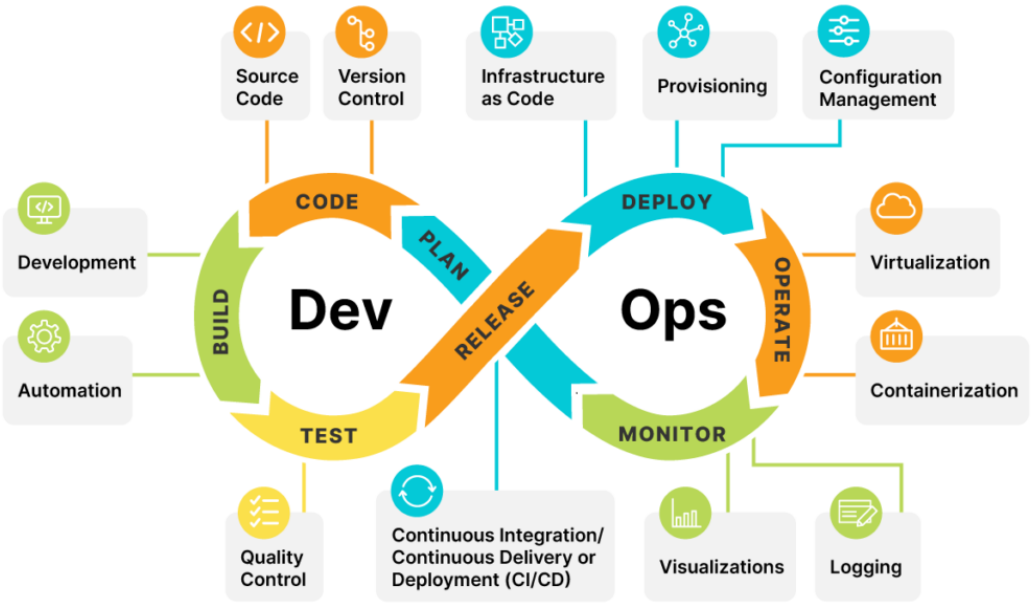
INTERNATIONAL INSTITUTE OF INFORMATION & TECHNOLOGY, BANGALORE

SOFTWARE PRODUCTION ENGINEERING, MAJOR PROJECT

REAL ESTATE APP WITH DEVOPS



SUBHAM BASU ROY CHOWDHURY MT2022118
VISHAL SINGH MT2022133

Table of Content:-

Project Summary for Realest Estate:

This is a large project that shows how to implement a real estate application with a variety of different features.
Features:

- Log in and log out for a common user. Both routes are protected by JSON Web Token.
- New User creation.
- Uploading images and storing them inside a folder segregated by currency date and month and year.
- Creation of admin using the superuser command in Django to update the details of the real estate by the respective agents.
- Implementation of Pagination and usage of custom Django Admin panel.
- Sending email with the help of google Gmail API and SMTP Port.

Tech Stack Used here:

1. Development:
    - **Language**: Python + Javascript
    - **Backend**: Django, Django Rest Framework
    - **Database**: MySQL
    - **Frontend**: React

2. DevOps:
    - **Version Control**: Git and GitHub
    - **CI/CD**: Jenkins
    - **Build**: Pip(python) + npm(react)
    - **Containerization**: Docker + Docker-Compose
    - **Configuration Management**: Ansible
    - **Monitoring**: ELK Stack
    - **Ngrok**: For tunneling the Jenkins from localhost port to the public IP

- **Testing**: Postman
- **Logging**: Python logging library

Important Project Links:

Backend:

- GitHub: https://github.com/testusername190/Realest_Estate_Backend.git
- DockerHub (Image): https://hub.docker.com/repository/docker/sbrc1996/realest_estate/general

Frontend:

- GitHub: https://github.com/testusername190/Realest_Estate_Frontend.git
- DockerHub(Image): https://hub.docker.com/r/vishalsin25/spe-major-front

DevOps:

What is DevOps?

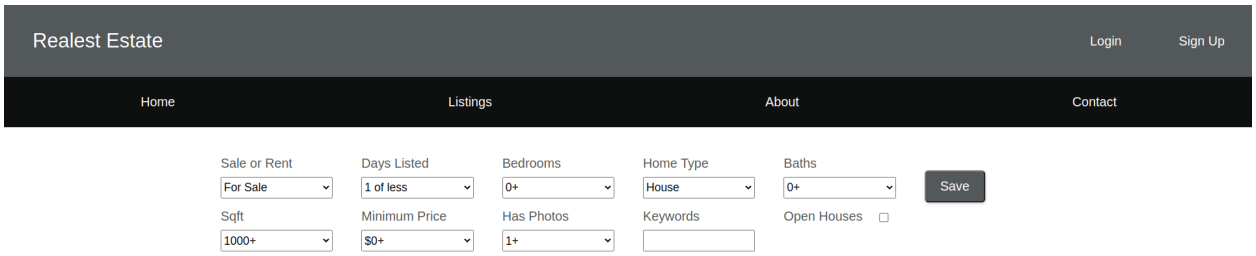
DevOps is a group of concepts emerging from the collision of two trends. It combines software development with IT ops. It aims to decrease SDLC time and provide continuous delivery and integration functionality to produce high-quality software.

Why DevOps?

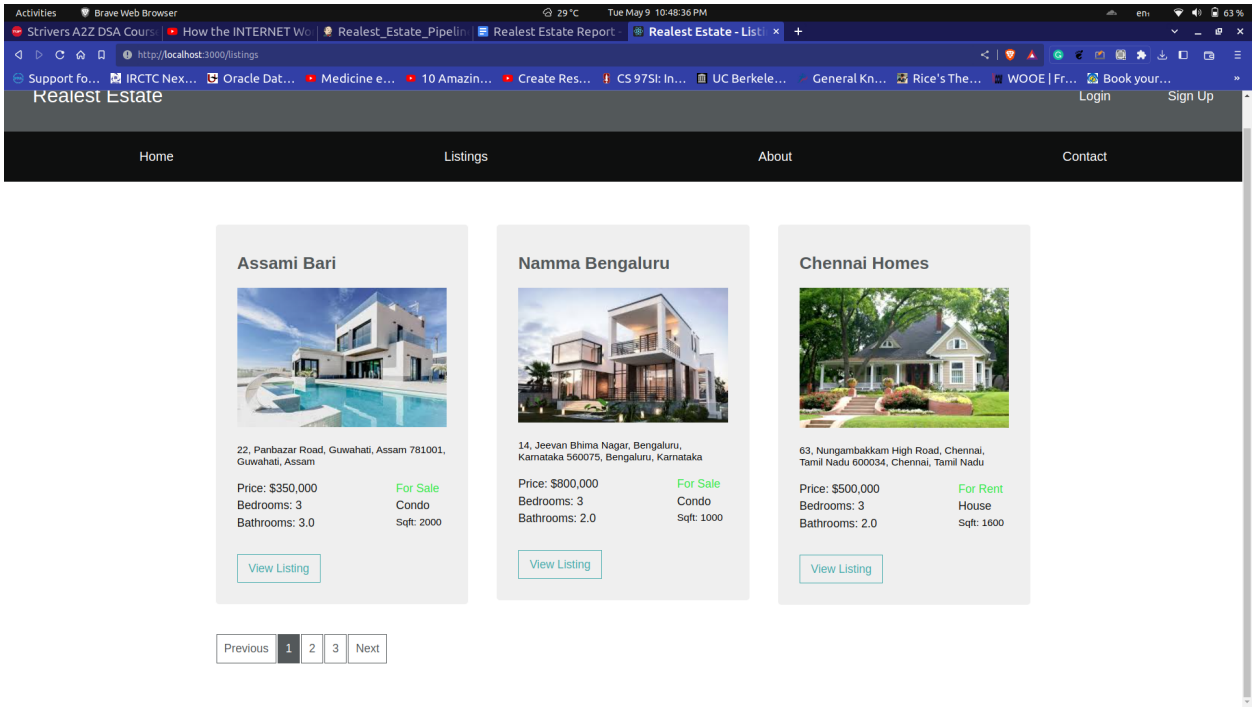
It has now become important to include DevOps in software development as it enables faster development of products, faster bug fixes, and easier maintenance as compared to traditional methods

Application Screenshots:

