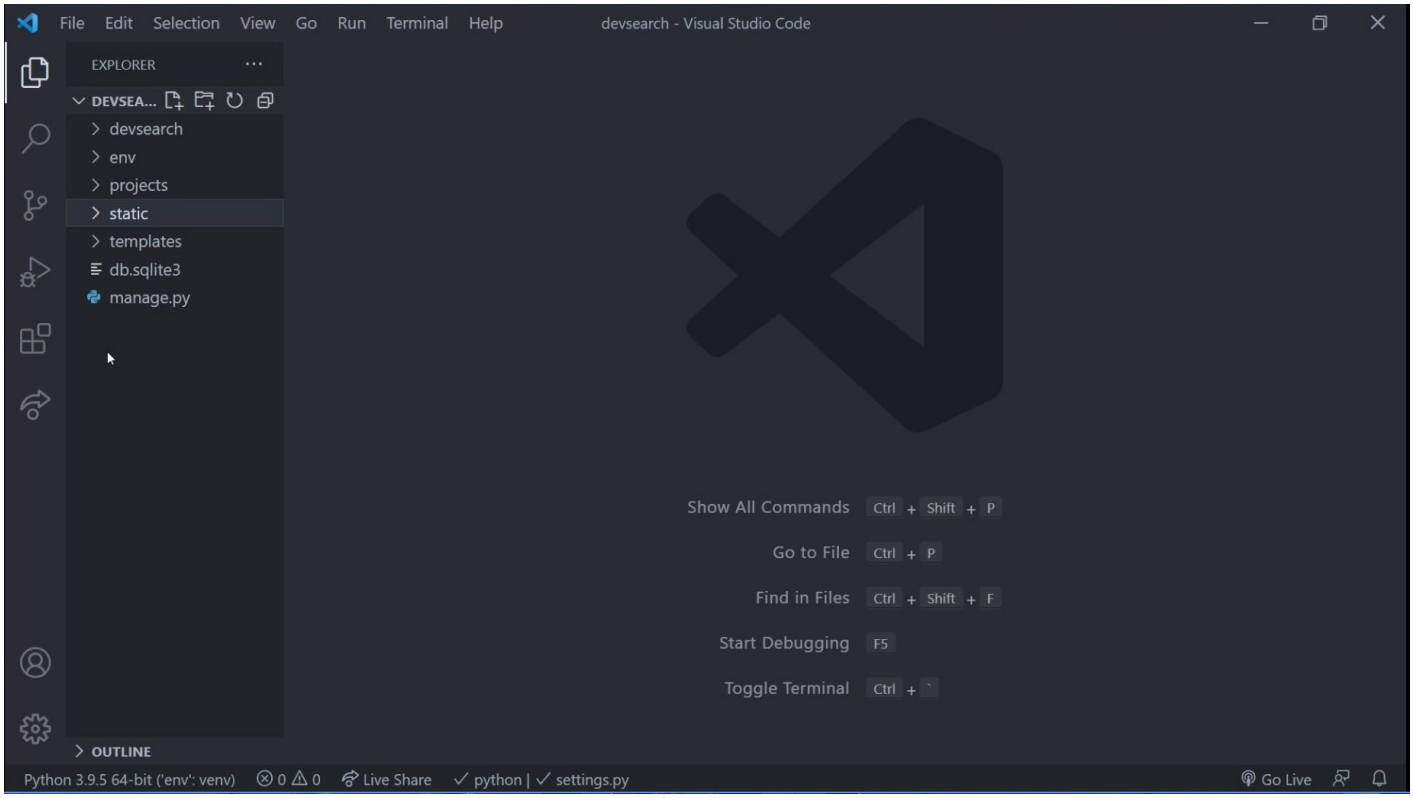
Template Styling and Static Files

Static files in Django projects are all CSS, JavaScript, image file (jpg, tiff, png, gif, icons, svg, etc), and in general, any kind of extrenal file to django.

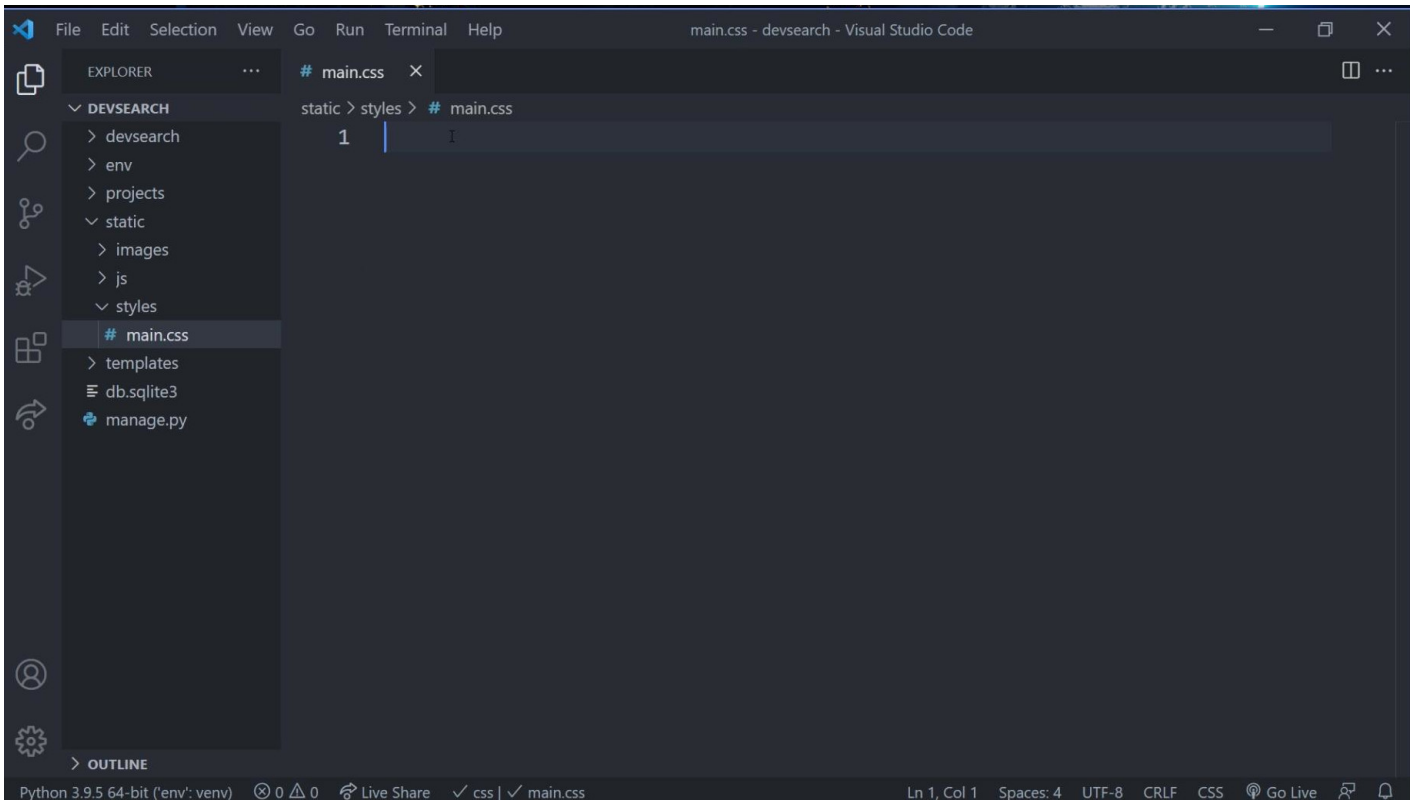
HTML allows us to style our templates directly. However, this method does not allow code reuse without having to heavily modify each page individually. This is the reason a CSS file is a page styling instructions in a separate file (a static file).

