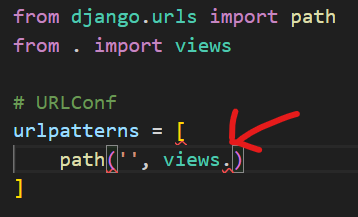


**Adding the Home Page** :

Currently we don’t have a home page in our application and hitting localhost:8000/ endpoint gives this page.



So in the core app, let’s add a new file called *urls.py*.



There is no need to create an explicit view here. *We can use one of the generic views in Django to render a template*.

So let’s import *TemplateView*,

from django.views.generic import TemplateView

Now we will reference this TemplateView and pass the *template\_name* as a keyword argument.

from django.views.generic import TemplateView

from django.urls import path

from . import views

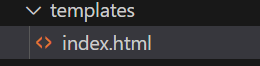
# URLConf

urlpatterns = [

    path('', TemplateView.as\_view(template\_name = 'index.html')) 🡪 *Here*

]

Next we create this index.html, so in the core app, we create a new folder called *templates* and here we add index.html.



Just add a basic HTML template using *! + Enter*.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

</body>

</html>

Note: In the urls.py module, earlier we talked about namespace and templates, so if we have another index.html in other app, that template might take over and replace this one.

So a better way is to namespace our templates. So in the templates folder we create a new folder called core and move index.html inside it.



Now in the urls.py module, we change the template name to core/index.html.

from django.views.generic import TemplateView

from django.urls import path

# URLConf

urlpatterns = [path("", TemplateView.as\_view(template\_name="core/index.html"))]

🡪 *Here*

This way we can prevent this template clashing with another template.

At last we need to link this urlpattern to main urlpatterns of our application.

urlpatterns = [

    path("", include("core.urls")), *🡪 Root url mapped with core.urls*

    path("admin/", admin.site.urls),

    path("playground/", include("playground.urls")),

    path("store/", include("store.urls")),

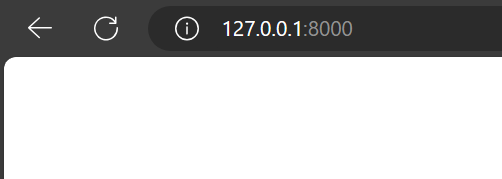
    path("auth/", include("djoser.urls")),

    path("auth/", include("djoser.urls.jwt")),

    path("\_\_debug\_\_/", include(debug\_toolbar.urls)),

]

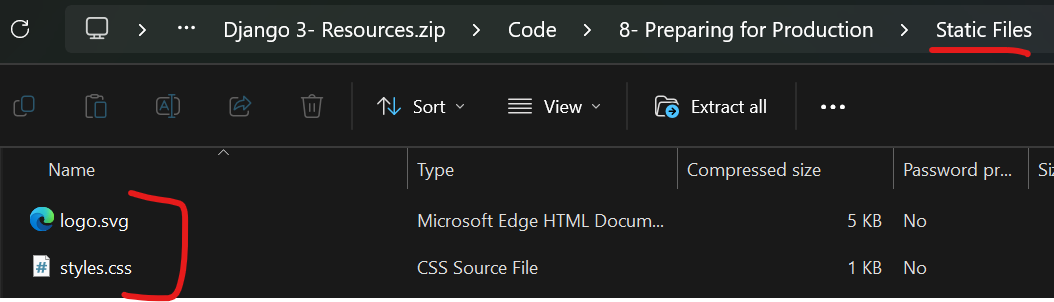
When we refresh this page, we see empty page.



**Adding Static Assets**:

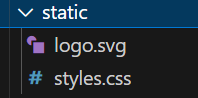
Now let’s make our home page pretty.

In the resources check Static Files folder with two files.



*In Django, we refer to images, stylesheets and JavaScript files that we bundle with our application as static files*. So we are going to add these in our project to make our homepage look beautiful.

In the core app, let’s create a new folder called *static* and add these two static files here.



In our index.html template, we need to use a special tag to load our static assets. So on the top we use,

{% load static %}

Next in the <head> section, we reference our stylesheet using <link> tag,

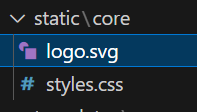
<link rel="stylesheet" href="">

In the *href* attribute, we need to type the path to our CSS file, so for that we will use another special tag called *static* and reference our styles.css file.

<link rel="stylesheet" href="{% static 'styles.css' %}">

But there is a problem with this approach. If we have another file with the same name in another app, that file may take over. So here we use namespacing.

Move these files in static/core folder,



And let’s prefix href static assets with core/styles.css.

In the body section, we will type a bit of HTML.

<body>

    <div class="home-page">

        <img src="{% static 'core/logo.svg' %}" alt="Logo" class="logo"/>

    </div>

</body>

*Entire index.html file looks like this*…

{% load static %}

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <link rel="stylesheet" href="{% static 'core/styles.css' %}">

</head>

<body>

    <div class="home-page">

        <img src="{% static 'core/logo.svg' %}" alt="Logo" class="logo"/>

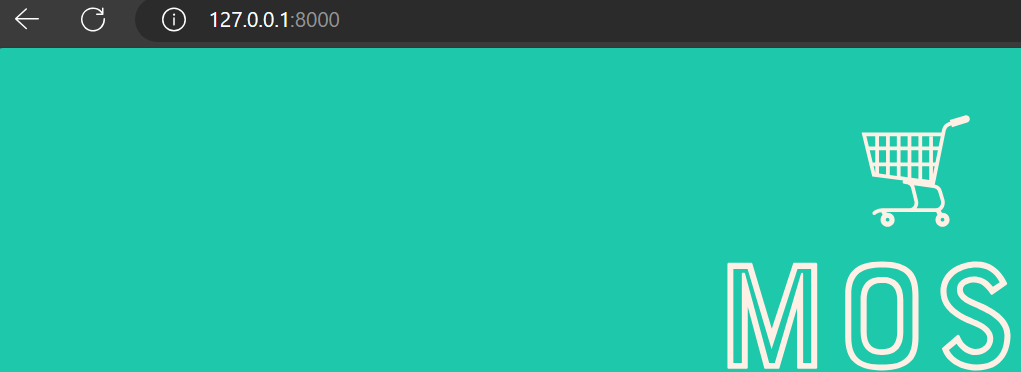
    </div>

</body>

</html>

Note: Refresh the home page after making these changes, if change does not reflect on the page automatically then stop and restart the server.