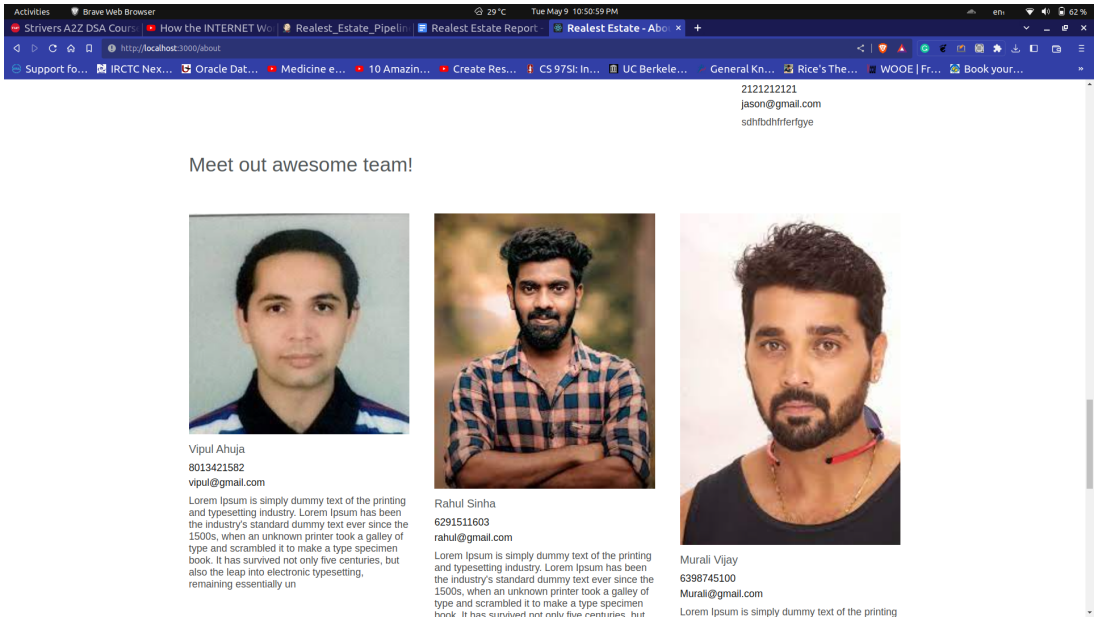
Home Page



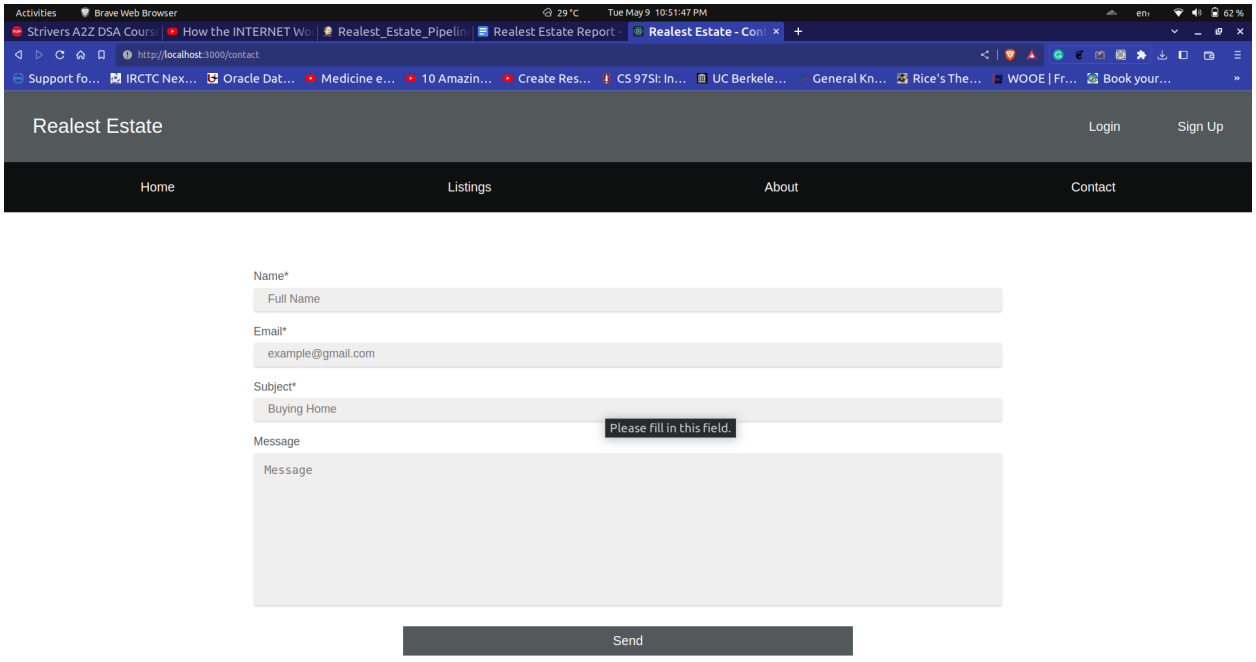
Listings Page



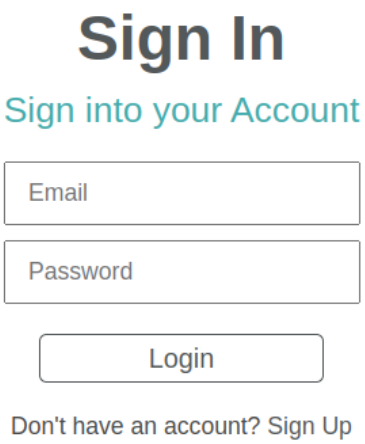
About Page



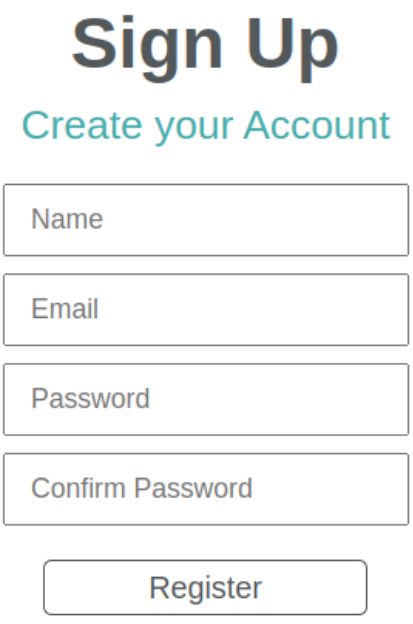
Contact Page



Login Page



SignUp



Backend:

In **Django**, the **code flow** is:

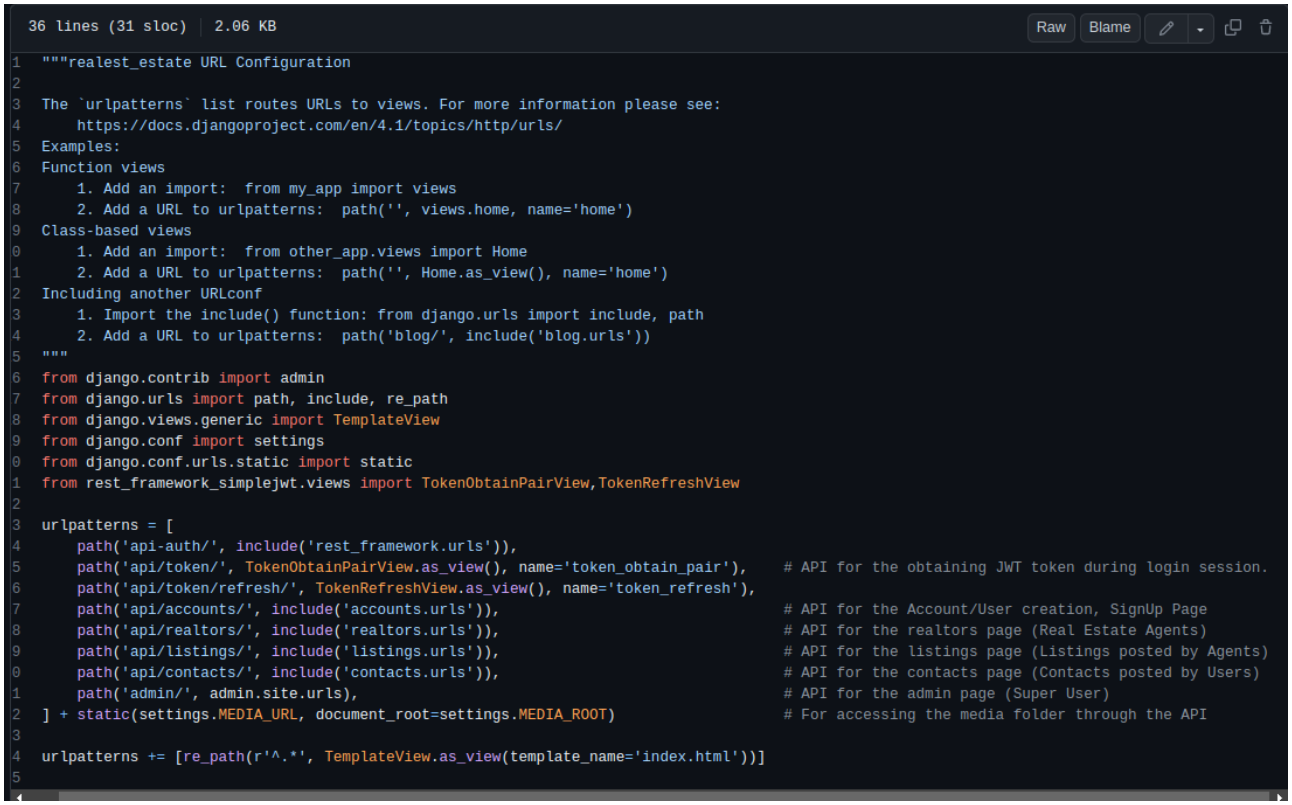
- We first create our app that will handle a particular functionality and add it to the settings.py file.
- We define our Model/Entity which will eventually form the table in our database.
- We register the given model with the Django Admin panel.
- We create the URL which will serve as the API Endpoints.
- We create our SERIALIZER file that will help us in doing the serialization and deserialization for the request and response coming and going to the client side.
- We write our business logic inside the VIEWs file.

Apps in our Project:

- **Accounts**: Stores the details of the Users visiting our website, also handles the login/logout and the signup features. JWT is used here to provide maximum security to the routes.
- **Contacts**: Handles the people who have queries regarding the website and can email us the detail of their query. Google API is used here to send emails.
- **Listings**: The houses along with their details and the photos uploaded by the Agents are handled here.
- **Realtors**: Handles the details of the real estate agents.

URLs defined in our Project:

- **Default URL:**

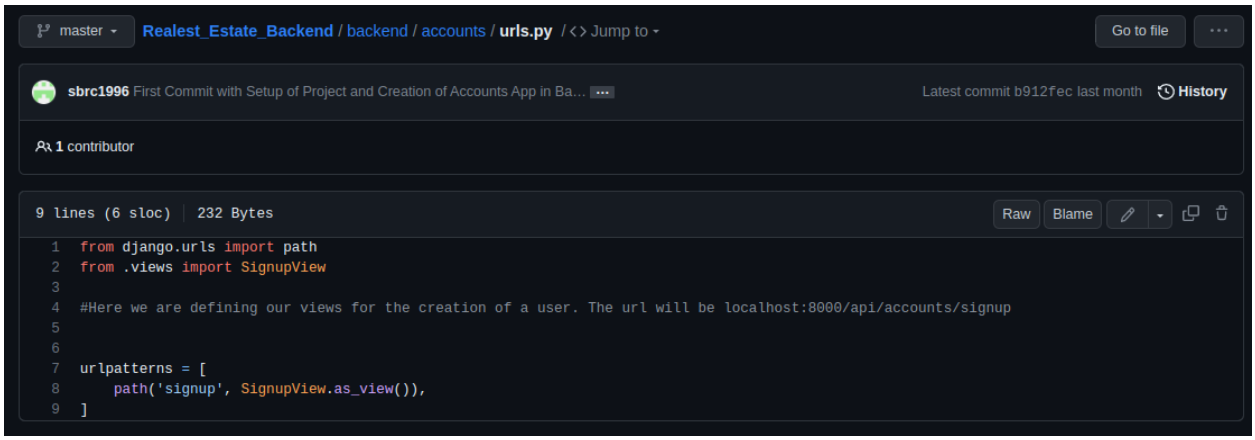


```
36 lines (31 sloc)  2.06 KB                                                    Raw  Blame
"""realest_estate URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/4.1/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path, include, re_path
from django.views.generic import TemplateView
from django.conf import settings
from django.conf.urls.static import static
from rest_framework_simplejwt.views import TokenObtainPairView,TokenRefreshView

urlpatterns = [
    path('api-auth/', include('rest_framework.urls')),
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),    # API for the obtaining JWT token during login session.
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/accounts/', include('accounts.urls')),                                # API for the Account/User creation, SignUp Page
    path('api/realtors/', include('realtors.urls')),                                # API for the realtors page (Real Estate Agents)
    path('api/listings/', include('listings.urls')),                                # API for the listings page (Listings posted by Agents)
    path('api/contacts/', include('contacts.urls')),                                # API for the contacts page (Contacts posted by Users)
    path('admin/', admin.site.urls),                                                # API for the admin page (Super User)
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)                   # For accessing the media folder through the API

urlpatterns += [re_path(r'^.*', TemplateView.as_view(template_name='index.html'))]
```

This is the URL configuration file for a Django project named "realest_estate". It defines the URLs for the APIs and admin pages of the project, as well as a catch-all route for serving a single-page application.The above code defines the URL patterns for the API endpoints of the project. It starts by importing necessary modules, including the admin module for the Django admin site, the path, include, and re_path modules for defining URL patterns, and the TemplateView module for serving templates. It also imports the TokenObtainPairView and TokenRefreshView classes from the rest_framework_simplejwt.views module for handling authentication with JSON Web Tokens (JWTs).The urlpatterns variable is a list of URL patterns for the project. The first seven URL patterns define the API endpoints for the project. Each path maps to a different API endpoint. The include function is used to include the URLs from the respective app, which are defined in separate URL configuration files.

For example, the api/accounts/ path maps to the URLs defined in the accounts.urls file.The path('admin/', admin.site.urls) URL pattern maps to the Django admin site, which provides an interface for managing the project's data and settings.
The + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT) code is used to serve static media files during development. It adds a URL pattern that maps to the project's MEDIA_ROOT directory, which is defined in the project's settings file. This allows the API to serve media files, such as images or videos, during development.
This code defines a catch-all URL pattern that serves a single-page application. The re_path function is used to define a regular expression pattern that matches any URL. The TemplateView.as_view() function is used to create a view that serves the index.html template, which is the entry point for the single-page application. By defining a catch-all URL pattern, any request that doesn't match one of the other URL patterns will be handled by the single-page application.