Typically in Django, CSS files are place in a directory at the project level, as shown here:



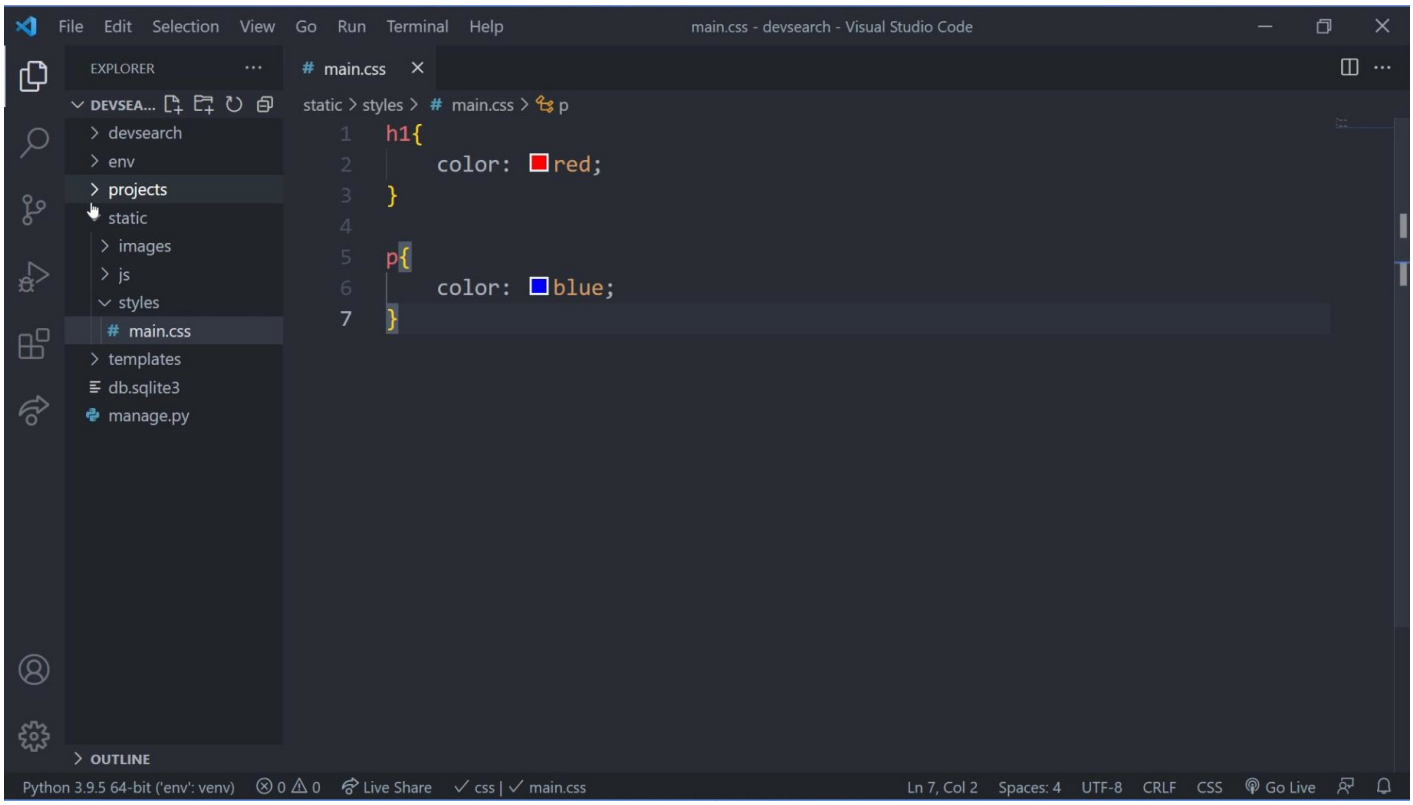
We named our static file folder "static." For structuring reasons, the static folder also contains different folders named on the basis of the file type they contain:

```
+ static
+---images          # Contains icons, images, logos, svg, etc.
+---js              # Contains javascript files.
+---styles          # Our CSS files usually go here
```



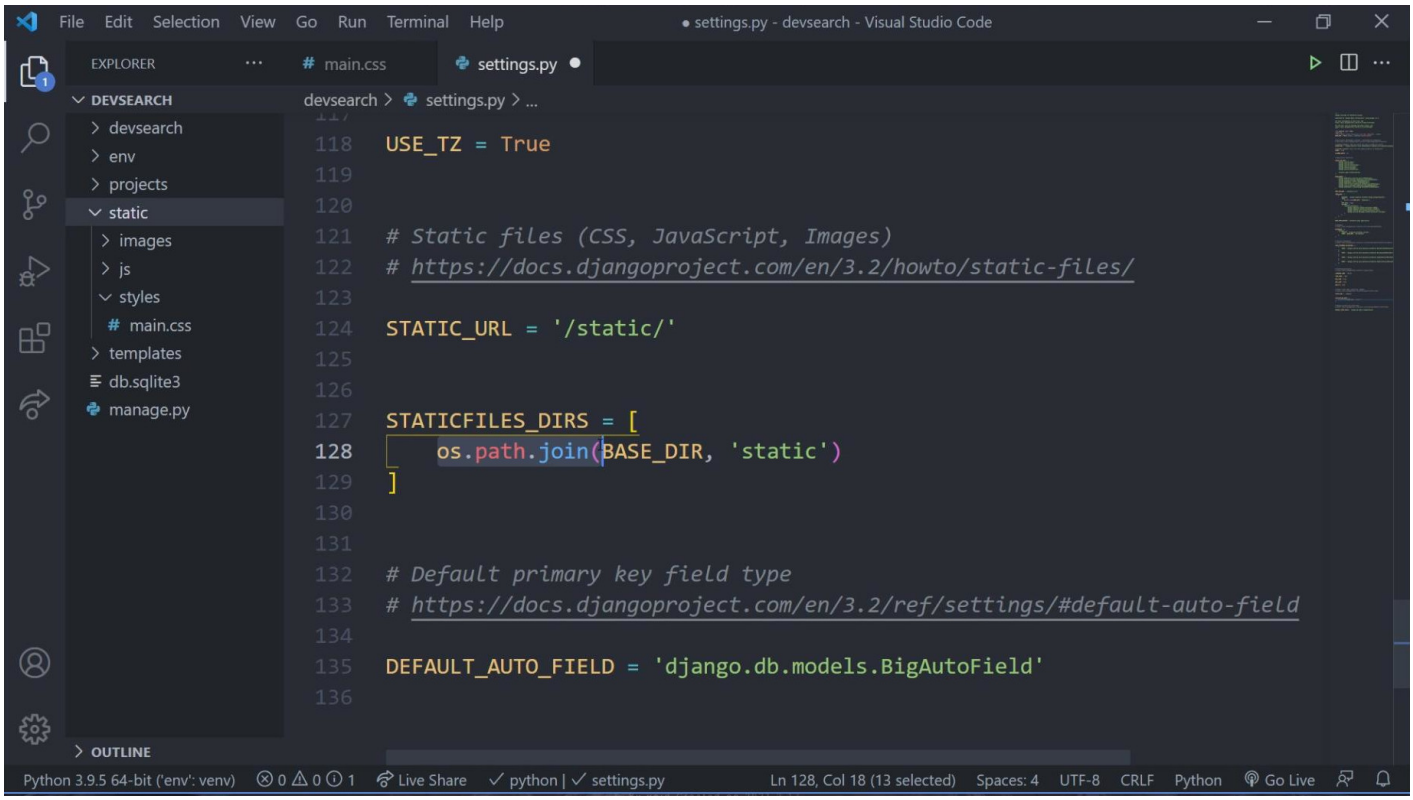
For the project styling, we will use the file /static/styles/main.css

Just to get an output, we'll type the following code into the main.css file we just created:



Next, we need to inform Django where to find our project's static files. For that, we have to include into our /devsite/settings.py file.