After refreshing the page,



We see our rendered home page now.

**Collecting Static Assets**:

In the previous lesson we added a couple of static files to the core app and everything magically worked, but we really need to understand what happens under the hood.

Because later when we talk about deployment, we are going to face some issues if we don’t understand how static files are served in django.

***“****When we are in development mode, means debug is turned on. Django goes to every app and if there is a static folder, Django is going to copy all its content to a special folder on disk****”***.

So all these folders, all these static folders from different apps, their content is going to be copied into a single bucket.

However *this is not going to work in production, it only works when debug is turned ON*.

*For production, we have a special command for collecting static files from different apps*, but for that to work first we need to configure a setting.

Let’s go to MEDIA\_ROOT setting,

MEDIA\_URL = "/media/"

MEDIA\_ROOT = os.path.join(BASE\_DIR, "media")

Earlier we talked about media files which are user uploaded files. Here we have two settings called MEDIA\_URL and MEDIA\_ROOT (*it’s a full path to a folder on disk that contains user uploaded files*).

We take all these files and serve them from MEDIA\_URL (*/media/ endpoint*). We have the same concept with static files.

Since we have a STATIC\_URL (*/static/ endpoint*), *we need to configure a STATIC\_ROOT setting, so django knows where our static assets are located on disk*.

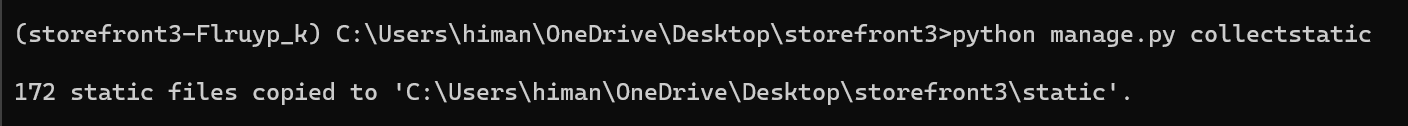
STATIC\_URL = "/static/"

STATIC\_ROOT = os.path.join(BASE\_DIR, "static")

With this setting, now we can explicitly collect static files from different apps.

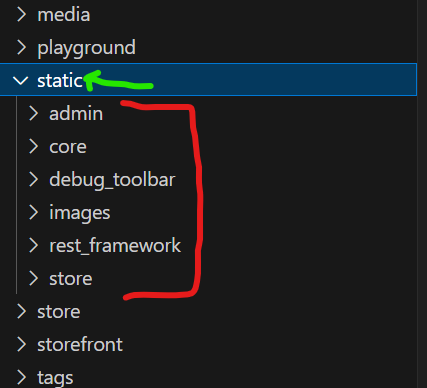
If we run this command on terminal,

python manage.py collectstatic



So we see that 172 static files were copied to the given folder.

A new folder called *static* is created in our project directory,



And it contains all the static files from different apps.

These are the apps that we have listed in our settings module,

INSTALLED\_APPS = [

    "django.contrib.admin",

    "django.contrib.sessions",

    "django.contrib.auth",

    "django.contrib.contenttypes",

    "django.contrib.messages",

    "django.contrib.staticfiles",

    "django\_filters",

    "corsheaders",

    "rest\_framework",

    "djoser",

    "playground",

    "debug\_toolbar",

    "store",

    "tags",

    "likes",

    "core",

]

So if we forget to include an app here, running collectstatic command is not going to copy static files of that app for us.

Therefore in production every time we want to deploy, we should run this command to collect static files from different apps.

Now in development we can add the /static/ folder in our .gitignore file to ignore tracking static files to out git repository.

\*.pyc

/media/

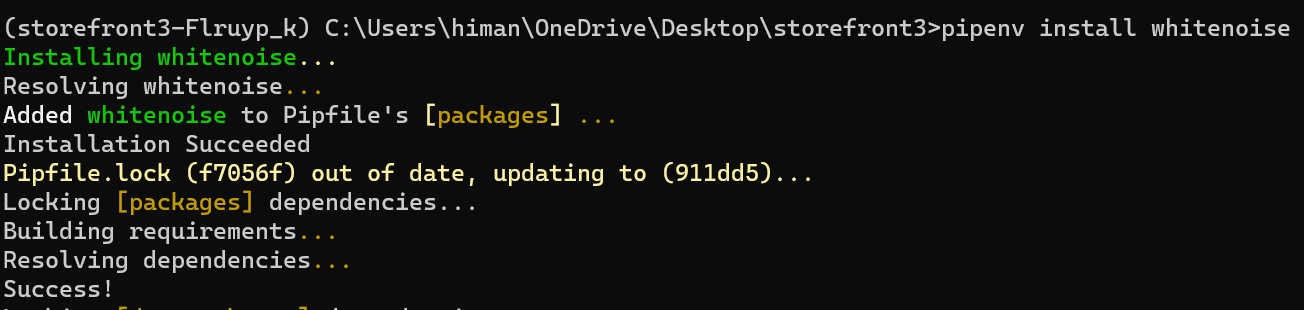
/static/

**Serving Static Assets**:

Django does not support serving static files in production, so even though we have a command for collecting these static files, we cannot serve them.

