framework, we use PEP-8 coding standards for writing the Django code.

PEP-8(Python Enhancement Proposal) is a document that provides guidelines and best practices on how to write Python code. PEP 8 exists to improve the readability of Python code. Writing clear, readable code shows professionalism.

The following are the rules according to PEP-8:

## 1.8.1 NAMING CONVENTIONS

When you write Python code, you must name a lot of things: variables, functions, classes, packages, and so on. Choosing a sensible name helps to understand from the name, what a certain variable, function, or class represents.

Naming Styles:

1. Function: Use a lowercase word or words. Separate words by underscores to improve readability. Ex: function, my\_function

2. Variable: Use a lowercase single letter, word, or words. Separate words with underscores to improve readability. Ex: x, var, my\_variable

Note: Never use l, O, or I single letter names as these can be mistaken for 1 and 0, depending on typeface:

O = 2 # This may look like you are trying to reassign 2 to zero

3. Class: Start each word with a capital letter. Do not separate words with underscores. This style is called camel case. Ex: Model, MyClass
4. Method: Use a lowercase word or words. Separate words with underscores to improve readability. Ex: class\_method, method
5. Constant: Use an uppercase single letter, word, or words. Separate words with underscores to improve readability. Ex: CONSTANT, MY\_CONSTANT, MY\_LONG\_CONSTANT
6. Module: Use a short, lowercase word or words. Separate words with underscores to improve readability. Ex: module.py, my\_module.py
7. Package: Use a short, lowercase word or words. Do not separate words with underscores. Ex: package, mypackage

How to Choose Names

Choosing names for your variables, functions, classes, and so forth can be challenging. You should put a fair amount of thought into your naming choices when writing code as it will make your code more readable. The best way to name your objects in Python is to use descriptive names to make it clear what the object represents. Single-letter lowercase names, like x can be used as the argument



of a mathematical function but not for storing a person's name as a string as it would not be much clear to the reader.

*Example:*

```
>>> # Not recommended
>>> x = 'John Smith'
>>> y, z = x.split()
>>> print(z, y, sep = ', ')
O/P: 'Smith, John'
```

This will work, but you will have to keep track of what x, y, and z represent. It may also be confusing for collaborators. The better way to write the above code is

*Example:*

```
>>> # Recommended
>>> name = 'John Smith'
>>> first_name, last_name = name.split()
>>> print(last_name, first_name, sep = ', ')
O/P: 'Smith, John'
```

Similar approach can be followed for functions, classes etc. as well.

## 1.8.2 CODE LAYOUT

PEP-8 recommends 79 characters line limit. How you lay out your code has a huge role in how readable it is.

1. **Blank Lines:** Vertical whitespace, or blank lines, can greatly improve the readability of your code. Code that is bunched up together can be overwhelming and hard to read. Similarly, too many blank lines in your code makes it look very sparse, and the reader might need to scroll more than necessary.

The three guidelines for this are:

- a. Surround top-level functions and classes with two blank lines.  
Top-level functions and classes should be self-contained and handle separate functionality. It makes sense to put extra vertical space around them, so that it is clear they are separate:

```
class MyFirstClass:
    pass

class MySecondClass:
    pass
```

```
def top_level_function():  
    return None
```

- b. Surround method definitions inside classes with a single blank line  
Inside a class, functions are all related to one another. It's good practice to leave only a single line between them:

```
class MyClass:  
    def first_method(self):  
        return None  
    def second_method(self):  
        return None
```

- c. Use blank lines sparingly inside functions to show clear steps.  
Sometimes, a complicated function must complete several steps before the return statement. To help the reader understand the logic inside the function, it can be helpful to leave a blank line between each step.

```
def calculate_variance(number_list):  
    sum_list = 0  
    for number in number_list:  
        sum_list = sum_list + number  
    mean = sum_list / len(number_list)  
  
    sum_squares = 0  
    for number in number_list:  
        sum_squares = sum_squares + number**2  
    mean_squares = sum_squares / len(number_list)  
  
    return mean_squares - mean**2
```

2. Maximum Line Length and Line Breaking: PEP 8 suggests lines should be limited to 79 characters. This is because it allows you to have multiple files open next to one another, while also avoiding line wrapping. PEP 8 outlines ways to allow statements to run over several lines. Python will assume line continuation if code is contained within parentheses, brackets, or braces:

```
def function(arg_one, arg_two,  
            arg_three, arg_four):  
    return arg_one
```

If line breaking needs to occur around binary operators, like + and \*, it should occur before the operator.

```
total = (first_variable
        + second_variable
        - third_variable)
```

3. Indentation: Indentation, or leading whitespace, is extremely important in Python. The indentation level of lines of code in Python determines how statements are grouped together.

```
x = 3
if x > 5:
    print('x is larger than 5')
```

- **The key indentation rules laid out by PEP 8 are the following:**

- Use 4 consecutive spaces to indicate indentation.
- Prefer spaces over tabs.