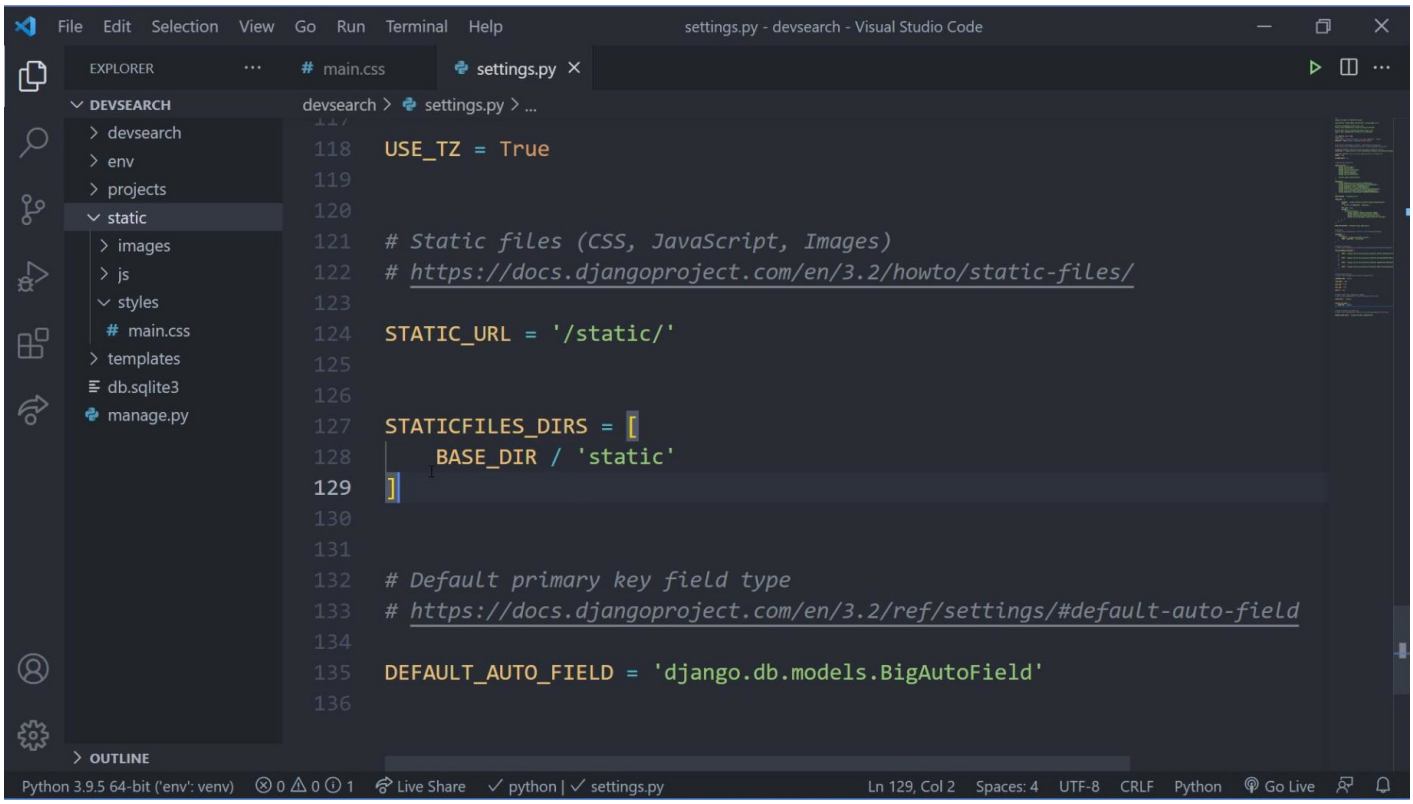
At the bottom of the file, just below the line STATIC_URL = '/static/', we have to add the variable STATICFILES_DIRS:



The variable name STATICFILES_DIRS is the name defined by the Django documentation, it is not a random variable name. The variable also accepts the path definition using pathlib.Path:

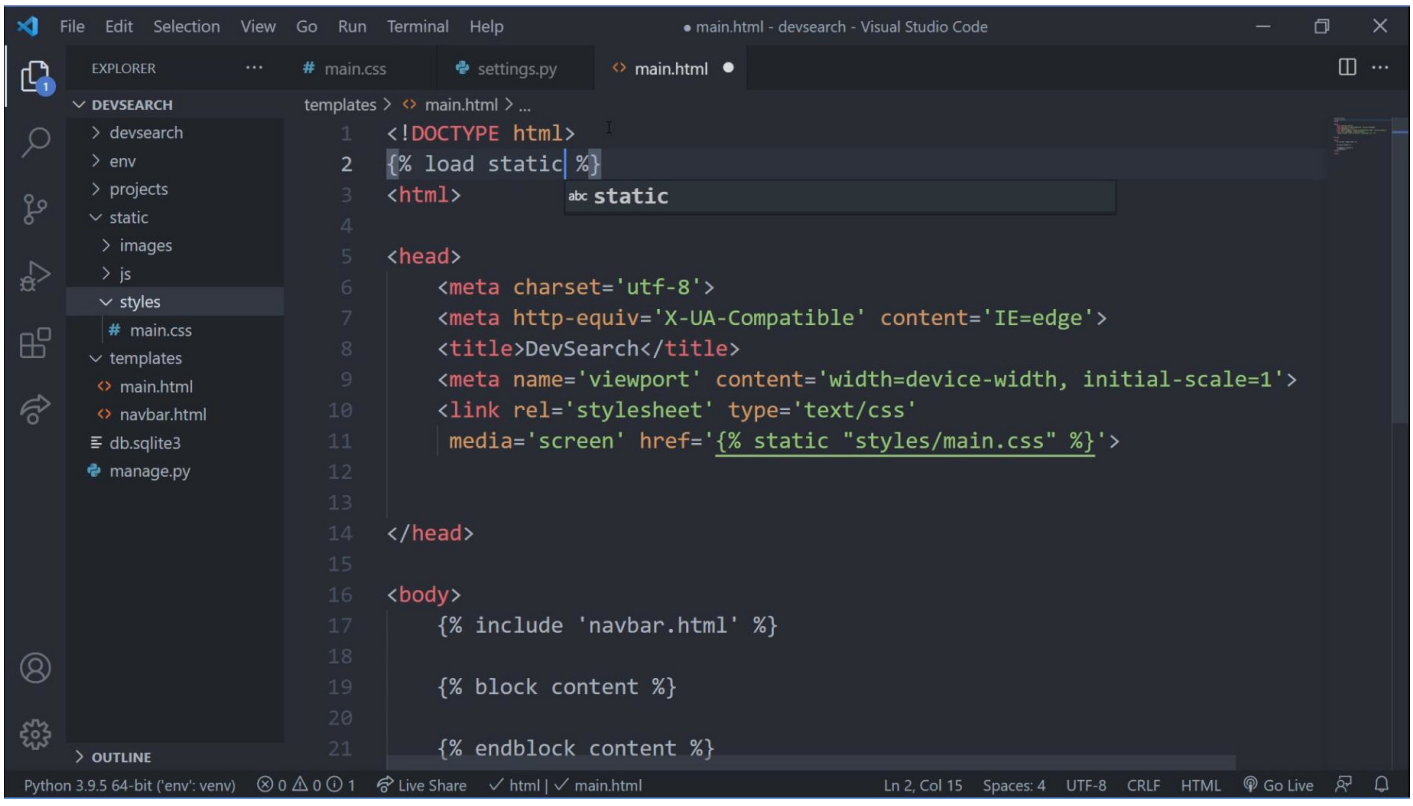
```
STATICFILES_DIRS = [
    Path(BASE_DIR, 'static'),
]
```

Another acceptable way to pass the new static path to django is:



but this is valid only in newer django versions.