To add this feature to django, we have to install a library called **whitenoise**.

pipenv install whitenoise



Now we should go to our middleware setting and add *WhiteNoiseMiddleware*.

MIDDLEWARE = [

    "corsheaders.middleware.CorsMiddleware",

    "debug\_toolbar.middleware.DebugToolbarMiddleware",

    "django.middleware.security.SecurityMiddleware",

    "whitenoise.middleware.WhiteNoiseMiddleware", 🡪 *Add Here*

    "django.contrib.sessions.middleware.SessionMiddleware",

Note: This middleware should be as high as possible, but it should come after SecurityMiddleware.

With this simple change, now we can serve static assets in production.

**Configuring Logging**:

Next thing we will talk about is configuring *logging* which is an important and flexible technique for diagnosing problems.



So later after we deploy our application, if some feature is not working we can look at our log file and troubleshoot the issue.

First step is configuring logging. So in the settings module, we will define a new setting called LOGGING and *set it to a dictionary*.

LOGGING = {

}

In this dictionary, we should aways add a couple of keys. One of them is ***version*** (*which we set to 1*), the other is ***disable\_existing\_loggers*** (*as a best practice set it to False because there are other loggers that come with django or other libraries we use, so we don’t want to miss them*).

LOGGING = {

    'version':1,

    'disable\_existing\_loggers':False

}

Then we define another key called ***handlers***, with this we determine what we want to do with log messages (*write them to console / to file etc.*)

LOGGING = {

    'version':1,

    'disable\_existing\_loggers':False,

    'handlers':{

    }

}

In handlers, we can define one or more handlers. For example, we can define one handler called *console* and set it to a dictionary.

LOGGING = {

    'version':1,

    'disable\_existing\_loggers':False,

    'handlers':{

        'console':{

        }

    }

}

Here we set *class* to logging.StreamHandler (*go to logging module of python and reference StreamHandler*)

LOGGING = {

    'version':1,

    'disable\_existing\_loggers':False,

    'handlers':{

        'console':{

            'class': 'logging.StreamHandler' *🡪 Here*

        }

    }

}

*With this class we can write log messages to the console*.

Similarly we can define another handler called *file*, and similarly we set *class* to *logging.FileHandler*.

LOGGING = {

    "version": 1,

    "disable\_existing\_loggers": False,

    "handlers": {

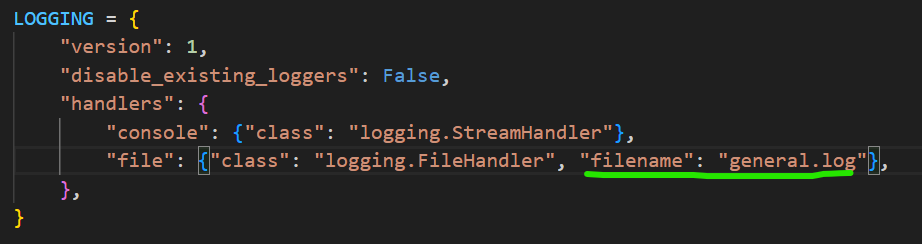
        "console": {"class": "logging.StreamHandler"},

        "file": {"class": "logging.FileHandler"}, *🡪 Here*

    },

}

Since we should also specify the name of the file, so we also set *filename*.



So now we have two handlers, next we define another key called *loggers* (*where we can define one or more loggers*).

For example, here we defined a logger for playground app and this will capture any log messages sent from this app or *we can be more specific and define a logger called* ***playground.views*** *which will only capture log messages sent from views module of playground app*.

LOGGING = {

    "version": 1,

    "disable\_existing\_loggers": False,

    "handlers": {

        "console": {"class": "logging.StreamHandler"},

        "file": {"class": "logging.FileHandler", "filename": "general.log"},

        "loggers":{

            'playground.views':{ *🡪 Here*

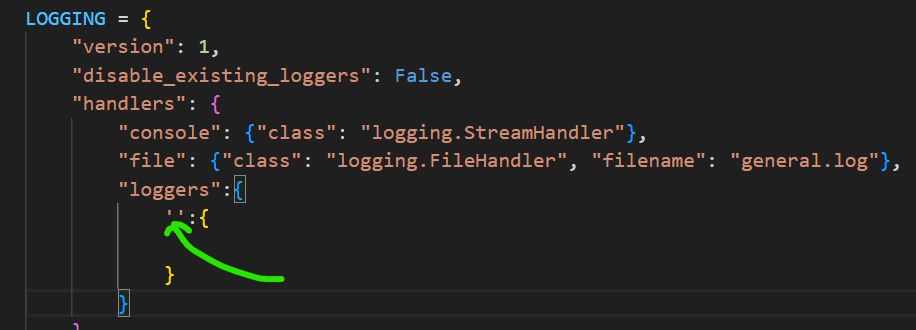
            }

        }

    },

}

*If we remove the logger and use an empty string instead, it will capture all log messages from all apps*.



So this logging framework gives us a lot of flexibility. Now most of the time we don’t want to have different loggers for different apps, this is really unnecessary.

So here we are defining a logger to capture all log messages in our project (*so keep empty string as key*) and in this dictionary first we need to specify is *handlers*, so *once we capture these log messages, what do we do with them like writing them to a console or a file*.