- **Indentation Following Line Breaks**

When you are using line continuations to keep lines to under 79 characters, it is useful to use indentation to improve readability. It allows the reader to distinguish between two lines of code and a single line of code that spans two lines. There are two styles of indentation you can use.

The first of these is to align the indented block with the opening delimiter:

```
def function(arg_one, arg_two,
            arg_three, arg_four):
    return arg_one
```

Sometimes you can find that only 4 spaces are needed to align with the opening delimiter. This will often occur in if statements that span multiple lines as the if, space, and opening bracket make up 4 characters. In this case, it can be difficult to determine where the nested code block inside the if statement begins:

```
x = 5
if (x > 3 and
    x < 10):
    print(x)
```

**In this case, PEP 8 provides two alternatives to help improve readability:**

- Add a comment after the final condition. Due to syntax highlighting in most editors, this will separate the conditions from the nested code:

```
x = 5
if (x > 3 and
    x < 10):
    # Both conditions satisfied
    print(x)
```

- Add extra indentation on the line continuation:

```
x = 5
if (x > 3 and
    x < 10):
    print(x)
```

4. **Comments:** You should use comments to document code as it is written. It is important to document your code so that you, and any collaborators, can understand it. When you or someone else reads a comment, they should be able to easily understand the code the comment applies to and how it fits in with the rest of your code.

- Here are some key points to remember when adding comments to your code:
  - Limit the line length of comments and docstrings to 72 characters.
  - Use complete sentences, starting with a capital letter.
  - Make sure to update comments if you change your code.
- **Block Comments:** Use block comments to document a small section of code. They are useful when you must write several lines of code to perform a single action, such as importing data from a file or updating a database entry. They are important as they help others understand the purpose and functionality of a given code block.

**PEP 8 provides the following rules for writing block comments:**

- Indent block comments to the same level as the code they describe.
- Start each line with a # followed by a single space.
- Separate paragraphs by a line containing a single #.

**Example:**

```
for i in range(0, 10):
    # Loop over i ten times and print out the value of i, followed by a
    # new line character
    print(i, '\n')
```

- **Inline Comments:** Inline comments explain a single statement in a piece of code. They are useful to remind you, or explain to others, why a certain line of code is necessary. Here is what PEP 8 has to say about them:
  - Use inline comments sparingly.
  - Write inline comments on the same line as the statement they refer to.
  - Separate inline comments by two or more spaces from the statement.
  - Start inline comments with a # and a single space, like block comments.
  - Do not use them to explain the obvious.

```
x = 5 # This is an inline comment
```

- **Documentation Strings:** Documentation strings, or docstrings, are strings enclosed in double ("""") or single (') quotation marks that appear on the first line of any function, class, method, or module. You can use them to explain and document a specific block of code. There is an entire PEP, PEP 257, that covers docstrings.

**The most important rules applying to docstrings are the following:**

- Surround docstrings with three double quotes on either side, as in """This is a docstring""".
- Write them for all public modules, functions, classes, and methods.
- Put the """ that ends a multiline docstring on a line by itself:

```
def quadratic(a, b, c, x):
    """Solve quadratic equation via the quadratic formula.
```

*A quadratic equation has the following form:*

$$ax^{**2} + bx + c = 0$$

*There always two solutions to a quadratic equation: x\_1 & x\_2.*

```
"""
```

$$x\_1 = (-b + (b^{**2} - 4*a*c)^{(1/2)}) / (2*a)$$

$$x\_2 = (-b - (b^{**2} - 4*a*c)^{(1/2)}) / (2*a)$$

```
return x_1, x_2
```

- For one-line docstrings, keep the """ on the same line:

```
def quadratic(a, b, c, x):
    """Use the quadratic formula"""
    x_1 = (-b + (b^{**2} - 4*a*c)^{(1/2)}) / (2*a)
    x_2 = (-b - (b^{**2} - 4*a*c)^{(1/2)}) / (2*a)
```

*return x\_1, x\_2*

## 5. Programming Recommendations

- Do not compare Boolean values to True or False using the equivalence operator. You will often need to check if a Boolean value is True or False.
- Use the fact that empty sequences are false in if statements. If you want to check whether a list is empty, you might be tempted to check the length of the list. If the list is empty, it is length is 0 which is equivalent to False when used in an if statement.
- Use `is` not rather than `not ... is` in if statements. If you are trying to check whether a variable has a defined value, there are two options. The first is to evaluate an if statement with `x is not None`, second option would be to evaluate `x is None` and then have an if statement based on `not`.
- Do not use `if x:` when you mean `if x is not None:`.
- Use `.startswith()` and `.endswith()` instead of slicing.