Next, we need to use our style /static/styles/main.css with our django project. There are two things We need to modify our base html template in /template/main.html:



First, the template needs to know that it has to use a static file. For that we type in the template:

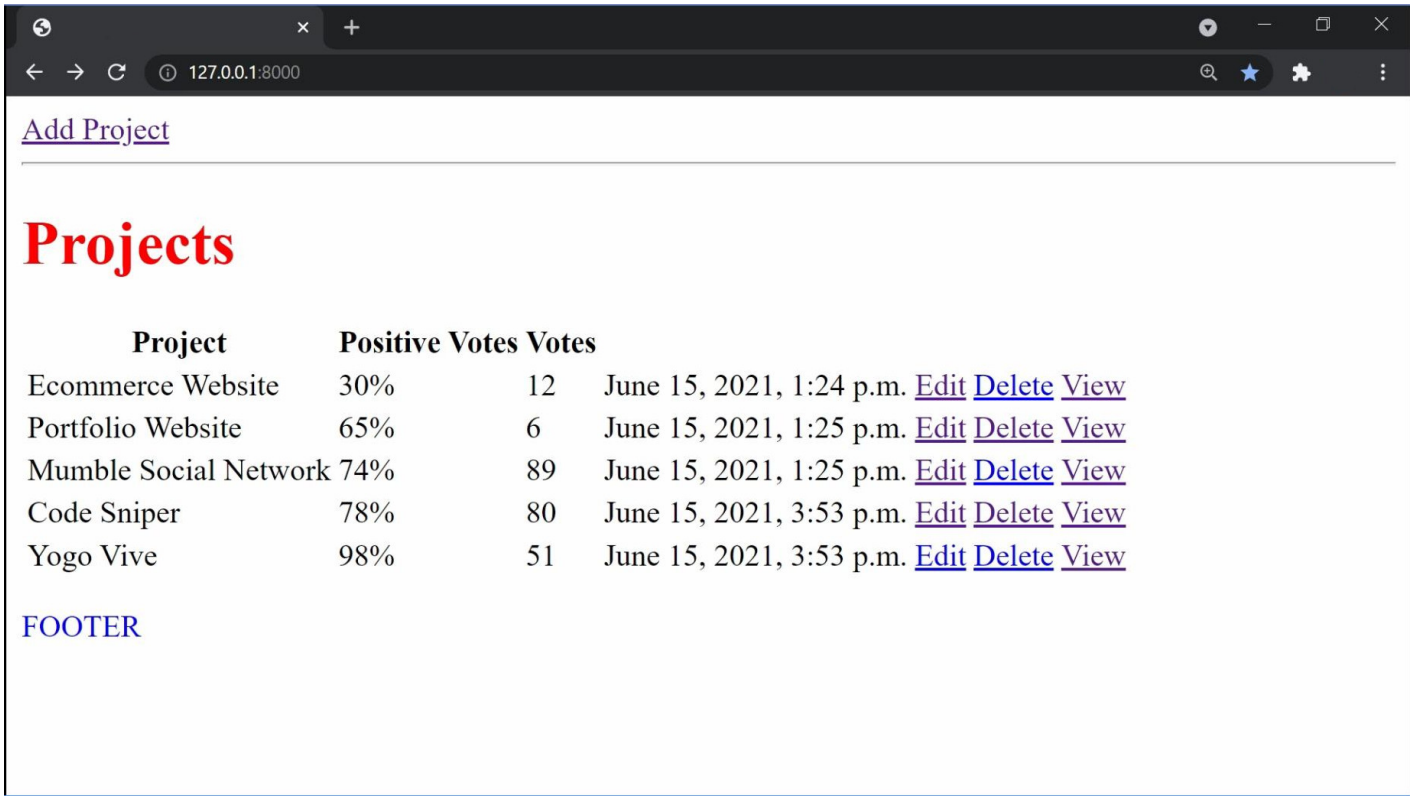
```
{% load static %}
```

And in order to include the specific styling files we include a variable:

```
href='{% static "styles/main.css" %}'
```

the static is a dynamic direction to the /static/ folder and the "styles/main.css" is the path to the main.css file. This is how it links up our stylesheet to our new "main.css" file in the /static/styles/ folder.

With all these modifications in our application, we should see two differences in our projects page:



Notice that the Project title page is now red colored and the footer is blue colored when they used to be all black.

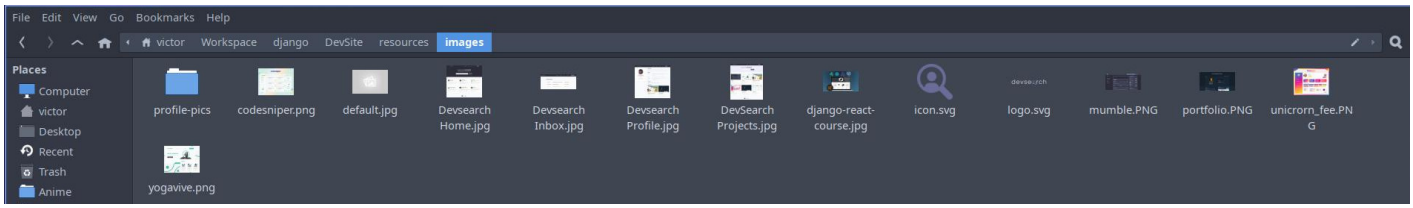
The ***h1*** in the main.css file refers to the HTML tag for header level 1 and the ***p*** refers to the HTML tag for paragraph.

All changes are automatic as long we keep using the specific styling with /static/styles/main.css.