LOGGING = {

    "version": 1,

    "disable\_existing\_loggers": False,

    "handlers": {

        "console": {"class": "logging.StreamHandler"},

        "file": {"class": "logging.FileHandler", "filename": "general.log"},

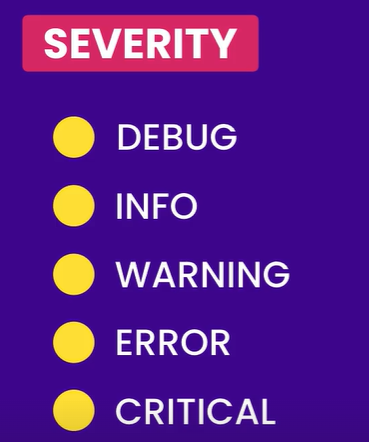
        "loggers": {"": {"handlers": ["console", "file"]}}, *🡪 Here*

    },

}

With this logging framework, we can define different loggers and say “*anything raised in the playground app should be written to the console, but anything raised in store app should be written to both console and a file*”

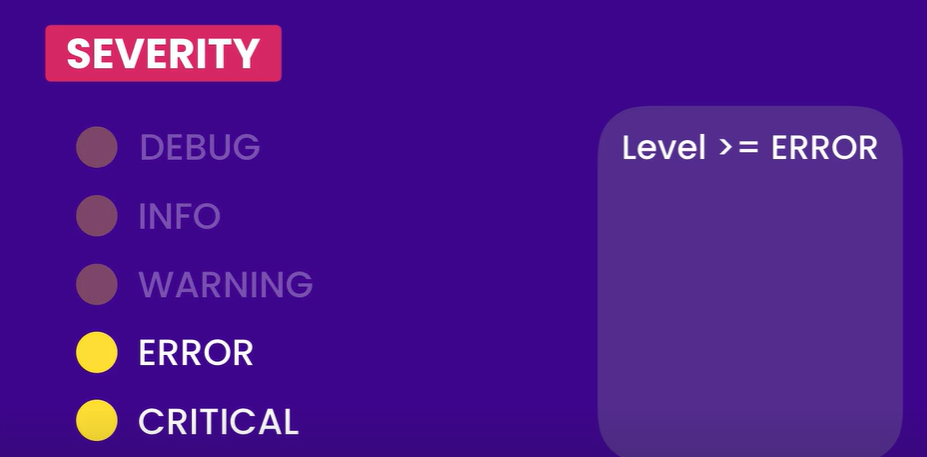
Now the next thing we need to specify here is the *level of log messages*. Log messages has a level or severity.



As we go down this list, the severity of log messages increase (*becomes more important*).

*When defining a logger we need to specify a level and this logger will only capture log messages at this level or higher*.

If we set *level* to **ERROR**. It will only capture ERROR and CRITICAL messages.



So DEBUG, INFO and WARNING messages are not captured.

Here we *do not need to hardcode a particular level like ERROR, we can use an environment variable* which gives us further flexibility.

LOGGING = {

    "version": 1,

    "disable\_existing\_loggers": False,

    "handlers": {

        "console": {"class": "logging.StreamHandler"},

        "file": {"class": "logging.FileHandler", "filename": "general.log"},

        "loggers":{

            '':{

                'handlers': ['console', 'file'],

                'level': os.environ.get('DJANGO\_LOG\_LEVEL', 'INFO') *🡪 Here*

            }

        }

    },

We are reading the environment variable called DJANGO\_LOG\_LEVEL and if this is not set we can assume INFO by default.

*With this implementation we can go to our production server and set this environment variable to capture different types of messages*.

Optionally we can also specify ***formatters*** (*to specify how our log messages should be formatted*).

Here we can define a *simple* formatter that shows only the message.

        'formatters':{

            'simple':

        }

Or we can define a formatter *verbose* that shows more information about the log.

        'formatters':{

            'verbose':

        }

Let’s set verbose to a dictionary and here we need to set a couple of keys.

First key is *format* of our log messages. These log messages have different attributes. So google python – logging – logrecord – attributes.

<https://docs.python.org/3/library/logging.html#logrecord-attributes>



Here we can see all attributes of logrecord class.

So when writing to our logfile, let’s say we need to start with date – time of our log message.

        'formatters':{

            'verbose':{

                'format':'{asctime}' 🡪 *Attribute of log record*

            }

        }

Now let’s say we also want to write level name (*is it a warning, error, critical error or what*).

        'formatters':{

            'verbose':{

                'format':'{asctime} ({levelname})'

            }

        }

If we also want to specify the name of the module that this log message was raised from and followed by the actual log message.

        "formatters": {

            "verbose": {"format": "{asctime} ({levelname}) - {name} - {message}"}

        },

This is the format and we can configure it any way we want to.

Next we specify the *style*

        "formatters": {

            "verbose": {

                "format": "{asctime} ({levelname}) - {name} - {message}",

                "style": '{', *🡪 translated to str.format()*

            }

        },

If we leave the style to a ‘**{**‘ (*left curly brace*), it will translate to *str.format*() means the string we have specified in ‘*format*’ will be passed here.

If we use ‘**$**’ then it will translate to *string.Template* class.

        "formatters": {

            "verbose": {

                "format": "{asctime} ({levelname}) - {name} - {message}",

                "style": "$", *🡪 translated to string.Template*

            }

        },

Complete LOGGING setting, that we are keeping,

LOGGING = {

    'version': 1,

    'disable\_existing\_loggers': False,

    'handlers': {

        'console': {

            'class': 'logging.StreamHandler'

        },

        'file': {

            'class': 'logging.FileHandler',

            'filename': 'general.log',

            'formatter': 'verbose'

        }

    },

    'loggers': {

        '': {

            'handlers': ['console', 'file'],

            'level': os.environ.get('DJANGO\_LOG\_LEVEL', 'INFO')

        }

    },

    'formatters': {

        'verbose': {

            'format': '{asctime} ({levelname}) - {name} - {message}',

            'style': '{'

        }

    }

}

**Logging**:

We will go to views module of the playground app and let’s import *logging* module.

import logging

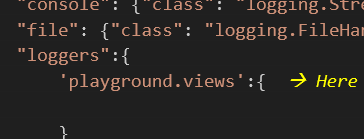
In this module we have a function called getLogger().

logging.getLogger()

Here we pass *\_\_name\_\_* magic attribute and this will translate to playground.views.

logging.getLogger(\_\_name\_\_) 🡪 *playground.views*

This is where our logger bucket comes in (*earlier we saw that we can define a logger called playground.views*).