- **Accounts URL:**



This is a Django URL configuration file that defines a single URL pattern for the creation of a user. The pattern maps the signup URL path to the SignupView view.
from django.urls import path
from .views import SignupView

This code imports the path function from the django.urls module and the SignupView view from the views.py file in the current directory.
urlpatterns = [path('signup', SignupView.as_view()),]
This code defines the URL pattern for the signup endpoint. The path function takes two arguments: the URL path as a string, and the view that should handle requests to that path.
In this case, the URL pattern is simply signup, which means that the endpoint will be accessed at localhost:8000/api/accounts/signup. The SignupView.as_view() method is used to create an instance of the SignupView class, which is responsible for handling requests to this endpoint.
The SignupView view should handle the logic for creating a new user account. This typically involves validating user input, creating a new User object, and storing it in the database. The exact implementation of this view will depend on the specific requirements of the project.

- **Contacts URL:**

This is a Django URL configuration file that defines a single URL pattern for creating a new contact. The pattern maps the root URL path to the ContactCreateView view.This code imports the path function from the django.urls module and the ContactCreateView view from the views.py file in the current directory.
urlpatterns = [path('', ContactCreateView.as_view()),]
This code defines the URL pattern for the root endpoint. The path function takes two arguments: the URL path as a string, and the view that should handle requests to that path.

In this case, the URL pattern is an empty string '', which means that the endpoint will be accessed at localhost:8000/api/contacts/. The ContactCreateView.as_view() method is used to create an instance of the ContactCreateView class, which is responsible for handling requests to this endpoint.

The ContactCreateView view should handle the logic for creating a new contact object. This typically involves validating user input, creating a new Contact object, and storing it in the database. The exact implementation of this view will depend on the specific requirements of the project.

- **Listings URL:**

This is a Django URL configuration file that defines the URL patterns for the listings app.
from django.urls import path
from .views import ListingsView, ListingView, SearchView
This code imports the path function from the django.urls module and the three views (ListingsView, ListingView, and SearchView) from the views.py file in the current directory.

```
urlpatterns = [
    path('', ListingsView.as_view(),name="ListALL"),
    path('search', SearchView.as_view()),
    path('<slug>', ListingView.as_view()),
]
```

This code defines the URL patterns for the listings appThe first URL pattern is for the root endpoint ''. It maps to the ListingsView view and specifies the name ListALL for this URL pattern. This means that this endpoint can be accessed using localhost:8000/api/listings/.
The second URL pattern is for the search endpoint. It maps to the SearchView view and can be accessed using localhost:8000/api/listings/search.
The third URL pattern is a dynamic URL pattern that includes a slug parameter. It maps to the ListingView view and can be accessed using localhost:8000/api/listings/<slug>, where <slug> is a dynamic value. This URL pattern is used for accessing a particular listing, not by primary key (id), but by the slug field.
All three URL patterns use the as_view() method to create an instance of the respective view classes. The name parameter is used to specify a name for the URL pattern, which can be used in reverse URL resolution (i.e., generating URLs based on the name of the pattern).

- **Realtors URL:**

The given code defines the URL patterns for the realtors app. It imports the required views from the views.py module.The first URL pattern '' is associated with the RealtorListView view, which will return a list of all the realtors. The second URL pattern topseller is associated with the TopSellerView view, which will return a list of all the realtors whose topseller field is set to True. The third URL pattern '<pk>' is associated with the

RealtorView view, which will return the details of a single realtor with the given primary key.

Here's a breakdown of each of the URL patterns:

'': This matches the base URL for the realtors app (/api/realtors/), and is associated with the RealtorListView view.

'topseller': This matches the URL /api/realtors/topseller/, and is associated with the TopSellerView view.

'<pk>': This matches any URL that has an integer after /api/realtors/. The integer is captured as a named parameter pk and is passed to the RealtorView view to retrieve the details of the realtor with that primary key.

Database: MYSQL:

```
mysql> use realest_estate;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------------------------------+
| Tables_in_realest_estate               |
+----------------------------------------+
| accounts_useraccount                   |
| accounts_useraccount_groups            |
| accounts_useraccount_user_permissions  |
| auth_group                             |
| auth_group_permissions                 |
| auth_permission                        |
| contacts_contact                       |
| django_admin_log                       |
| django_content_type                    |
| django_migrations                      |
| django_session                         |
| listings_listing                       |
| realtors_realtor                       |
+----------------------------------------+
13 rows in set (0.00 sec)

mysql>
```

The tables above are the ones present in our database realest_estate.

Settings.py (Configurations):

```
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = ['*']


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'corsheaders',
    'rest_framework',
    'accounts',
    'realtors',
    'listings',
    'contacts',
]
```

Debug= True helps in debugging the code.
ALLOWED_HOSTS = ['*'] means code can run in any host.

```

# Database
# https://docs.djangoproject.com/en/4.1/ref/settings/#databases

DATABASES = {
    # 'default': {
    #     'ENGINE': 'django.db.backends.mysql',
    #     'NAME': 'realest_estate',
    #     'HOST': 'mydb',               # '127.0.0.1', Change this to the
    #     'PORT': '3306',
    #     'USER': 'root',
    #     'PASSWORD': 'subham123',
    # }
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': os.environ.get('DB_NAME'),
        'HOST': os.environ.get('DB_HOST'),
        'PORT': os.environ.get('DB_PORT'),
        'USER': os.environ.get('DB_USER'),
        'PASSWORD': os.environ.get('DB_PASSWORD'),
    }
}
```

The settings for the Database connection of the backend code with the MYSQL database. Initially, we had directly given the information in the Dictionary to do the connection but later we changed and provided the

information through environment variables so that it can run in any environment.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_HOST_USER = 'testusername190@gmail.com'
EMAIL_HOST_PASSWORD = 'nkzqsqaqfgivtrhg'
EMAIL_USE_TLS = True


# Password validation
# https://docs.djangoproject.com/en/4.1/ref/settings/#auth-password-valida
```

Email configuration using Google API and SMTP Port 587 with its API key.

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated'
    ],
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication'
    ],
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 3
}

CORS_ORIGIN_ALLOW_ALL = True

FILE_UPLOAD_PERMISSIONS=0o640

AUTH_USER_MODEL = 'accounts.UserAccount'
```

Library addition to including the JWT Refresh and Access Tokens and adding CORS-Header to interact with our React Frontend.

Testing:

For testing, we have utilized the Django inbuild testing library **django.test**

```python
from django.test import TestCase, Client
from django.urls import reverse
from .models import Realtor


# Create your tests here.

class TestRealtorsModels(TestCase):
    def setUp(self):
        Realtor.objects.create(
            name='Test Realtor',
            phone='555-555-5555',
            email='testrealtor@example.com',
            top_seller=True,
            )
    def test_realtor_model(self):
        realtor = Realtor.objects.get(name='Test Realtor')
        self.assertEqual(realtor.phone, '555-555-5555')
        self.assertEqual(realtor.email, 'testrealtor@example.com')
        self.assertTrue(realtor.top_seller)


# python3 manage.py test realtors.tests
```

Testing File for our Realtors App.

```
subham@subham-HP-Laptop-15s-fq2xxx:~/Desktop/RealEstate/backend$ python3 manage.py test realtors.tests
Found 1 test(s).
Creating test database for alias 'default'...
INFO 2023-05-08 11:43:24,732 django views views.py 20 SignupView
 Inside the SignupView API of accounts APP. Here we are creating an account of a user. JWT token not needed.
INFO 2023-05-08 11:43:24,733 django views views.py 16 RealtorListView
 Inside the RealtorListView API of realtor App. Here we are getting all the realtors from DB..
INFO 2023-05-08 11:43:24,733 django views views.py 24 RealtorView
 Inside the RealtorView API of realtor App. Here we are a particular realtors from DB based on its PK..
INFO 2023-05-08 11:43:24,733 django views views.py 30 TopSellerView
 Inside the TopSellerView API of realtor App. Here we are getting all the realtors who are topsellers..
INFO 2023-05-08 11:43:24,734 django views views.py 17 ListingsView
 Inside the ListingsView API of listings App. Here we are getting all the data filtered by list date and ispublished value..
INFO 2023-05-08 11:43:24,734 django views views.py 27 ListingView
 inside the ListingView API of listings App. Here we are getting the info of a particular View using id and JWT token is madatory here..
INFO 2023-05-08 11:43:24,735 django views views.py 35 SearchView
 Inside the SearchView API of listings App. Here we are getting the search field values from the frontend Form and using it to retrieve data from the backend. This is done in the home page itself.
System check identified some issues:

WARNINGS:
?: (staticfiles.W004) The directory '/home/subham/Desktop/RealEstate/backend/build/static' in the STATICFILES_DIRS setting does not exist.

System check identified 1 issue (0 silenced).
/home/subham/.local/lib/python3.10/site-packages/django/db/models/fields/__init__.py:1564: RuntimeWarning: DateTimeField Realtor.date_hired received a naive datetime (2023-05-08 11:43:24.769016) while time zone
support is active.
  warnings.warn(
.
----------------------------------------------------------------------
Ran 1 test in 0.006s

OK
Destroying test database for alias 'default'...
subham@subham-HP-Laptop-15s-fq2xxx:~/Desktop/RealEstate/backend$
```

The output of Testing of Realtors App.

45 lines (38 sloc) | 1.49 KB

```python
from django.test import TestCase, Client
from django.urls import reverse
from .models import Listing
from realtors.models import Realtor

# Create your tests here.

class TestListings(TestCase):
    def setUp(self):
        self.realtor = Realtor.objects.create(
            name='Test Realtor',
            phone='555-555-5555',
            email='testrealtor@example.com',
            top_seller=True,
        )

        Listing.objects.create(
            title = 'Mera Ghar',
            address = '433, Jadav Ghosh Road',
            city = 'Kolkata',
            state = 'West Bengal',
            description = 'A 3 storey house in Behala Sarsuna Kolkata about 3 km from metro station',
            sale_type = 'For Sale',
            price = 10000000,
            bedrooms = 5,
            bathrooms = 5.0,
            home_type = 'House',
            sqft  = 1000,
            realtor = self.realtor
            )

    def test_listings_model(self):
        listing = Listing.objects.get(title='Mera Ghar')
        self.assertEqual(listing.address, '433, Jadav Ghosh Road')
        self.assertEqual(listing.city, 'Kolkata')
        self.assertEqual(listing.state, 'West Bengal')
        self.assertNotEqual(listing.description, '433, Jadav Ghosh Road')
        self.assertEqual(listing.price, 10000000)
        self.assertEqual(listing.bedrooms, 5)
        self.assertEqual(listing.sqft, 1000)
        self.assertEqual(listing.address, '433, Jadav Ghosh Road')



# python3 manage.py test listings.tests
```