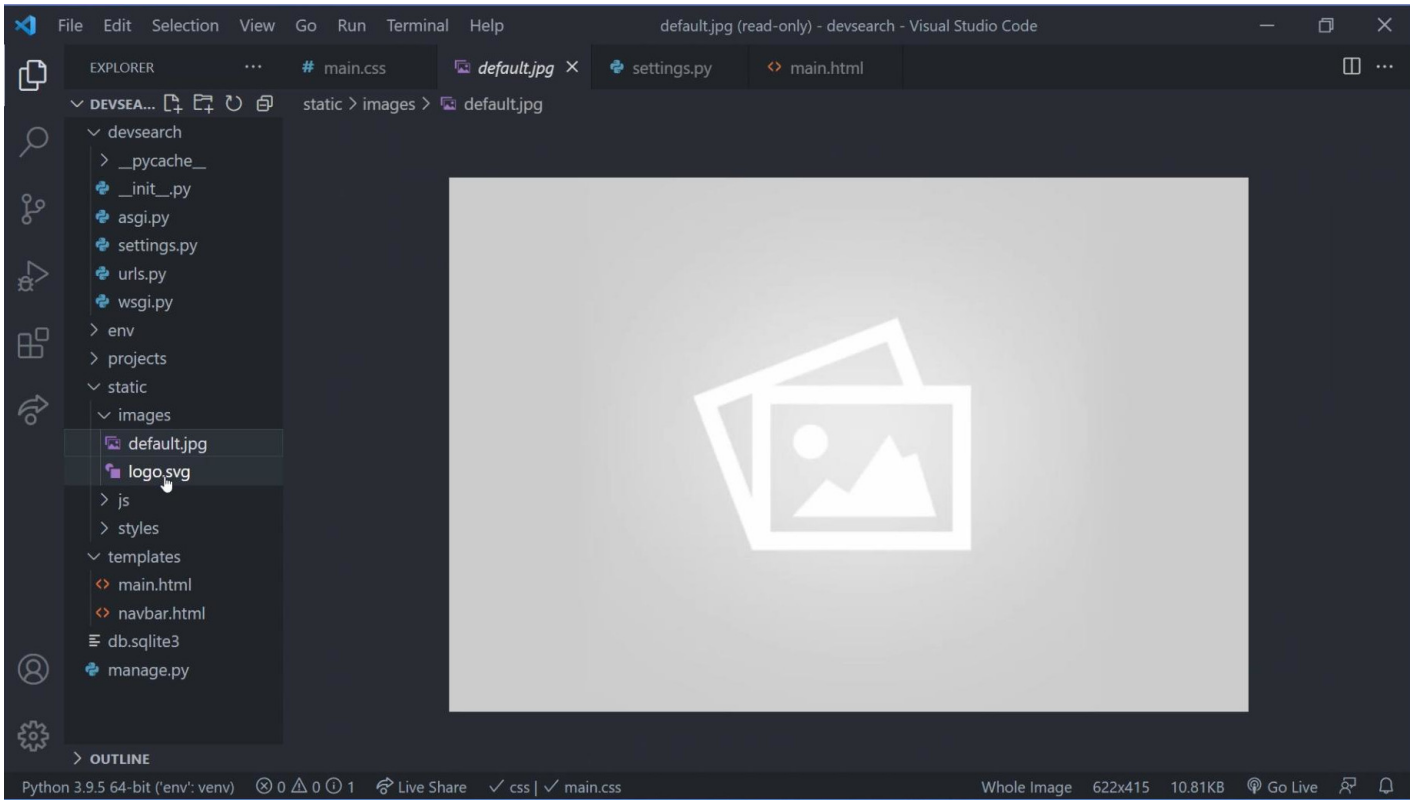
From now onwards, the specific styles in main.css are applied to our templates derived from the main.html template.

Add a logo to the Navbar

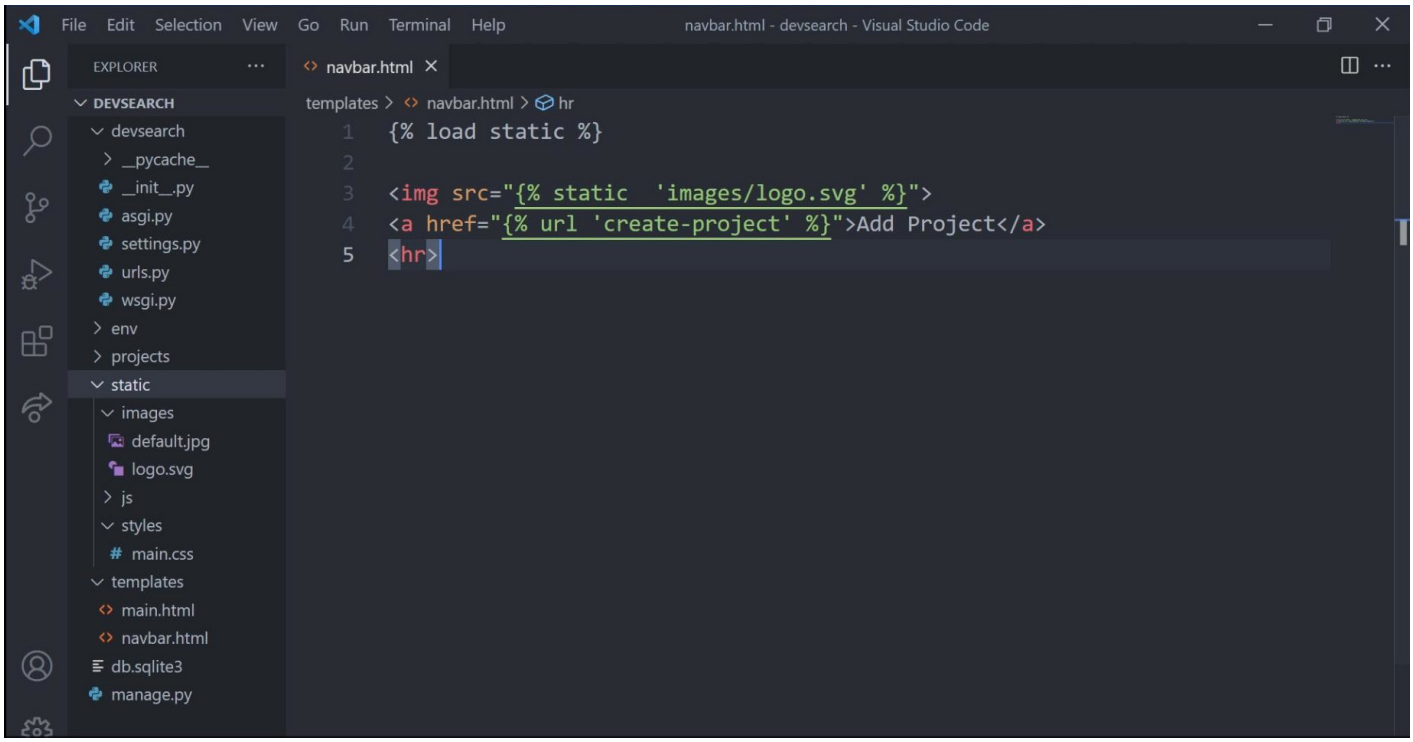
Copy the logo.svg and the default.jpg from the resources folder, that you downloaded from the repository, to the /static/images folder:



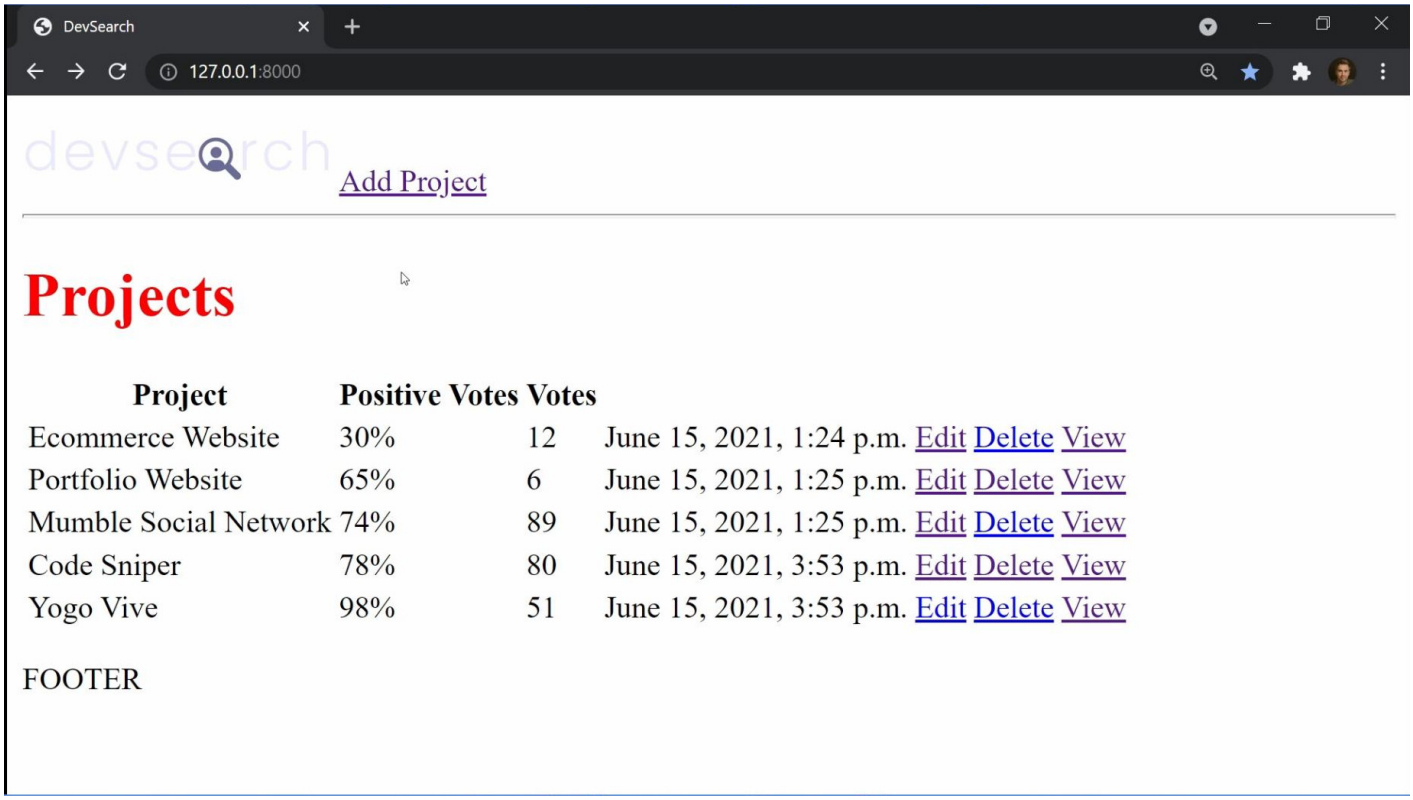
Place both files into your /static/images folder:



The following step is to modify the /templates/navbar.html file:



The navbar area should have now a logo image on the left top corner of the page:



if you right-click the logo and then open the image on a right tab, you'll find the full file path for the logo image. Try it!

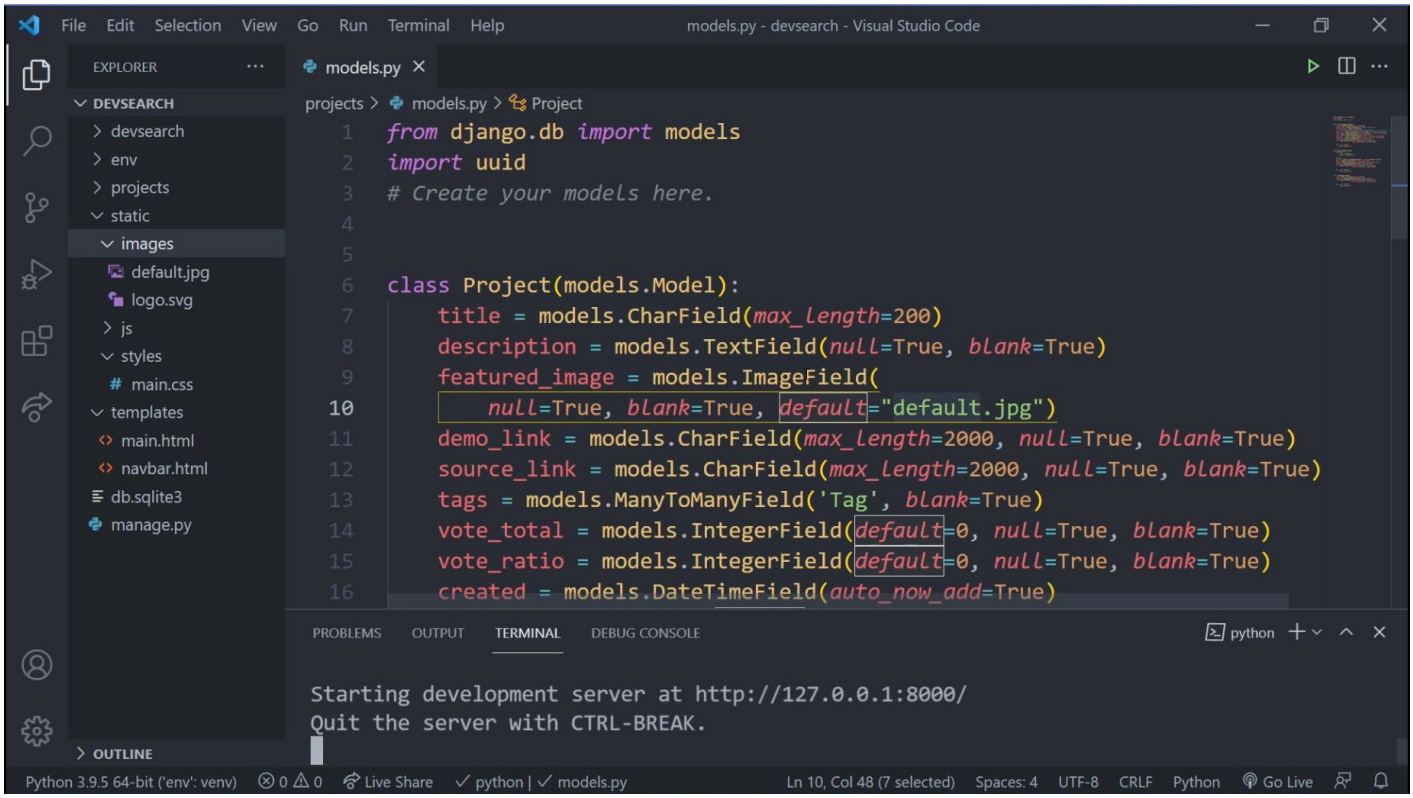
The static approach is enough for details such as logos or backgrounds, but if we want to add an image to each project in the projects page, this approach is inadequate.

There's another way to implement the images.

Adding Images for Each Project on the Projects Page

The way we're going to add individual images for each project is by storing them in a folder using our database to store the full image file path and retreiving each image from that folder.

In order to do this, we'll have to modify our project model in the **/projects/models.py** file.