We get *logger* object from this module and this object has methods for writing different types of log messages.

logger = logging.getLogger(\_\_name\_\_)

logger.info()

logger.debug()

logger.critical() ...

As an example, we are calling *httpbin* as an external service,

def say\_hello(request):

        response = requests.get("https://httpbin.org/delay/2")

        data = response.json()

    return render(request, "hello.html", {"name": "Himanshu"})

Let’s say this is a critical part of our application and in case something goes wrong we want to be able to easily troubleshoot the issue.

So before making this call, we can write an *info* message and write another info message after.

def say\_hello(request):

        logger.info("Calling httpbin") 🡪 *here*

        response = requests.get("https://httpbin.org/delay/2")

        logger.info("Received the response") 🡪 *And here*

        data = response.json()

    return render(request, "hello.html", {"name": "Himanshu"})

If something goes wrong, we need to write a critical message to our logger, so we should wrap this inside a try – catch block like this,

def say\_hello(request):

    try:

        logger.info("Calling httpbin")

        response = requests.get("https://httpbin.org/delay/2")

        logger.info("Received the response")

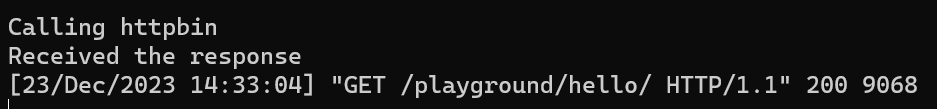
        data = response.json()

    except request.ConnectionError:

        logger.critical("httpbin is offline") 🡪 *critical error catch here*

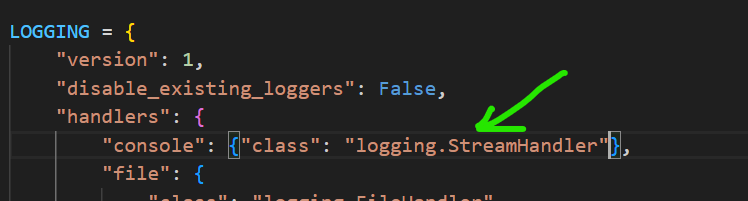
    return render(request, "hello.html", {"name": "Himanshu"})

When we hit this endpoint, we see two log messages…



Here we only see the log messages, we don’t see any additional attributes about this message like DateTime, severity and so on…

It is *because in our logging configuration when defining console handler, we only specified the StreamHandler class and not a formatter*.



We define a formatter in our file handler (*verbose*).

        "file": {

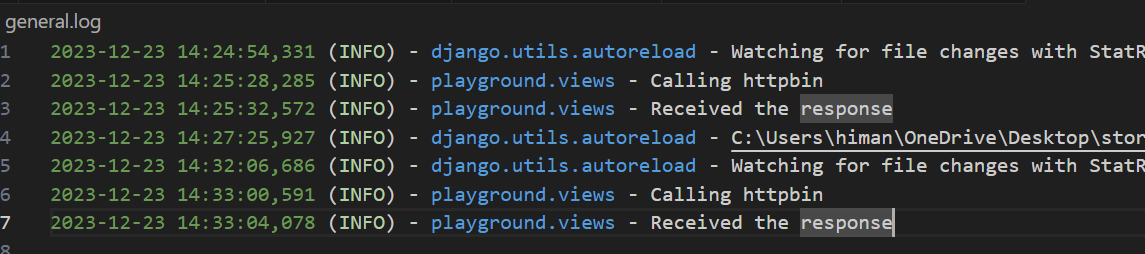
            "class": "logging.FileHandler",

            "filename": "general.log",

            "formatter": "verbose",

        },

In the general.log file,



We have log messages formatted in this particular way.

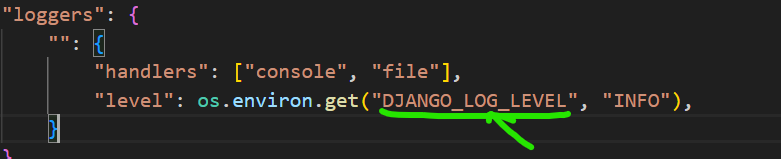
Note: First message is coming from *django.utils.autoreload* because we did not disable other loggers while configuring our LOGGING setting.

LOGGING = {

    "version": 1,

    "disable\_existing\_loggers": False, *🡪 here*

Note: In the future when we deploy our application, if we are not interested in info and warning messages then we can simply set the environment variable for *level* to say a higher level like ERROR and with this we can only see ERROR and CRITICAL messages in our log file.



One last thing before we finish this lesson. Let’s go to our views module.

def say\_hello(request):

    try:

        logger.info("Calling httpbin")

        response = requests.get("https://httpbin.org/delay/2")

        logger.info("Received the response")

        data = response.json()

    except request.ConnectionError:

        logger.critical("httpbin is offline")

    return render(request, "hello.html", {"name": "Himanshu"})

Adding these logger statements are useful but its also making our code a lot verbose, so this is the code we are paying for diagnosing problems.

So we need to use logging wisely. DON’T ever write sensitive information to log because if someone gets access to log file they can see all this information in plain text.

We also need to add this logfile in our .gitignore file (*because it gets updated too often*).

\*.pyc

/media/

/static/

general.log 🡪 *here*

**Managing development and Production settings**:

In every application we need to separate development from production settings. For example, loot at the database settings.