Testing File for Listings App

```
subham@subham-HP-Laptop-15s-fq2xxx:~/Desktop/RealEstate/backend$ python3 manage.py test listings.tests
Found 1 test(s).
Creating test database for alias 'default'...
INFO 2023-05-08 11:44:42,752 django views views.py 20 SignupView
 Inside the SignupView API of accounts APP. Here we are creating an account of a user. JWT token not needed.
INFO 2023-05-08 11:44:42,753 django views views.py 16 RealtorListView
 Inside the RealtorListView API of realtor App. Here we are getting all the realtors from DB..
INFO 2023-05-08 11:44:42,753 django views views.py 24 RealtorView
 Inside the RealtorView API of realtor App. Here we are a particular realtors from DB based on its PK..
INFO 2023-05-08 11:44:42,753 django views views.py 30 TopSellerView
 Inside the TopSellerView API of realtor App. Here we are getting all the realtors who are topsellers..
INFO 2023-05-08 11:44:42,754 django views views.py 17 ListingsView
 Inside the ListingsView API of listings App. Here we are getting all the data filtered by list date and ispublished value..
INFO 2023-05-08 11:44:42,754 django views views.py 27 ListingView
 inside the ListingView API of listings App. Here we are getting the info of a particular View using id and JWT token is madatory here..
INFO 2023-05-08 11:44:42,754 django views views.py 35 SearchView
 Inside the SearchView API of listings App. Here we are getting the search field values from the frontend Form and using it to retrieve data from the backend. This is done in the home page itself.
System check identified some issues:

WARNINGS:
?: (staticfiles.W004) The directory '/home/subham/Desktop/RealEstate/backend/build/static' in the STATICFILES_DIRS setting does not exist.

System check identified 1 issue (0 silenced).
/home/subham/.local/lib/python3.10/site-packages/django/db/models/fields/__init__.py:1564: RuntimeWarning: DateTimeField Realtor.date_hired received a naive datetime (2023-05-08 11:44:42.778686) while time zone support is active.
  warnings.warn(
.
----------------------------------------------------------------------
Ran 1 test in 0.008s

OK
Destroying test database for alias 'default'...
subham@subham-HP-Laptop-15s-fq2xxx:~/Desktop/RealEstate/backend$
```

The output of Testing of Listings App.

```python
from django.test import TestCase, Client
from django.urls import reverse
from .models import Contact

# Create your tests here.

class TestContactsModel(TestCase):
    def setUp(self):
        Contact.objects.create(
            name= 'Test User',
            email= 'testuser@example.com',
            subject =  'This is a test query',
            message = 'Yo Yo!! Whats Up Dude??'
        )
    def test_Contacts_model(self):
        contact = Contact.objects.get(name= 'Test User')
        self.assertEqual(contact.email , 'testuser@example.com')
        self.assertNotEqual(contact.subject, 'dskjfbdsf')
        self.assertEqual(contact.message , 'Yo Yo!! Whats Up Dude??')

#command to exeute the test script : python3 manage.py test contacts.tests
```

Testing File for Contacts App.

```
subham@subham-HP-Laptop-15s-fq2xxx:~/Desktop/RealEstate/backend$ python3 manage.py test contacts.tests
Found 1 test(s).
Creating test database for alias 'default'...
INFO 2023-05-08 11:45:35,167 django views views.py 20 SignupView
 Inside the SignupView API of accounts APP. Here we are creating an account of a user. JWT token not needed.
INFO 2023-05-08 11:45:35,168 django views views.py 16 RealtorListView
 Inside the RealtorListView API of realtor App. Here we are getting all the realtors from DB..
INFO 2023-05-08 11:45:35,168 django views views.py 24 RealtorView
 Inside the RealtorView API of realtor App. Here we are a particular realtors from DB based on its PK..
INFO 2023-05-08 11:45:35,168 django views views.py 30 TopSellerView
 Inside the TopSellerView API of realtor App. Here we are getting all the realtors who are topsellers..
INFO 2023-05-08 11:45:35,169 django views views.py 17 ListingsView
 Inside the ListingsView API of listings App. Here we are getting all the data filtered by list date and ispublished value..
INFO 2023-05-08 11:45:35,169 django views views.py 27 ListingView
 inside the ListingView API of listings App. Here we are getting the info of a particular View using id and JWT token is madatory here..
INFO 2023-05-08 11:45:35,169 django views views.py 35 SearchView
 Inside the SearchView API of listings App. Here we are getting the search field values from the frontend Form and using it to retrieve data from the backend. This is done in the home page itself.
System check identified some issues:

WARNINGS:
?: (staticfiles.W004) The directory '/home/subham/Desktop/RealEstate/backend/build/static' in the STATICFILES_DIRS setting does not exist.

System check identified 1 issue (0 silenced).
/home/subham/.local/lib/python3.10/site-packages/django/db/models/fields/__init__.py:1564: RuntimeWarning: DateTimeField Contact.contact_date received a naive datetime (2023-05-08 11:45:35.195182) while time zon
e support is active.
  warnings.warn(
.
----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
Destroying test database for alias 'default'...
subham@subham-HP-Laptop-15s-fq2xxx:~/Desktop/RealEstate/backend$
```

The output of Testing of Contacts App.

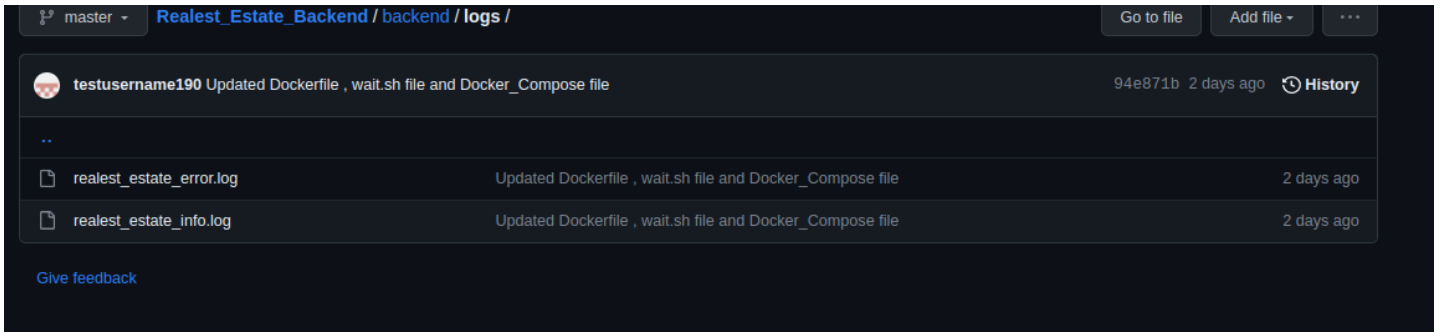
Logging:

Configurations in settings.py for our Logger:

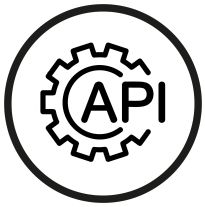
```python
LOGGERS = (
    {
        "django": {
            "handlers": ["console_handler", "info_handler"],
            "level": "INFO",
        },
        "django.request": {
            "handlers": ["error_handler"],
            "level": "INFO",
            "propagate": True,
        },
        "django.template": {
            "handlers": ["error_handler"],
            "level": "DEBUG",
            "propagate": True,
        },
        "django.server": {
            "handlers": ["error_handler"],
            "level": "INFO",
            "propagate": True,
        },
    },
)


LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": FORMATTERS[0],
    "handlers": HANDLERS,
    "loggers": LOGGERS[0],
}
```

For logging, we have used Python's inbuild logging library **logging** and we are doing persistent storage of log file through our docker-compose file using Volumes.

The output of Loggin File:



API Table:



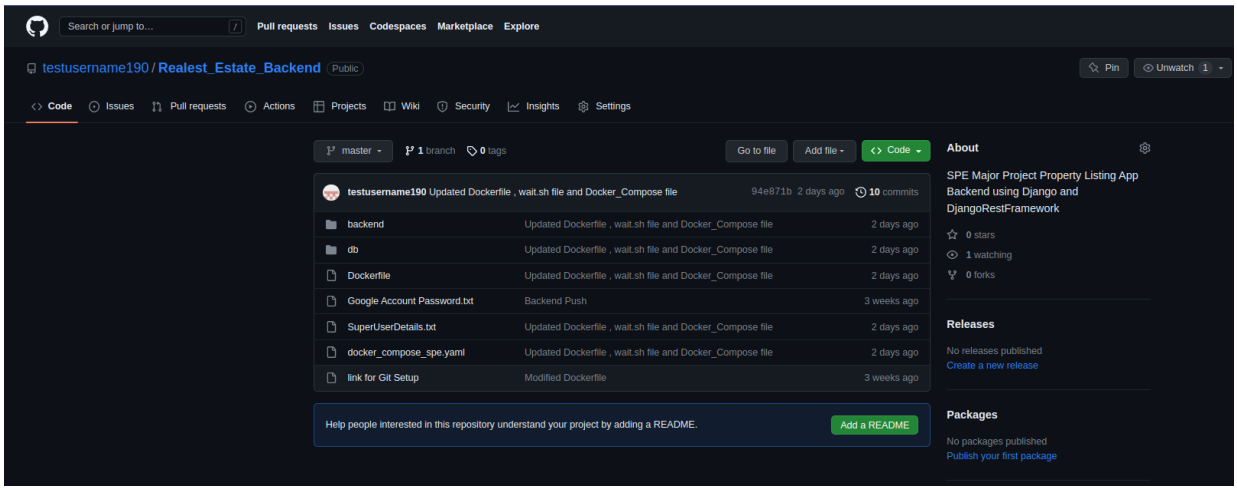
API Endpoint	*Request Type*	*Details of the API*
http://127.0.0.1:8000/api/accounts/signup	POST	Creation of a new User Account
http://127.0.0.1:8000/api/token/	POST	Validating the credentials of the existing user and generating a JWT token on successful verification
localhost:8000/api/realtors/	GET	Get all the realtors
localhost:8000/api/realtors/1	GET	Get a particular realtor by id/pk
localhost:8000/api/realtors/topseller	GET	Get only the top-selling realtors
http://127.0.0.1:8000/api/listings/	GET	Get all the listings
http://127.0.0.1:8000/api/listings/Shyama-Prasad-Mukherjee-Rd	GET	Get a particular listing by its slug field
http://127.0.0.1:8000/api/listings/search	POST	Get the data from the client used for searching the listings data
http://127.0.0.1:8000/api/contacts/	POST	Send a request email with the client email id and query

DEVOPS:

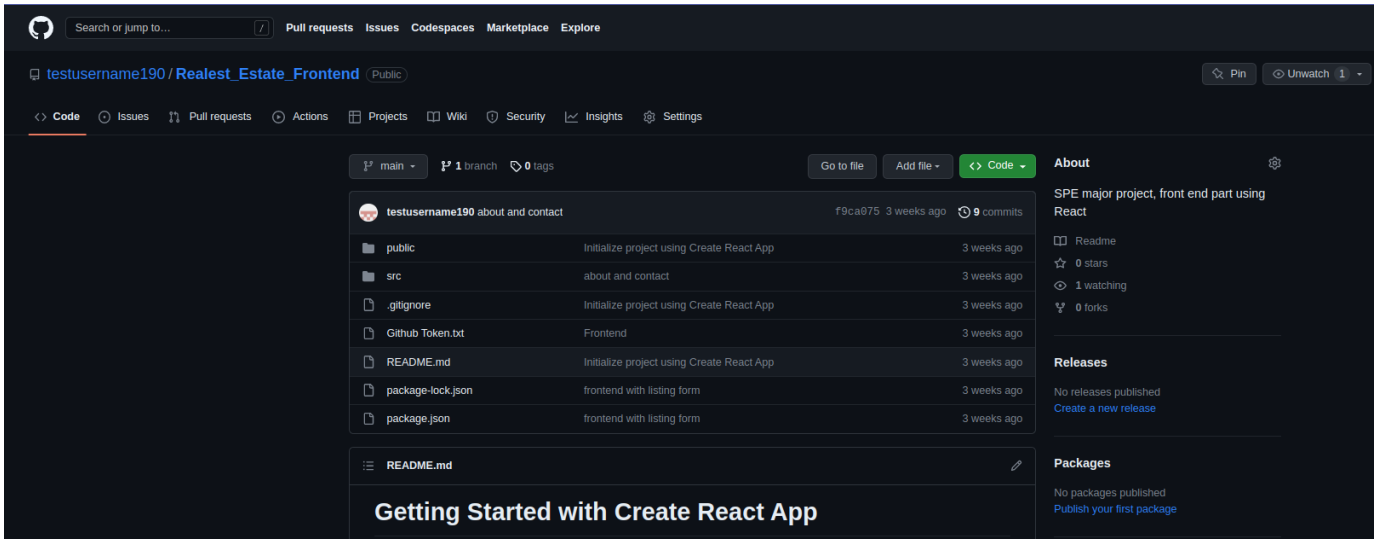
Version Controlling System:

We have used Github as VCS. GitHub is a website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code.