## 1.8.3 MIDDLEWARE CUSTOM SSO SETUP

1. Install the following packages using pip command inside your virtual environment and in the correct path of your project:

```
PyJWT==1.7.1
requests==2.25.0
cryptography==3.0
```

2. Write a **custom authentication middleware** file for the **SSO** process from the middleware. Also write a file to verify the **JWT** named **verify\_jwt.py**. Refer to the images below for an example on how to write the custom authentication file.]

The below snippet shows the **regionmiddleware.py** file

```
8 from UserRegionMap.models import UserRegion
9 from django.utils.deprecation import MiddlewareMixin
10 from django.http import HttpResponse
11 from django.conf import settings
12 import json
13 import jwt
14 from UserRegionMap.verify_jwt import verify_token
15
16
17 class CustomAuthMiddleware(MiddlewareMixin):
18
19     def process_request(self, request):
20         print('entered custom middleware')
21
22         auth_header = request.META.get('HTTP_AUTHORIZATION', '')
23
24         if not auth_header:
25             if request.body:
26                 body_unicode = request.body.decode('utf-8')
27                 body_data = json.loads(body_unicode)
28                 if "token" not in body_data:
29                     print("no auth header and no token in body")
30                     return HttpResponse("Unauthorised", status=401)
31                 else:
32                     print("userRegionMap call")
33             else:
34                 print("no auth header and no body")
35                 return HttpResponse("Unauthorised", status=401)
```

```

36         else:
37             if request.body:
38                 body_unicode = request.body.decode('utf-8')
39                 body_data = json.loads(body_unicode)
40                 if "idToken" not in body_data:
41                     print("no idToken as part of POST body")
42                     return HttpResponse("Unauthorised", status=401)
43             else:
44                 decoded_token = verify_token(body_data['idToken'])
45                 if decoded_token:
46                     token_type, _, credentials = auth_header.partition([' '])
47                     if token_type == "Bearer":
48                         print("reached decoding part")
49                         decoded = jwt.decode(
50                             credentials, settings.JWT_SECRET, algorithms=['HS256'])
51                         if decoded_token['unique_name'].split('@')[0] == decoded['userID']:
52                             if (UserRegion.objects.filter(userid=decoded['userID'],
53                                 isadmin=decoded['isAdmin'], isactive=1)):
54                                 print("user valid")
55                             else:
56                                 print("here")
57                                 return HttpResponse("Unauthorised", status=401)
58                     else:
59                         print(decoded_token['unique_name'].split(
60                             '@')[0], decoded['userID'])
61                         print("userid idToken does not match jwt token")
62                         return HttpResponse("Unauthorised", status=401)
63
64         else:
65             print("user idToken is not valid")
66             return HttpResponse("Unauthorised", status=401)
67     else:
68         print("no body in request")
69         return HttpResponse("Unauthorised", status=401)

```

The above snippets show the code for the **custom middleware authentication**.

- When a user tries to login to your application the user's **credentials** goes through this file to verify if the user is **authorized** or not.
- Line 22 the **authorization header** of the user is stored which is passed in the **auth header** that contains the token while calling the API.
- If the auth header is empty the user is denied from the access.
- If the auth header is valid the information from the **request body** is loaded.
- If the keyword "**idToken**" is not present in the request body, then **access** is **denied** else the "**idToken**" is decoded using the **verify\_token** function that is present in the **verify.jwt** file.
- Now, if the decoded token is True, then we store the token type and credentials in line 46.
- If the token type is '**bearer**' we decode the **credentials** in line 49 using the specified **jwt secret key** present in the project setting file and the specified **algorithm** and store it in the variable called **decoded**.
- If the unique name on the **decoded token** matches the User Id decoded above, then in line 52 we check if the User Id is present **isactive** is set to 1 and **isadmin** is present in the decoded value above then the user is valid.
- Otherwise, in the following lines we show user is **unauthorized**.