DATABASES = {

    "default": {

        "ENGINE": "django.db.backends.mysql",

        "NAME": "storefront3",

        "HOST": "localhost",

        "USER": "root",

        "PASSWORD": "MyPassword...",

    }

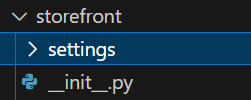
}

Currently we are running MySQL on localhost with a particular user account. This is only for development, so *tomorrow when we want to deploy our application, we don’t have to come back here and modify this setting every single time AND* ***never store database password here***.

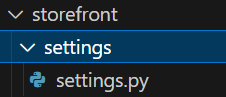
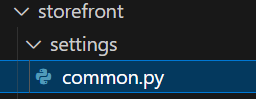
***For production always store password in an environment variable and then read it in this file***.

Let’s see how we can separate development from production settings.

Go to storefront project and create a folder called *settings* here.

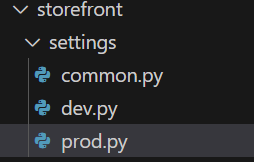


Then we move our settings.py module into this folder and rename it to *common.py*.

 🡪 🡪 

Now we are going to have three files in this folder, one includes all the settings that are common across different environments and then *we are going to have environment specific files* which is one file for development and another for production (*also potentially for other environments like staging and test*).

So let’s create files called dev.py & prod.py.



In each file we need to import all the settings from common.py module.

from .common import \*

Now let’s start moving stuff one by one from common.py.

We will start from SECRET\_KEY setting.

# SECURITY WARNING: keep the secret key used in production secret!

SECRET\_KEY = "django-insecure-hs6j037urx6iav+7#10%-vu4l4f5@@-1\_zo)oft4g7$vf2$jmp"

Notice, the warning. We don’t want to store the secret key in plain text in this file, this is only for development. In production we should keep it in environment variable.

So cut these lines and move them to dev.py.

from .common import \*

SECRET\_KEY = "django-insecure-hs6j037urx6iav+7#10%-vu4l4f5@@-1\_zo)oft4g7$vf2$jmp"

In prod.py.

import os

from .common import \*

SECRET\_KEY = os.environ["SECRET\_KEY"]

Next is DEBUG = *True* setting, we want to turn this on only in development, so move this to dev.py as well.

from .common import \*

DEBUG = True

SECRET\_KEY = "django-insecure-hs6j037urx6iav+7#10%-vu4l4f5@@-1\_zo)oft4g7$vf2$jmp"

In production, we need to set this to false.

import os

from .common import \*

DEBUG = False

SECRET\_KEY = os.environ["SECRET\_KEY"]

When we turn off debugging, we need to set another setting called ALLOWED\_HOSTS. (*with this setting we allow server or servers that can run this application, only required if debug is turned OFF*). So move this to production settings file.

import os

from .common import \*

DEBUG = False

SECRET\_KEY = os.environ["SECRET\_KEY"]

ALLOWED\_HOSTS = [] 🡪 *Here*

Note: Currently we don’t have a production server, in next section we will provision a production server in Heroku, once we have that we can store its address here.

Next we have INSTALLED\_APPS, MIDDLEWARE etc. which are common across all environments.

So we move directly to DATABASES setting, let’s move it to our development file.

from .common import \*

DEBUG = True

SECRET\_KEY = "django-insecure-hs6j037urx6iav+7#10%-vu4l4f5@@-1\_zo)oft4g7$vf2$jmp"

DATABASES = {

    "default": {

        "ENGINE": "django.db.backends.mysql",

        "NAME": "storefront3",

        "HOST": "localhost",

        "USER": "root",

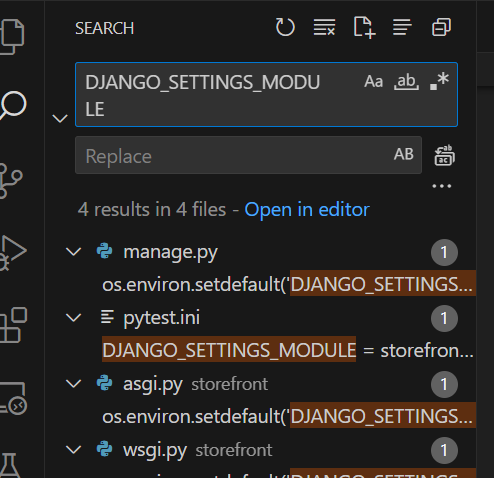
        "PASSWORD": "MyPassword",

    }

}

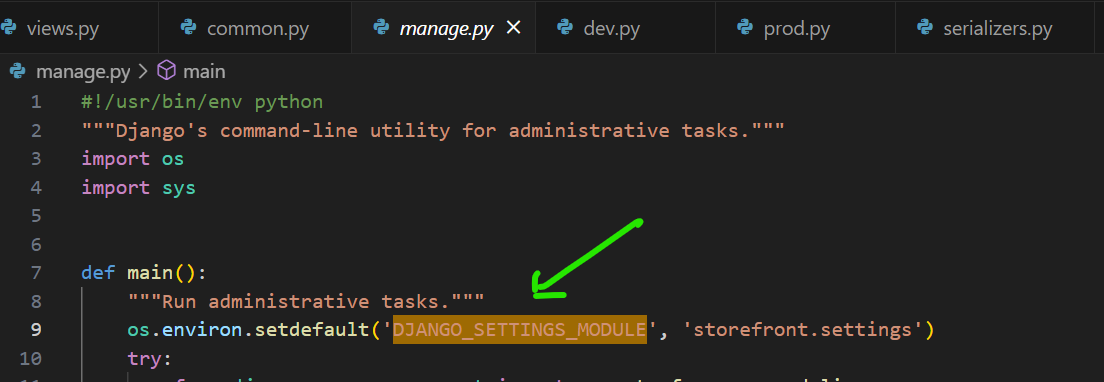
For production, we will have a different approach for configuring our database.