Git hub repo for Backend Code:



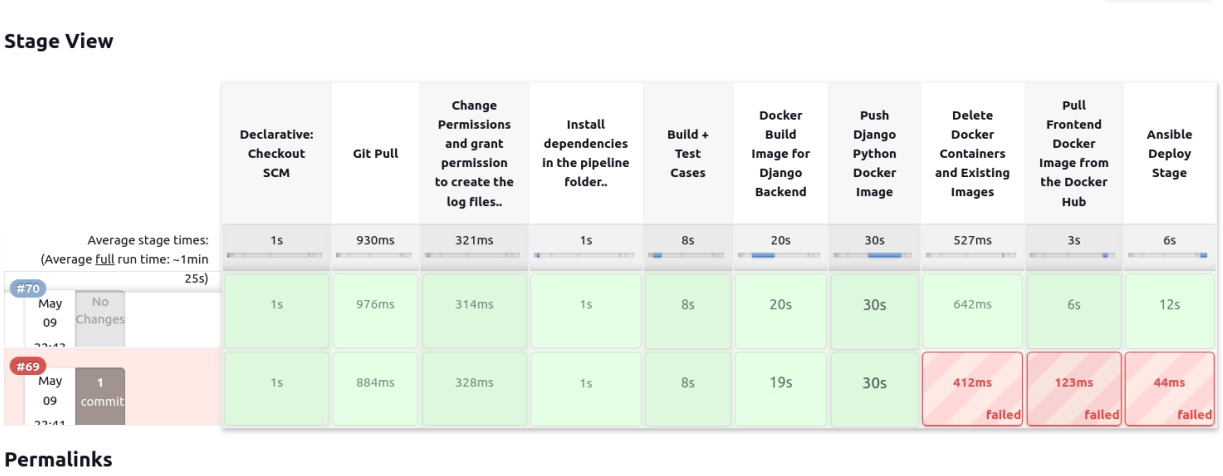
Git hub repo for Frontend Code:



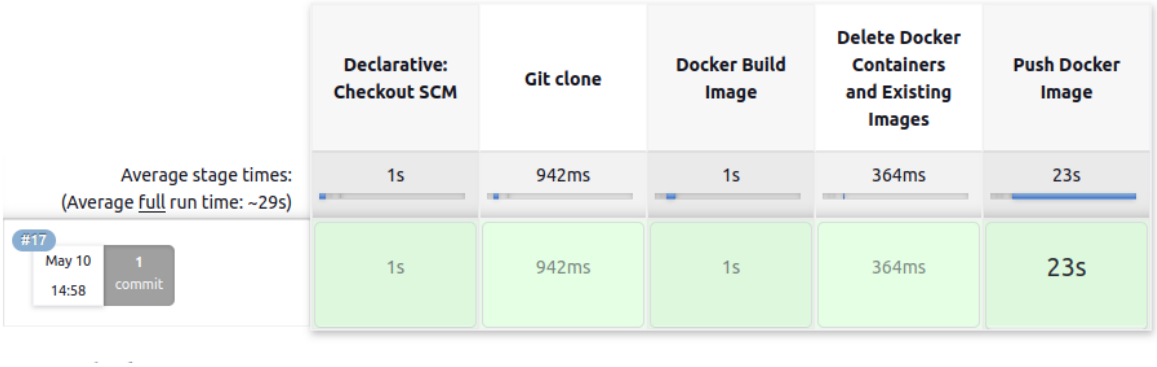
CI/CD Pipeline using Jenkins:

A CI/CD pipeline is a series of steps that must be performed in order to deliver a new version of the software. We have used Jenkins for our pipeline. Jenkins is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production

Pipeline run for Backend:



Pipeline run for Frontend:



JenkinsFile for Frontend:

This Jenkinsfile for frontend is pipeline scripts that will automate the build and will create the docker image for the front-end and will push the image file to the Docker hub.
Below is the script for the Jenkinsfile for automation:

```
pipeline {
     environment{
        dockerimage=""
    }
    agent any
    stages {
        stage('Git clone') {
            steps {
              git 'https://github.com/testusername190/Realest_Estate_Frontend.git'
            }
        }
```

Git clone:Clones the source code from a GitHub repository for the front-end.

```
    stage('Docker Build Image') {
        steps {
            script{
                dockerimage=docker.build "vishalsin25/spe-major-front:latest"
            }
        }
    }
```

Docker Build Image: Builds the latest Docker image for the front-end development code.

```
        stage('Delete Docker Containers and Existing Images') {
    steps {
        script{
            // here we are checking if there are any containers running in our system if so then delete them.
            // def running_containers = sh (returnStdout: true, script: 'docker ps -q').trim()
            // if (running_containers) {
            //     sh 'docker rm -f $(docker ps -aq)'
            // }

            // sh 'docker image rm -f sbrc1996/speminiproject'
            sh '''
                # Remove all images with the tag <none>
                docker rmi --force $(docker images | grep "<none>" | awk '{print $3}')
                '''
        }
    }
}
```

Delete Existing Image: Clear the previous/old existing images so that only the latest image can be pushed to the dockerhub.

```
    stage('Push Docker Image') {
        steps {
            script{
                docker.withRegistry('','dockerhub'){
                dockerimage.push()
                }
            }
        }
    }
```

Push Docker Image: Pushing the docker image to the docker hub which will be further pulled and then can be used with backend and database containers.

JenkinsFile for Backend and Database and Docker Compose:

This Jenkinsfile is a pipeline script that automates the build, testing, and deployment of a Django backend application using Docker. Here's a stage-wise explanation of the code:

```
pipeline {
    environment{
        dockerimg = ''
        PYTHONPATH = "/home/subham/.local/lib/python3.10/site-packages:/var/lib/jenkins/workspace/Realest_Estate_Pipeline_Django/backend"
        DB_NAME = "realest_estate"
        DB_HOST = "127.0.0.1"
        DB_PORT = "3306"
        DB_USER = "root"
        DB_PASSWORD = "subham123"
    }
    agent any

```

- environment: Sets up environment variables for the pipeline, including the Docker image name, PYTHONPATH, and database credentials. As Jenkins is unable to fetch the system env variables.
- agent any: Specifies that the pipeline can run on any available agent.

```
stages {
    stage('Git Pull') {
        steps {
            git 'https://github.com/testusername190/Realest_Estate_Backend.git'
        }
    }
    stage('Change Permissions and grant permission to create the log files..') {
        steps {
            sh 'chmod -R 777 backend/logs'
        }
    }
    stage('Install dependencies in the pipeline folder..') {
        steps {
            sh 'pip install django djangorestframework django-cors-headers djangorestframework-simplejwt Pillow mysqlclient'
        }
    }
    stage('Build + Test Cases') {
        steps {
            echo 'Build not required for our django project, we just need to install pip in our Docker file'
            echo 'Test cases written for Model Testing in Django!!'
            sh 'cd backend/ && python3 manage.py test contacts.tests'
            sh 'cd backend/ && python3 manage.py test listings.tests'
            sh 'cd backend/ && python3 manage.py test realtors.tests'
        }
    }
```

- stages: Divides the pipeline into different stages to perform specific tasks in sequence.
- Git Pull: Clones the source code from a GitHub repository.
- Change Permissions and grant permission to create the log files..: Changes the permissions of the log files to allow write access.
- Install dependencies in the pipeline folder..: Installs the required dependencies in the pipeline folder.
- Build + Test Cases: Runs the test cases for the Django models.

```
stage('Docker Build Image for Django Backend') {
    steps {
        script{
            // sh 'cd backend/'
            dockerimg=docker.build("sbrc1996/realest_estate:latest")
        }
    }
}
stage('Push Django Python Docker Image') {
    steps {
        script{
            docker.withRegistry('','dockerhub'){
            dockerimg.push()
            }
        }
    }
}
stage('Delete Docker Image') {
    steps {
        script{
            // sh 'docker image rm -f sbrc1996/speminiproject'
            sh '''
                # Remove all images with the tag <none>
                docker rmi --force $(docker images | grep "<none>" | awk '{print $3}')
                '''
        }
    }
}
```

- Docker Build Image for Django Backend: Builds a Docker image for the Django backend.
- Push Django Python Docker Image: Pushes the Docker image to a Docker registry (in this case, Docker Hub).
- Delete Docker Image: Removes any dangling Docker images with the tag <none>

```
stage('Pull Frontend Docker Image from the Docker Hub')
{
    steps
    {
        sh 'docker pull vishalsin25/spe-major-front'
    }
}
```

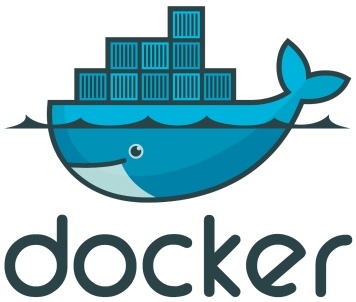
Pulling the frontend Image from the Dockerhub of Vishal that contains the image of node used to run our frontend react application.

```
stage('Ansible Deploy Stage') {
            steps {
                //Ansible Deploy to remote server
                ansiblePlaybook colorized: true,
                                disableHostKeyChecking: true,
                                inventory: 'inventory',
                                installation:'Ansible',
                                playbook: 'playbook.yml',
                                sudoUser: null
            }
        }
```

- The stage is called "Ansible Deploy Stage" and is defined using the stage block in the Jenkins pipeline syntax.
- The steps block defines the steps that should be executed as a part of this stage.
- The ansiblePlaybook step is used to execute an Ansible playbook on the remote server. The arguments passed to this step are:
- colorized: true - enables colorized output for the playbook execution.
- disableHostKeyChecking: true - disables the host key checking for SSH connections.
- inventory: 'inventory' - specifies the inventory file containing the list of hosts to which the playbook will be executed.
- installation:'Ansible' - specifies the installation directory for Ansible on the remote server.
- playbook: 'playbook.yml' - specifies the name of the Ansible playbook that should be executed on the remote server.
- sudoUser: null - specifies that the playbook should be executed as the root user on the remote server.

Overall, this Jenkinsfile sets up a continuous integration and continuous deployment (CI/CD) pipeline for a Django backend application, automating the build, testing, and deployment processes.