The below snippet shows the code for **verify\_jwt.py** file.

- When the **verify\_token()** is called in the **regionmiddleware.py** file, the following runs.
- In **verify\_token()** we create **open\_id\_metadata\_url** which uses the **settings** files to get the **tenant id** as shown on line 81.
- Now we make a **requests.get()** call on **open\_id\_metadata\_url** and convert this into json format in line 83 and 84.
- Then we check if the jwt received in the **verify\_token()** is true or not. If **jwt** is True, then we validate it line 93 using **validate\_jwt()** as shown below.

- **validate\_jwt()** returns the decoded value
- In **verify\_token()** line 93 the decoded variable has the decoded information.
- Username is extracted from this decoded variable and is checked if it is same domain as we want in line 99 and 100. If is True, then the decoded information is return to **regionmiddleware.py** file.

```

80 def verify_token(jwt):
81     open_id_metadata_url = settings.OPEN_ID_METADATA_URL.format(
82         **{'tenant': settings.TENANT_ID})
83     r = requests.get(open_id_metadata_url)
84     metadata_response = r.json()
85     jwks_uri = metadata_response["jwks_uri"]
86     r = requests.get(jwks_uri)
87     jwks_response = r.json()
88     # configuration, these can be seen in valid JWTs from Azure B2C:
89     if not jwt:
90         print('Please pass a valid JWT')
91         return None
92     try:
93         decoded = validate_jwt(jwks_response, jwt)
94     except Exception as ex:
95         traceback.print_exc()
96         print('The JWT is not valid!')
97         return None
98     else:
99         username = decoded['unique_name']
100        if username.split('@')[1] == 'effem.com':
101            print('The JWT is valid!')
102            return decoded
103        else:
104            print('The JWT is not valid!')
105            return None
106

```

The snippet below shows the **validate\_jwt()**

```

65 def validate_jwt(jwks_response, jwt_to_validate):
66     public_key = get_public_key(jwks_response, jwt_to_validate)
67     valid_audiences = [settings.CLIENT_ID]
68     decoded = jwt.decode(jwt_to_validate,
69                          public_key,
70                          verify=True,
71                          algorithms=['RS256'],
72                          audience=valid_audiences,
73                          issuer=settings.ISSUER)
74     # do what you wish with decoded token:
75     # if we get here, the JWT is validated
76     # print(decoded)
77     return decoded

```

- Add the **custom middleware authentication** in the settings file in the same order as shown in the snippet below.



```

91 MIDDLEWARE = [
92     'django.middleware.security.SecurityMiddleware',
93     'django.contrib.sessions.middleware.SessionMiddleware',
94     'corsheaders.middleware.CorsMiddleware', # new
95     'django.middleware.common.CommonMiddleware',
96     'django.contrib.auth.middleware.AuthenticationMiddleware',
97     'django.contrib.messages.middleware.MessageMiddleware',
98     'django.middleware.clickjacking.XFrameOptionsMiddleware',
99     'UserRegionMap.regionmiddleware.CustomAuthMiddleware',
100 ]

```

## STEPS TO CLONE THE BRANCH IN YOUR LOCAL SYSTEM

1. In your local system create a folder for the project.
2. Install **anaconda** or **mini conda** in your system.
3. Open the conda command prompt and navigate to the above created folder.
4. Create a virtual environment for the project using the following commands:

In the terminal client enter the following where yourenvname is the name you want to call your environment, and replace x.x with the Python version you wish to use.

```
conda create -n yourenvname python=x.x anaconda
```

Press y to proceed. This will install the Python version and all the associated anaconda packaged libraries at "path\_to\_your\_anaconda\_location/anaconda/envs/yourenvname"

1. Activate your virtual environment using:

```
conda activate yourenvname
```

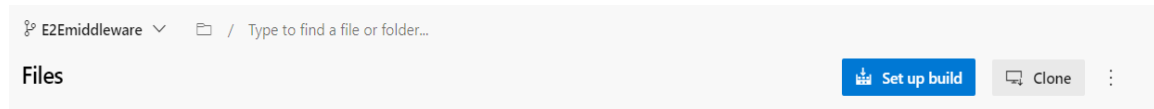
Activating a conda environment modifies the **PATH** and **shell** variables to point to the specific isolated Python set-up you created. The command prompt will change to indicate which conda environment you are currently in by prepending (**yourenvname**).