So we are done with these files. Now *we need to do a search in our project and anywhere we have a reference to django settings module, we need to make a slight change there*.



We have all these files that have a reference to this variable. Let’s modify them one by one.

First one is in manage.py.



When we run python manage.py runserver command, manage.py tries to get this variable and if its not set, its set to storefront.settings (*previous settings module before we move it to settings folder*).

Now we need to change it to storefront.settings.dev.

def main():

    """Run administrative tasks."""

    os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "storefront.settings.dev")

In production we will set this to storefront.settings.prod, so when we run this application in a production environment our production settings will come into effect.

Next reference is in pytest.ini

[pytest]

DJANGO\_SETTINGS\_MODULE = storefront.settings.dev

In asgi.py

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "storefront.settings.dev")

**Sidenote**: Celery was not configured, if it was then we needed to change there to settings.dev as well

Last one is wsgi.py

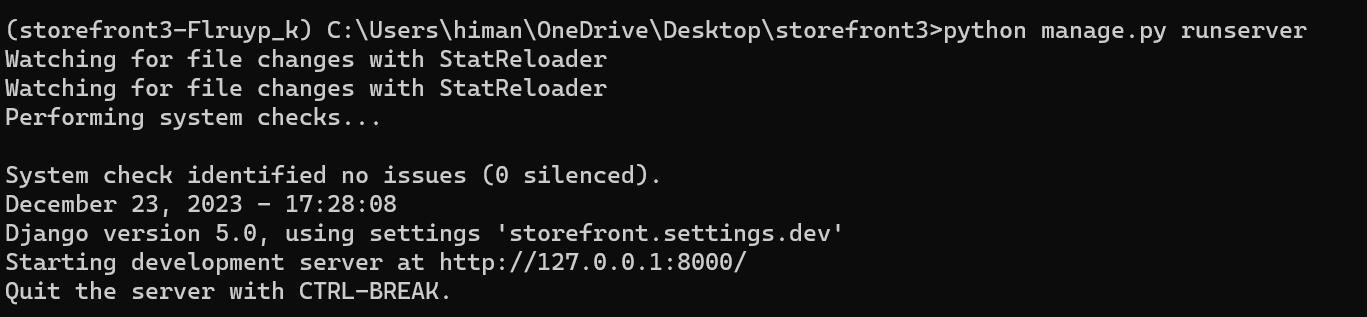
import os

from django.core.wsgi import get\_wsgi\_application

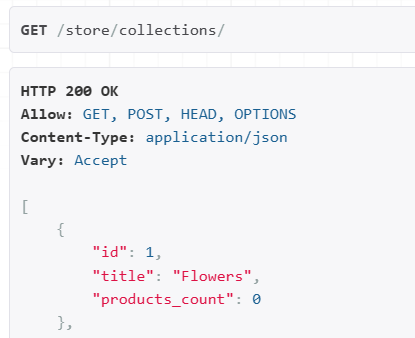
os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "storefront.settings.dev")

application = get\_wsgi\_application()

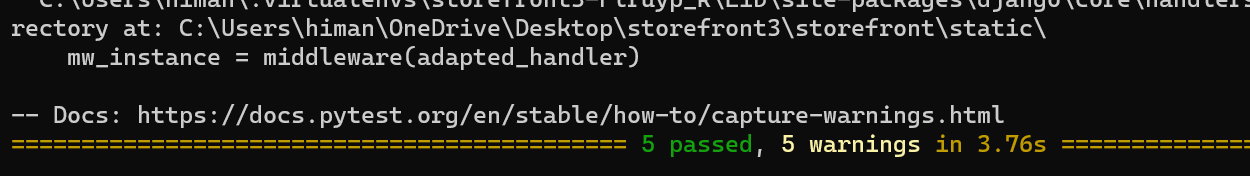
Now let’s make sure we have not broken anything,



Our application loads, now let’s check our endpoints.



Also check if tests are running or not.

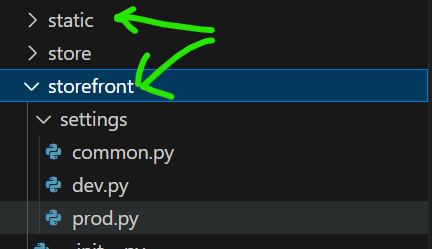


All our tests passed but we have 5 warnings. Due to this

UserWarning: No directory at: C:\Users\himan\OneDrive\Desktop\storefront3\storefront\static\

mw\_instance = middleware(adapted\_handler)

It is trying to find \static\ directory in storefront folder.



But its inside the root folder.

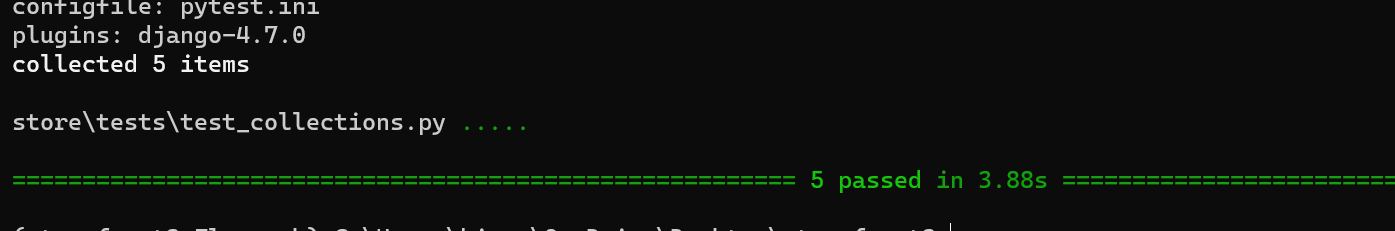
The reason we are seeing this warning is because we restructured our settings module

Go to common.py module and add *parent* at the end of BASE\_DIR setting (*because it is pointing to storefront directory not the root*).