Containerization:



Docker is a set of platform-as-a-service products that use OS-level virtualization to deliver software in packages called containers. The service has both free and premium tiers. The software that hosts the containers is called Docker Engine. It was first started in 2013 and is developed by Docker.

Dockerfile for Backend:

```
# Creating the Base Image
FROM python:3.10.6-alpine3.15

# setup environment variable
ENV DockerHOME=/app

# set work directory
RUN mkdir -p $DockerHOME

# where your code lives
WORKDIR $DockerHOME

# set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1


# install dependencies mysqlclient using mariadb connector and delete it after installation.
RUN apk update && \
    apk add --no-cache mariadb-connector-c-dev && \
    apk add --no-cache python3 python3-dev mariadb-dev build-base && \
    pip3 install --no-cache-dir mysqlclient && \
    apk del python3-dev mariadb-dev build-base

# checking the network adapters inside the docker container
RUN apk add netcat-openbsd

# copy whole project to your docker app directory.
COPY ./backend $DockerHOME

# run this command to install all dependencies
RUN pip3 install --no-cache-dir django djangorestframework django-cors-headers djangorestframework-simplejwt Pillow

# we create a new wait.sh file who makes our django app wait for the db to startup.

COPY ./backend/wait.sh $DockerHOME/wait.sh
RUN chmod +x $DockerHOME/wait.sh


```

Give feedback

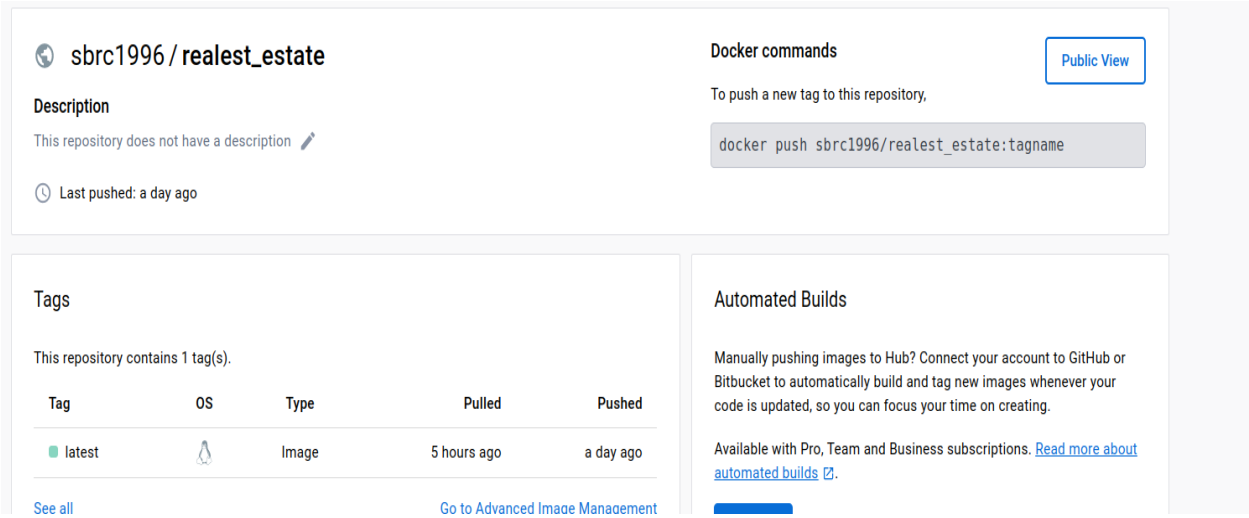
1. The Dockerfile is used to create a Docker image of the Django project.
2. FROM python:3.10.6-alpine3.15: This line specifies the base image to be used, in this case, Python 3.10.6 with Alpine Linux version 3.15. Alpine Linux is a lightweight distribution that is popular in Docker containers.
3. ENV DockerHOME=/app: This line sets the environment variable DockerHOME to "/app". This will be the root directory for our Docker container.
4. RUN mkdir -p $DockerHOME: This line creates a directory named DockerHOME, with the -p flag used to create the parent directories if they do not exist.
5. WORKDIR $DockerHOME: This line sets the working directory for our container to DockerHOME.
6. ENV PYTHONDONTWRITEBYTECODE 1: This line sets the environment variable PYTHONDONTWRITEBYTECODE to 1, which prevents Python from writing .pyc files.
7. ENV PYTHONUNBUFFERED 1: This line sets the environment variable PYTHONUNBUFFERED to 1, which ensures that Python output is not buffered.
8. RUN apk update && \ apk add --no-cache mariadb-connector-c-dev && \ apk add --no-cache python3 python3-dev mariadb-dev build-base && \ pip3 install --no-cache-dir mysqlclient && \ apk del python3-dev mariadb-dev build-base: This section installs the dependencies required for our Django project, including MariaDB connector, Python 3, and the development libraries for MariaDB and Python. After the dependencies are installed, it removes the packages that are no longer needed.
9. RUN apk add netcat-openbsd: This line installs netcat-openbsd, which will be used to check network adapters inside the Docker container.
10. COPY ./backend $DockerHOME: This line copies the entire backend directory to the Docker container's app directory.
11. RUN pip3 install --no-cache-dir django djangorestframework django-cors-headers djangorestframework-simplejwt Pillow: This line installs the required Python packages using pip.
12. COPY ./backend/wait.sh $DockerHOME/wait.sh: This line copies the wait.sh script to the Docker container's app directory.
13. RUN chmod +x $DockerHOME/wait.sh: This line makes the wait.sh script executable.
14. Finally, once the Dockerfile is built, we can use this image to create a Docker container that can run our Django project.

Dockerfile for Frontend:

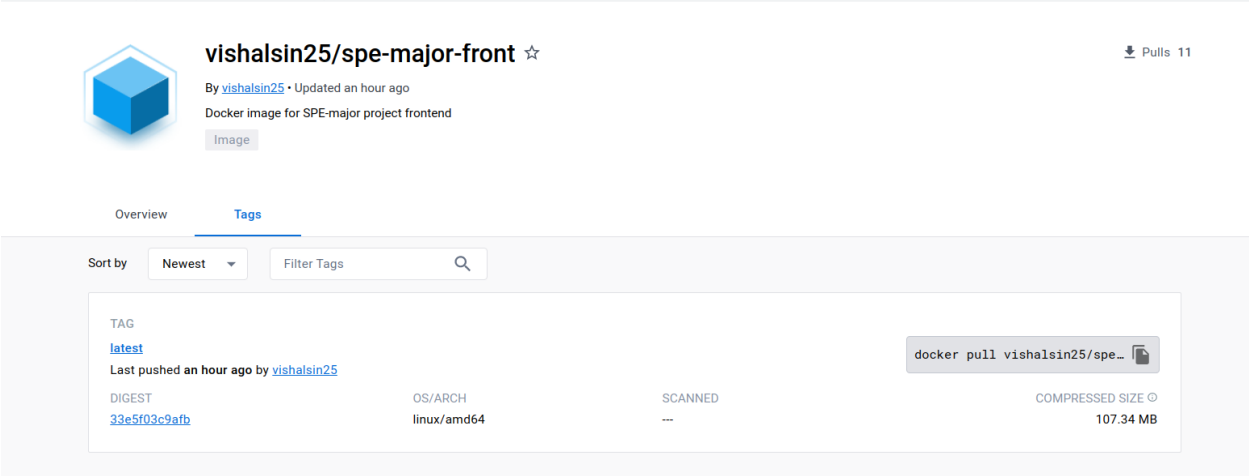
```
#Telling Docker to use the the node:17-alpine image as the base image
FROM node:12-alpine
# Layer 2: Telling Docker to create a directory called `/usr/src/app` in the container and set it as the working directory.
WORKDIR /frontend

# Layer 3: Copying the package.json file from the root of the project to the `app` directory in the container.
COPY ./package.json ./
COPY ./package-lock.json ./

# Layer 4: Installing the dependencies listed in the package.json file.
RUN npm i --force
# RUN NODE_ENV=development npm i --force

# Layer 5: Copying all the files from the root of the project to the `app` directory in the container.
COPY . .

# Layer 6: Telling Docker that the container will listen on port 3000.
EXPOSE 3000

# Layer 7: Telling Docker to run the `npm start` command when the container is started.
CMD [ "npm", "start" ]
```

1. FROM node:12-alpine: This instruction tells Docker to use the node:12-alpine image as the base image for our new image. This means that our new image will have all the features of the node:12-alpine image.
2. WORKDIR /frontend: This instruction tells Docker to create a new directory called frontend in the container and set it as the working directory.
3. COPY ./package.json ./ and COPY ./package-lock.json ./: These instructions copy the package.json and package-lock.json files from the root directory of the project into the frontend directory in the container.
4. RUN npm i --force: This instruction runs the npm i command in the frontend directory in the container, which installs all the dependencies listed in the package.json file.
5. COPY . .: This instruction copies all the files from the root directory of the project into the frontend directory in the container.
6. EXPOSE 3000: This instruction tells Docker that the container will listen on port 3000.
7. CMD ["npm", "start"]: This instruction tells Docker to run the npm start command when the container is started. This command will start the application on port 3000.

DockerHub Repo For Backend:



DockerHub For Frontend:



Docker-Compose File:

```yaml
version: "3"

services:
  realest_estate_backend:
    image: sbrc1996/realest_estate
    container_name: realest_estate_backend
    environment:
      - DB_HOST=172.16.128.119
      - DB_PORT=30000
      - DB_NAME=realest_estate
      - DB_USER=root
      - DB_PASSWORD=root
    ports:
      - "8000:8000"
    depends_on:
      - mydb
    entrypoint: ["./wait.sh"]
    volumes:
      - realest_estate_logs:/app/logs     # Persistent Log storing.
      - realest_estate_media:/app/media   # Persistent Media storage.

  mydb:
    image: mysql:8.0.21
    ports:
      - "30000:3306"
    environment:
      #These environment variables are to be not included in the Settings.py file of Django.
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: realest_estate
    # This will create the database realest_estate here if it not exists.
    volumes:
      - mydb_data:/var/lib/mysql

  realest_estate_frontend:
    image: vishalsin25/spe-major-front
    container_name: realest_estate_frontend
    stdin_open: true
    ports:
      - "3000:3000"
    depends_on:
      - realest_estate_backend


volumes:
  mydb_data:             # Creating a volume named mydb_data and store the data of the MYSQL running in the container by doing the mapping.
  realest_estate_logs:   # Creating a volume to store the logs persistently.
  realest_estate_media:  # Creating a volume to store the photos and other media files persistently.
```

wait.sh file:

15 lines (9 sloc) | 376 Bytes Raw Blame

```bash
#!/bin/sh

while ! nc -z mydb 3306 ; do
    echo "Waiting for the MySQL Server"
    sleep 3
done


python3 manage.py makemigrations
python3 manage.py migrate

DJANGO_SUPERUSER_USERNAME=adminbrc DJANGO_SUPERUSER_EMAIL=adminbrc@example.com DJANGO_SUPERUSER_PASSWORD=password123 python3 manage.py cr

python3 manage.py runserver 0.0.0.0:8000
```

Brief Explanation for the DockerCompose file and Wait.sh file:

1. version: "3": This instruction specifies the version of Docker Compose being used in this file.