Example:

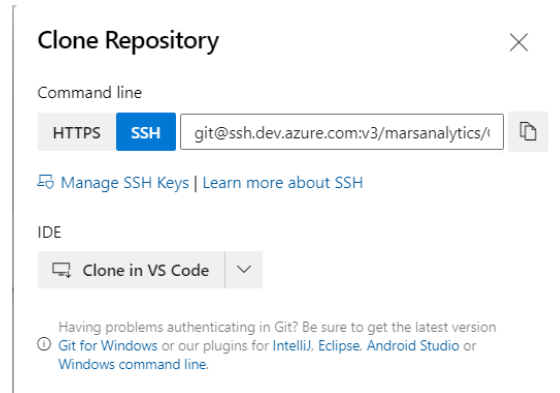
```
(e2e) C:\Users\sahacha>
```

2. Open VS code and in the terminal navigate to the correct path.
3. Copy the **SSH link** of your branch from the project repository in the Azure DevOps

## Click on clone



## Copy the SSH link



4. In your local system in the Command prompt of VsCode type:

```
PS C:\Users\sahacha\OneDrive - Mars Inc\Documents\MARS> git clone git@ssh.dev.azure.com:v3/marsanalytics/
```

5. This will initialize a local repository of the remote repository in your system.

6. Now run the following commands:

- **git fetch origin** - this syncs all the branches in the remote repository to your local
- **git checkout origin branchname** – this switches to the branch u wish to work on

7. Once you make any changes in the files then save the files and in the VsCode command prompt move to the proper path and run the following commands:

This **adds** all the **changes** you made in the files

```
PS C:\Users\sahacha\OneDrive - Mars Inc\Documents\MARS\global-mw-supplychain-e2e-kpis\src> git add .
```

This **commits** your **changes** with a message

```
PS C:\Users\sahacha\OneDrive - Mars Inc\Documents\MARS\global-mw-supplychain-e2e-kpis\src> git commit -m "Updating the change"
```

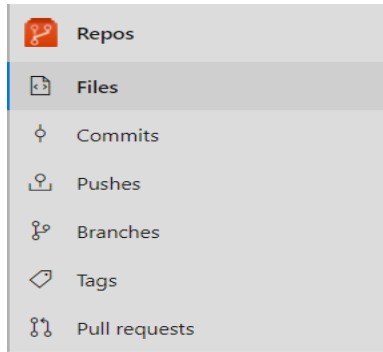
The branch you want to **pull** in and **merge** from

```
PS C:\Users\sahacha\OneDrive - Mars Inc\Documents\MARS\global-mw-supplychain-e2e-kpis\src> git pull origin <branch name>
```

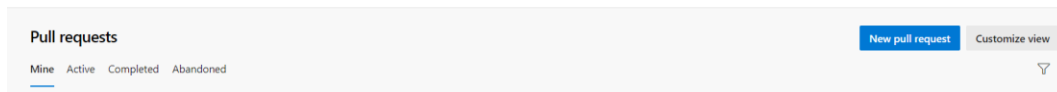
**Push** the **changes** to your branch in the **remote repo**(Azure DevOps)

```
PS C:\Users\sahacha\OneDrive - Mars Inc\Documents\MARS\global-mw-supplychain-e2e-kpis\src> git push origin <your branch name>
```

8. Go to the **Azure DevOps** and create a **Pull request**:



## 9. Create a new pull request



10. Select the **branches** to **pull** from (your branch ex: E2Emiddleware) and the **branch** you want to **push** the **changes** to (master branch) one to the other. Update the title and description of the changes and click on **create**. This will create a pull request and **merge** your changes into the **destination branch** and run the **pipelines** to reflect the changes made in the files.

A screenshot of the 'New pull request' form in GitHub. The form is titled 'New pull request' and shows the source branch as 'E2Emiddleware' and the target branch as 'master'. Below the title, there are tabs for 'Overview', 'Files' (764+), and 'Commits' (100+). The 'Overview' tab is selected. The form has a 'Title' field with the text 'Update' and a 'Description' field with the text 'update'. There is a 'Markdown supported' link and a 'Link work items' link. Below the description, there is a rich text editor with various formatting options. The 'Reviewers' section has a search bar and a button 'Add required reviewers'. The 'Work items to link' section has a search bar and a button 'Clear all'. There is a list of work items, including 'Task 34641: Create config json with ZREPs for modelling, and reader function' which is marked as 'Done'. At the bottom, there is a 'Tags' field and a 'Create' button.