# Build paths inside the project like this: BASE\_DIR / 'subdir'.

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

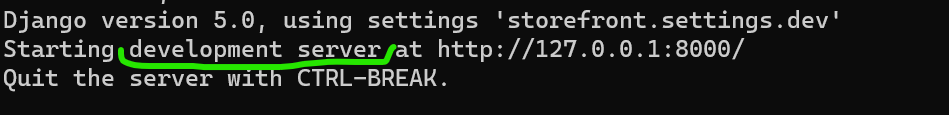
If we run pytest again,



We get no warnings now.

**Serving the Application with Gunicorn**:

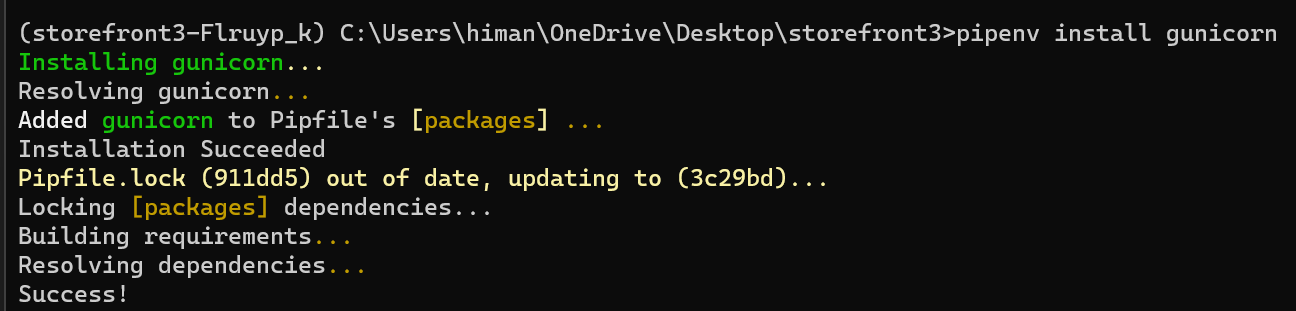
So far we have been using this development web server that is built into Django.



Now this is great for development, because every time we change our code, this web server automatically restarts our application.

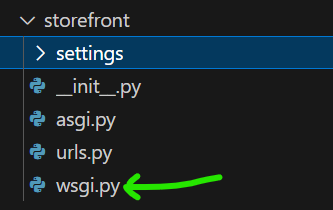
But this server is only meant for development, *in a production environment, we need a fast and robust web server* and for that we are going to use *gunicorn* (*short for green unicorn*).

pipenv install gunicorn



Now to start our application with *gunicorn*, we start with gunicorn and then specify the name of a *special module* in our application.

Go to storefront project directory and find a file called *wsgi*.py (*viz - ge*e)



WSGI stands for *webserver gateway interface*.

import os

from django.core.wsgi import get\_wsgi\_application

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "storefront.settings.dev")

application = get\_wsgi\_application()

In this file, first we are giving DJANGO\_SETTINGS\_MODULE (*environment variable*) a default value (*storefront.settings.dev*) and then we call get\_wsgi\_application to get an application instance.

*So when launching our application with gunicorn, we need to specify wsgi.py module as the entry point*.

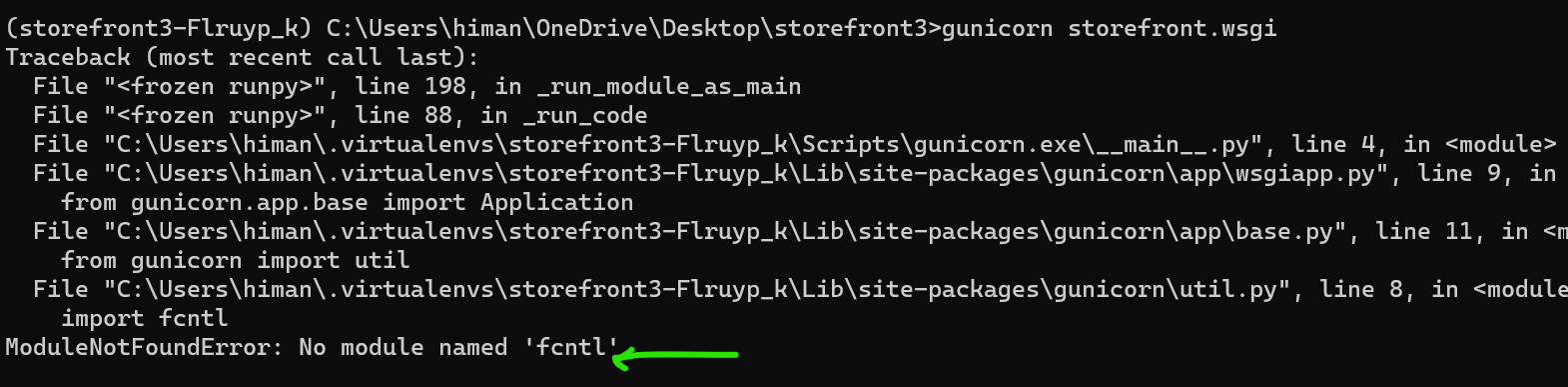
So in the terminal,

gunicorn storefront.wsgi

We are going to the storefront folder and loading *wsgi* module

**Personal Side note**:

We are getting this error.



It is because gunicorn is not compatible with windows as ‘fcntl’ module can only be found in UNIX system.

Note: Unlike the development server that comes with django, gunicorn does not pick changes in our code. So if we make any changes we have to manually start our webserver.

We don’t have to worry about it for now, because in our development environment, we will keep using the default development server.