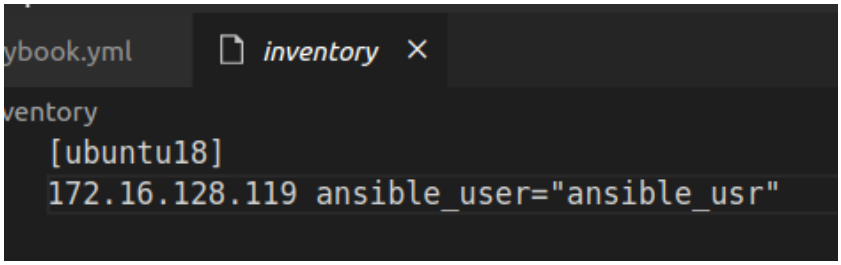
2. services: This is a list of the services (containers) that will be created by Docker Compose.
3. realest_estate_backend: This service is the backend container for the real estate application. It uses an existing Docker image from Docker Hub (sbrc1996/realest_estate), sets the container name to realest_estate_backend, and configures environment variables for the container's database connection settings (DB_HOST, DB_PORT, DB_NAME, DB_USER, and DB_PASSWORD). It also maps the container's port 8000 to the host's port 8000, sets a dependency on the mydb service, runs a script before starting the container (entrypoint: ["./wait.sh"]), and defines two volumes for the container's logs and media files.
4. mydb: This service is the database container for MySQL. It uses the official MySQL Docker image (mysql:8.0.21), maps the container's port 3306 to the host's port 30000, sets environment variables for the MySQL root password and database name (MYSQL_ROOT_PASSWORD and MYSQL_DATABASE), and defines a volume for the container's data.
5. realest_estate_frontend: This service is the frontend container for the real estate application. It uses an existing Docker image from Docker Hub (vishalsin25/spe-major-front), sets the container name to realest_estate_frontend, maps the container's port 3000 to the host's port 3000, and sets a dependency on the realest_estate_backend service.
6. volumes: This is a list of the volumes that will be created by Docker Compose.
7. mydb_data: This volume is created to store the data of the MySQL container persistently.
8. realest_estate_logs and realest_estate_media: These volumes are created to store the logs and media files of the backend container persistently.
9. The wait.sh is a shell script. It checks if the MySQL Server is up and running on the mydb host at port 3306 using the nc command.
10. If the server is not up, it waits for 3 seconds and then checks again.

11. Once the server is up, it runs the Django migration commands to create the database schema.
12. It creates a superuser with the username adminbrc, email adminbrc@example.com, and password password123.
13. Finally, it starts the Django development server on 0.0.0.0:8000 which listens to all incoming requests.

Continuous Deployment:

Ansible is a suite of software tools that enables infrastructure as code. It is open-source and the suite includes software provisioning, configuration management, and application deployment functionality.

Inventory file:

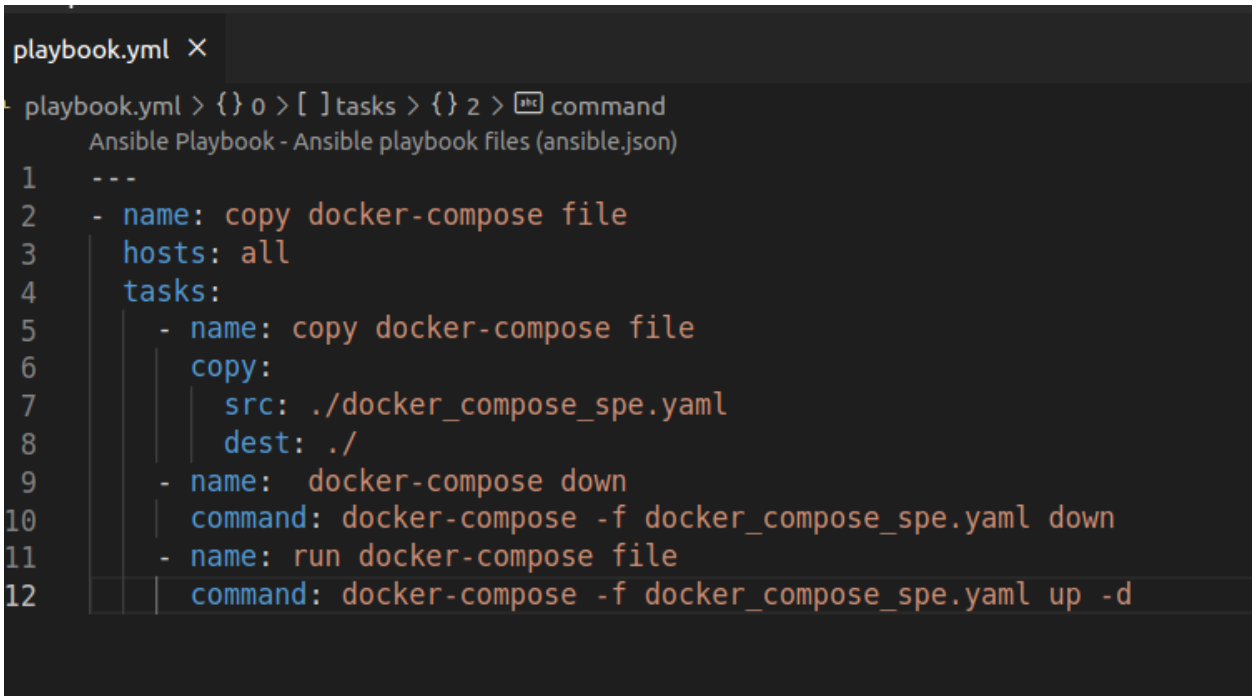


[ubuntu18] is a group name used to represent a group of hosts in Ansible. Here, it is used to group a single host with the specified IP address 172.16.128.119 (which is my IP Address)

172.16.128.119 is the IP address of the host that Ansible will operate on.

ansible_user="ansible_usr" is an optional parameter that specifies the username that Ansible should use to connect to the host via SSH.
Here, the username is set to ansible_usr.

Playbook file:



This ansible playbook.yml contains three tasks that will be executed on all hosts specified in the inventory file.

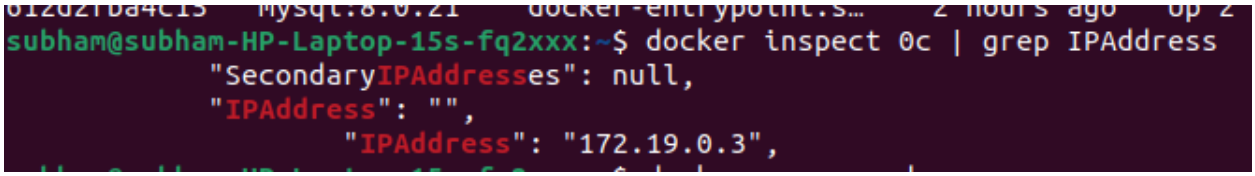
- The first task is to copy the docker_compose_spe.yaml from the local machine to the target host. The copy module is used for this task, which takes src and dest arguments to specify the source and destination file paths.
- The second task is to stop any running containers and remove the network, volumes, and images associated with the containers using the docker-compose down command. This command is executed using the command module.

- The third task is to start the containers defined in the docker_compose_spe.yaml file in detached mode using the docker-compose up -d command. Again, this command is executed using the command module.

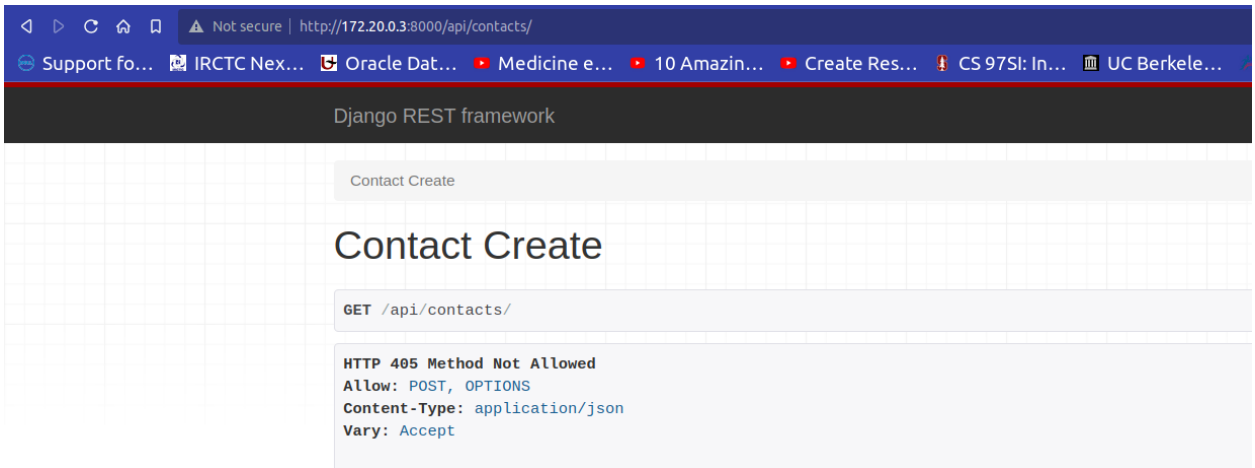
The application will start running but as a background process, to access the application we need to run localhost:8000 or localhost:3000. But to run the individual container with their assigned ip we need to do the following:

- To access the application through localhost:8000, you need to first identify the IP address of the Docker container.
- You can use the **docker ps** command to get a list of running containers. Look for the container that is running your Django application and note its container ID.
- Then run this command:

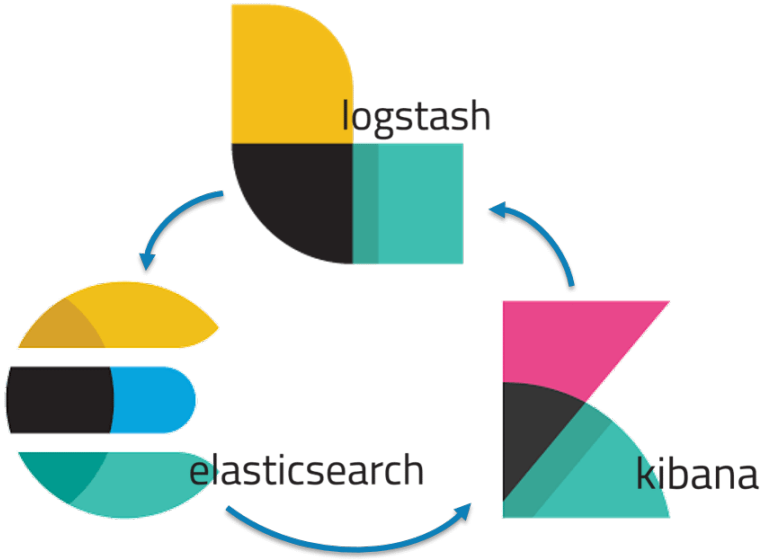
  docker inspect <container ID> | grep IPAddress



- Now run the application in the Browser using the command http://<container IP address>:8000 ,
  For example : http://172.20.0.3:8000



CONTINOUS MONITORING:



First we fetch our log files from inside the container where our backend application is running:

docker exec -it c1 /bin/sh

```
subham@subham-HP-Laptop-15s-fq2xxx:~$ docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED          STATUS          PORTS
c1d967b36cb9   sbrc1996/realest_estate  "./wait.sh"              16 minutes ago   Up 16 minutes   0.0.0.0:8000->8000/tcp, :::8000->8000/
b5be7e4703dd   mysql:8.0.21             "docker-entrypoint.s…"   16 minutes ago   Up 16 minutes   33060/tcp, 0.0.0.0:30000->3306/tcp, ::
subham@subham-HP-Laptop-15s-fq2xxx:~$ docker exec -it c1 /bin/sh
/app # ls
accounts       contacts       listings       logs           manage.py      media          realest_estate  realtors       wait.sh
/app # cd logs
/app/logs # ls
```

Now see where the log files are then fetch it and place it in your local using this command.

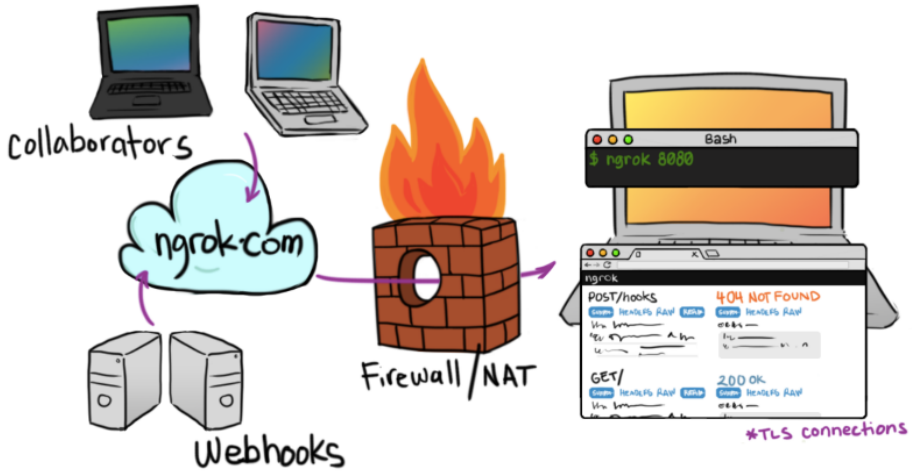
docker cp <container_name>:<path_to_file> <host_path>

docker cp realest_estate_backend:/app/logs/realest_estate_info.log /home/subham/Desktop/

Now we have our log with us.. We need to upload it to the ELK stack. But for the sake of simplicity, we will upload it manually in Kibanna or we could use the Filebeat to upload it.

Below are the Kibana visualizations for the log file, based on different fields:-

Ngrok:-

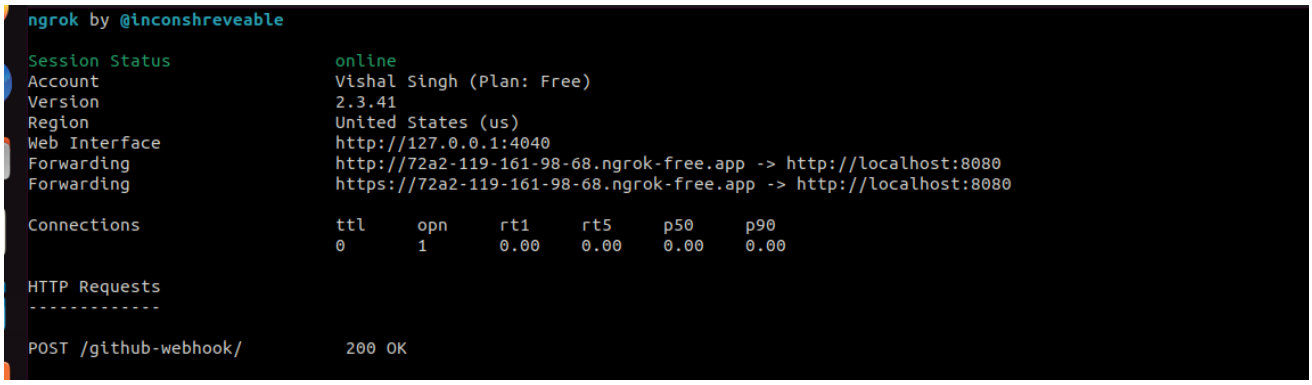


Ngrok is a tool that allows you to expose a local server to the internet by creating a secure tunnel. It provides a public URL that redirects incoming

requests to your local server, enabling you to test and share your web applications without deploying them to a public server.

Ngrok can be used to test webhooks locally by creating a tunnel to your local development server, allowing you to receive webhook payloads from third-party services like GitHub.

To start ngrok in terminal give command: ngrok http 8080
Then ngrok will provide a public URL to receive payload from Github. So, as soon as any changes are made in github it will deliver payload to Jenkins and Jenkins will trigger the new build for the payload received.



SECURITY:

For security we have used JWT or JSON Web Tokens. JWT (JSON Web Token) is a form of transmitting a JSON object as information between parties. Let's learn more about what JWTs are and how they work.

We use JWTs to exchange information in cases where they are signed – for example using public-private key pairs to make sure that the integrity of the information is not compromised since the payload and header are used to compute signatures.

Working of JWT:
You can get a JWT through logging in with a username and password. In exchange the server returns an access and refresh token in the form of a JWT. The tokens access resources on the server.
The lifetimes of the access and refresh tokens vary since access tokens last for five minutes or less while the refresh tokens can last for 24 hours. But you can customize the timelines of both types of tokens.
Parts of JWT:

```
header.payload.signature
```

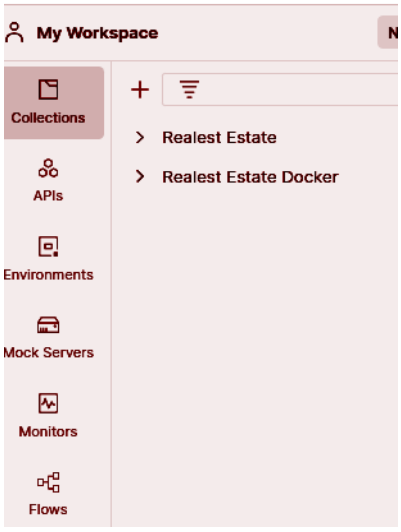
Illustration of a JWT in simple English

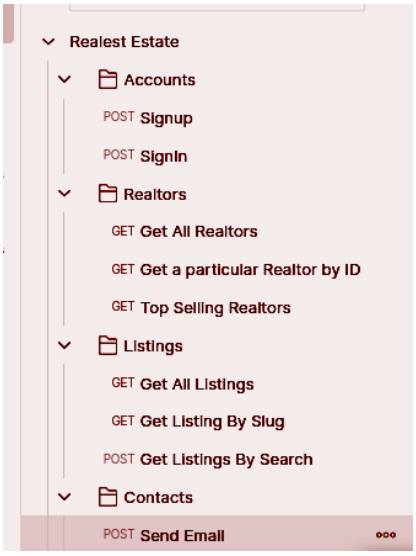
```
header = eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9F5
payload = eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTQzODI4NDMxLCJqdGkiOiI3ZjU5OTdiNzE1MGQ0NjU3OWRjMm
signature = Ju70kdcaHKn1Qaz8H42zrOYk0Jx9kIciuhkTn9Xx7vhikY
```

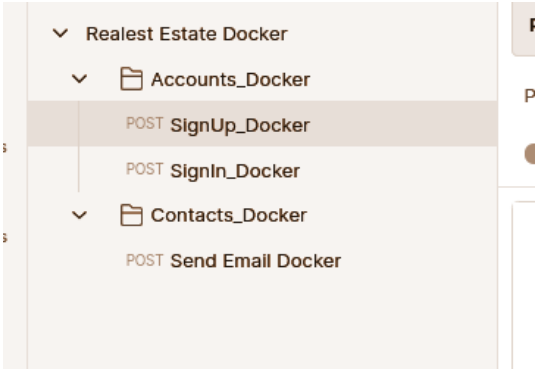
More details here:
https://www.freecodecamp.org/news/how-to-use-jwt-and-django-rest-framework-to-get-tokens/

API TESTING:

We have a total of 9 api in our project, during the development our backend and our frontend code was developed separately.
Hence while developing the backend to test if the APIs were indeed working we used POSTMAN. Here are some of the screenshots to understand it better:
The first folder was for developing the application in my locally.
The second folder was for testing if the APIs are working in the docker container.

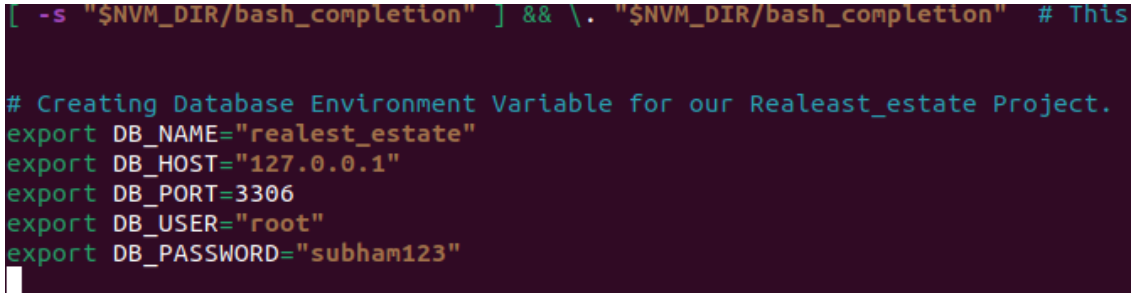






PROBLEMS FACED:

Problems faced by me while creating this project were many.

1. The first major issue faced by me was the library djangorestframework-simplejwt was not compatible with the django virtual environment, hence I had to proceed without using the virtual environment.
2. Sending email through Google API and using the SMTP port 527 was very much new to me hence I had to do a lot of research in coming up with the solution.
3. The docker daemon process doesnot have the sufficient permissions hence we need to manually add these permissions manually on every restart using the cmd: **sudo chmod 666 /var/run/docker.sock**
4. The django settings.py was provided the required environment variables in docker container to get the required details for a successful connection to the MYSQL database running in another container. But when I ran that same code in local it was failed due to absence of those env variables. Hence I created those variables in my local bashrc file. In these steps:
    - **nano ~/.bashrc**
    - 

    - To apply the changes, either open a new terminal window or run the following command in the current terminal: **source ~/.bashrc**
5. Even after creating these variables our Jenkinsfile was unable to fetch the environment variables from the **bashrc** file, hence we had to manually declare the environment variables in the Jenkinsfile to solve this issue.

6. Jenkins was also unable to read the environment variable of the python path, strangely on running the test scripts manually in the directory /var/lib/jenkins/workspace/<curr_wsk> the test cases ran perfectly but on running them through the Jenkinsfile was not possible hence I had to manually install all the dependencies in the Jenkins workspace to ensure smooth testing.

REFERENCES:

1. Dockerize a Django App: https://blog.logrocket.com/dockerizing-django-app/
2. Dockerfile for Django and React Application and Running it with the DockerCompose: https://www.honeybadger.io/blog/docker-django-react/
3. Logging in Django: https://www.freecodecamp.org/news/logging-in-python-debug-your-django-projects/
4. Dockerizing Django and MySQL using Docker Compose: https://roytuts.com/docker-compose-dockerizing-django-mysql-app/
5. GitHub Error resolution: https://www.a2hosting.com/kb/developer-corner/version-control-systems1/403-forbidden-error-message-when-you-try-to-push-to-a-github-repository/
6. JWT: https://www.freecodecamp.org/news/how-to-use-jwt-and-django-rest-framework-to-get-tokens/
7. Git and GitHub: https://www.freecodecamp.org/news/git-and-github-for-beginners/
8. Docker and DockerHub: https://www.freecodecamp.org/news/the-docker-handbook/
9. React: https://legacy.reactjs.org/docs/getting-started.html
10. ChatGPT