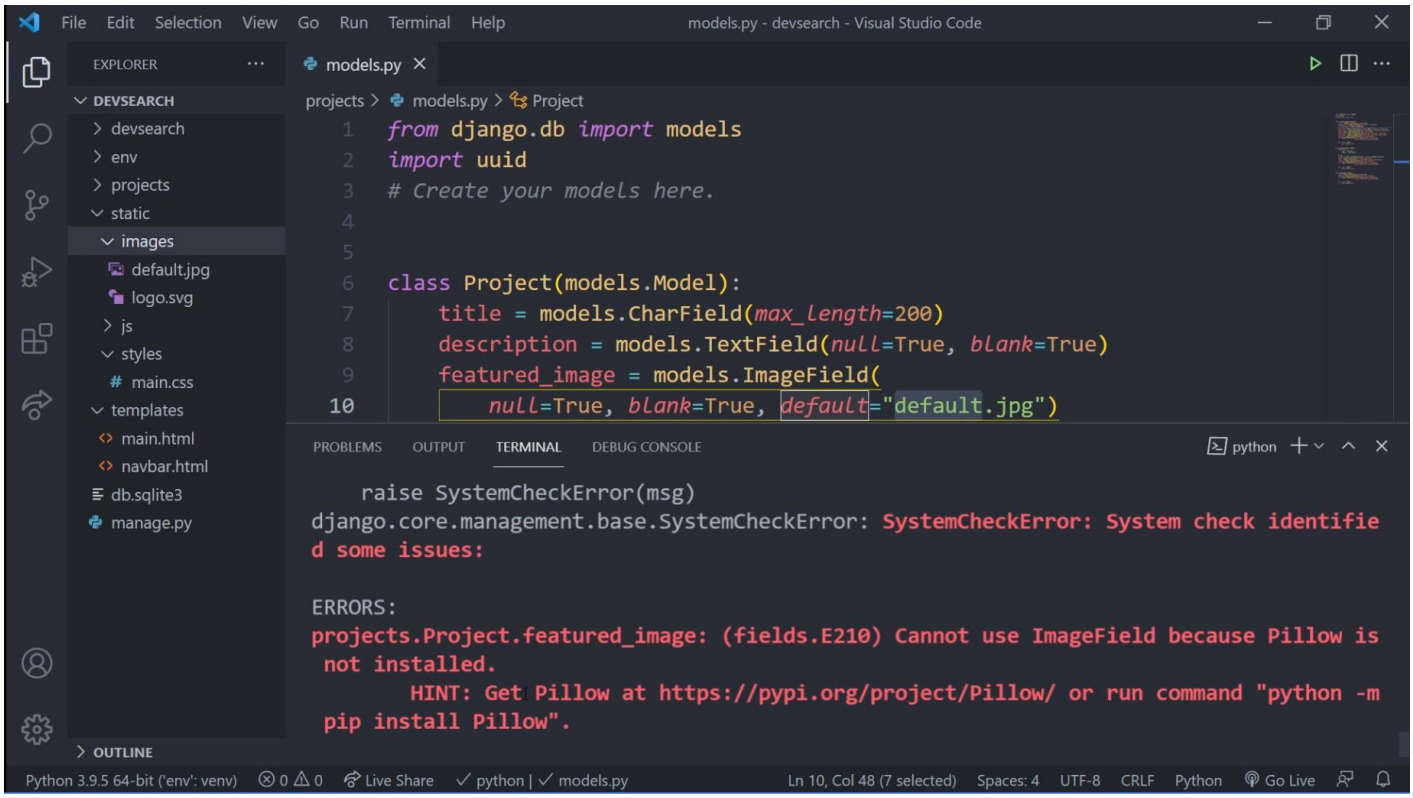
We included the code into the Project(models.Model) class:

```
featured_image = models.ImageField(null=True, blank=True,
default="default.jpg")
```

The models.fieldImage is set so it can be *null* (we don't have to define a specific file path), and *blank* (shows a blank image or default image if defined), and if that's the case, then use a default image instead *default="default.jpg"* which is our copied ***default.jpg*** image.

IMPORTANT NOTE:

If you didn't include the Pillow module during your environment set up you'll get an error (*fields.E210*):



To correct the error stop your django server and within your virtual environment run:

```
python -m pip install pillow
```

or if you used pipenv:

```
pipenv install pillow
```

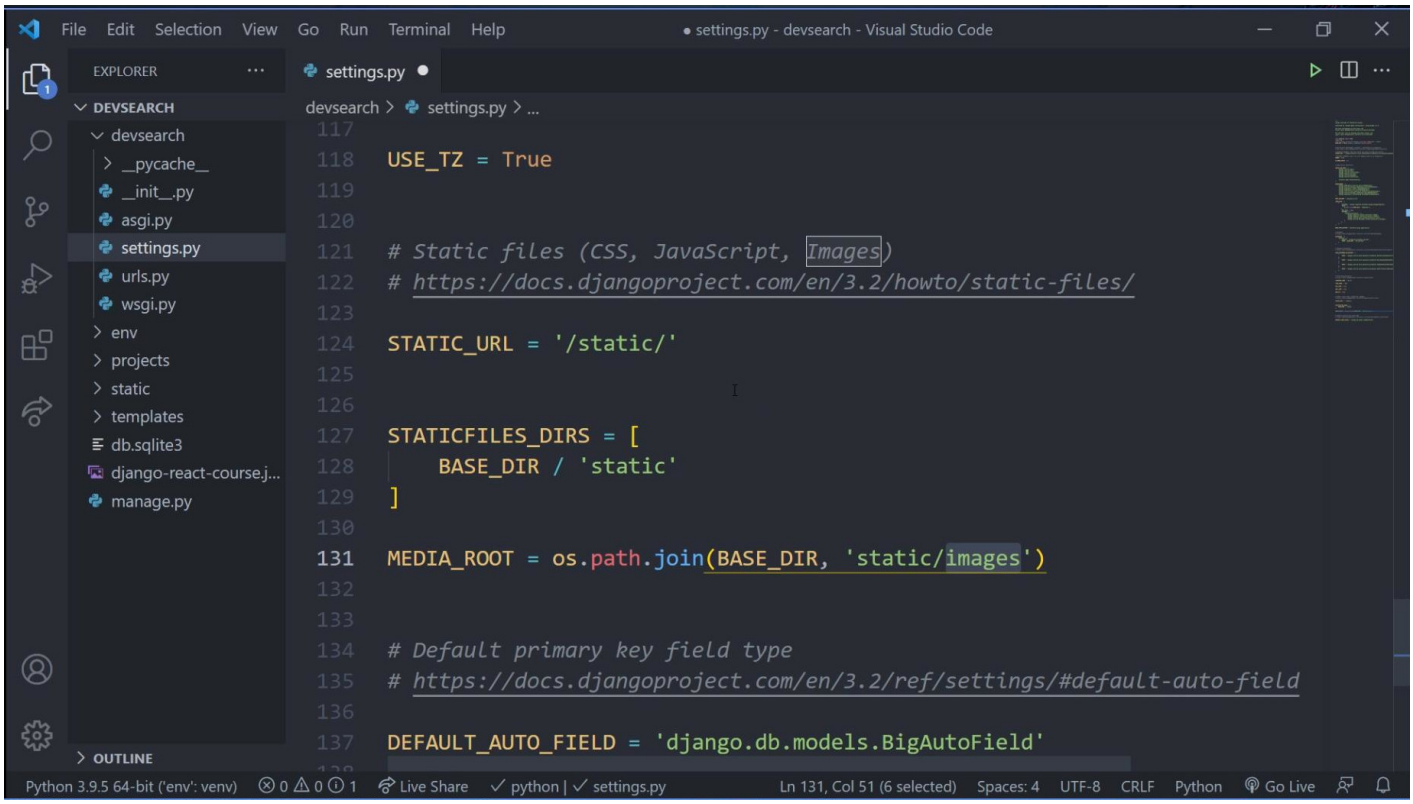
and check if the error disapeared when runing the server again:

```
python manage.py runserver
```

Do not forget to Migrate the Model Again! Run the commands:

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

Next, we need to indicate django in which folder to place the images. We do that by adding some code into the ***/devsite/settings.py***



and we added the code line:

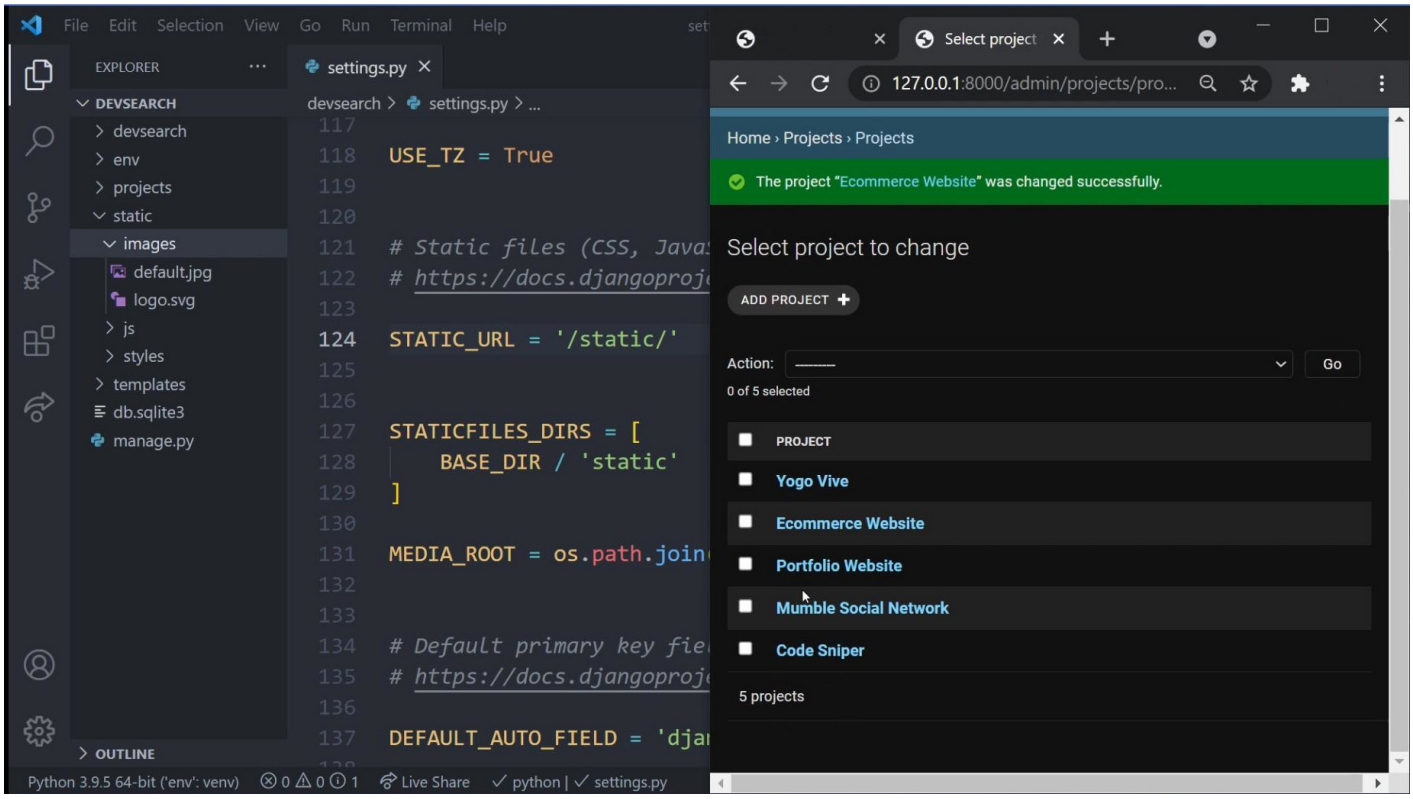
```
MEDIA_ROOT = os.path.join(BASE_DIR, 'static/images')
```

The **MEDIA_ROOT** constant is used by Django to identify any folder containing any **user generated content**. If you wish to place your project images in some other folder, then just create the folder and change the **MEDIA_ROOT** accordingly.

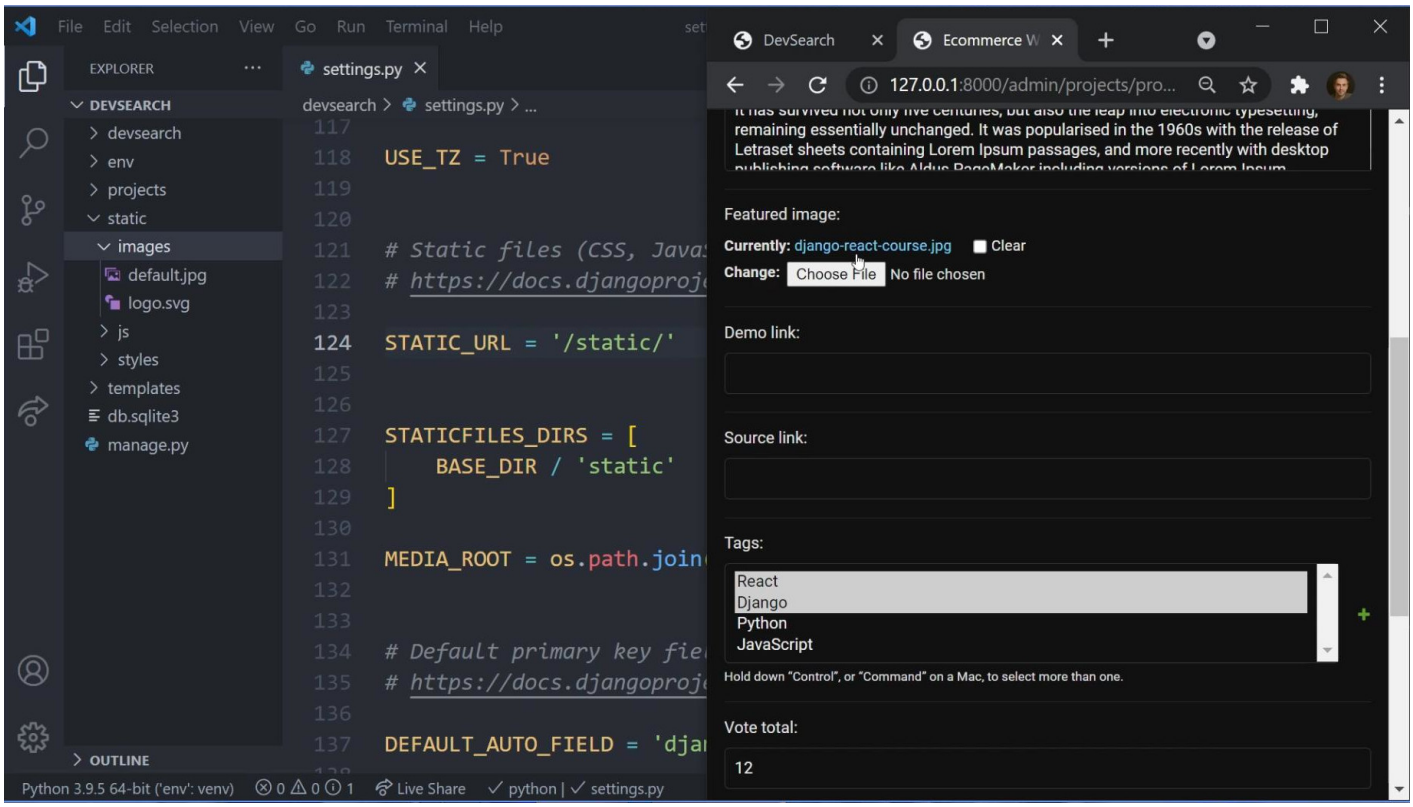
In this project we'll continue to use the **/static/images** folder just to simplify things.

Testing Where the Project Image is Placed

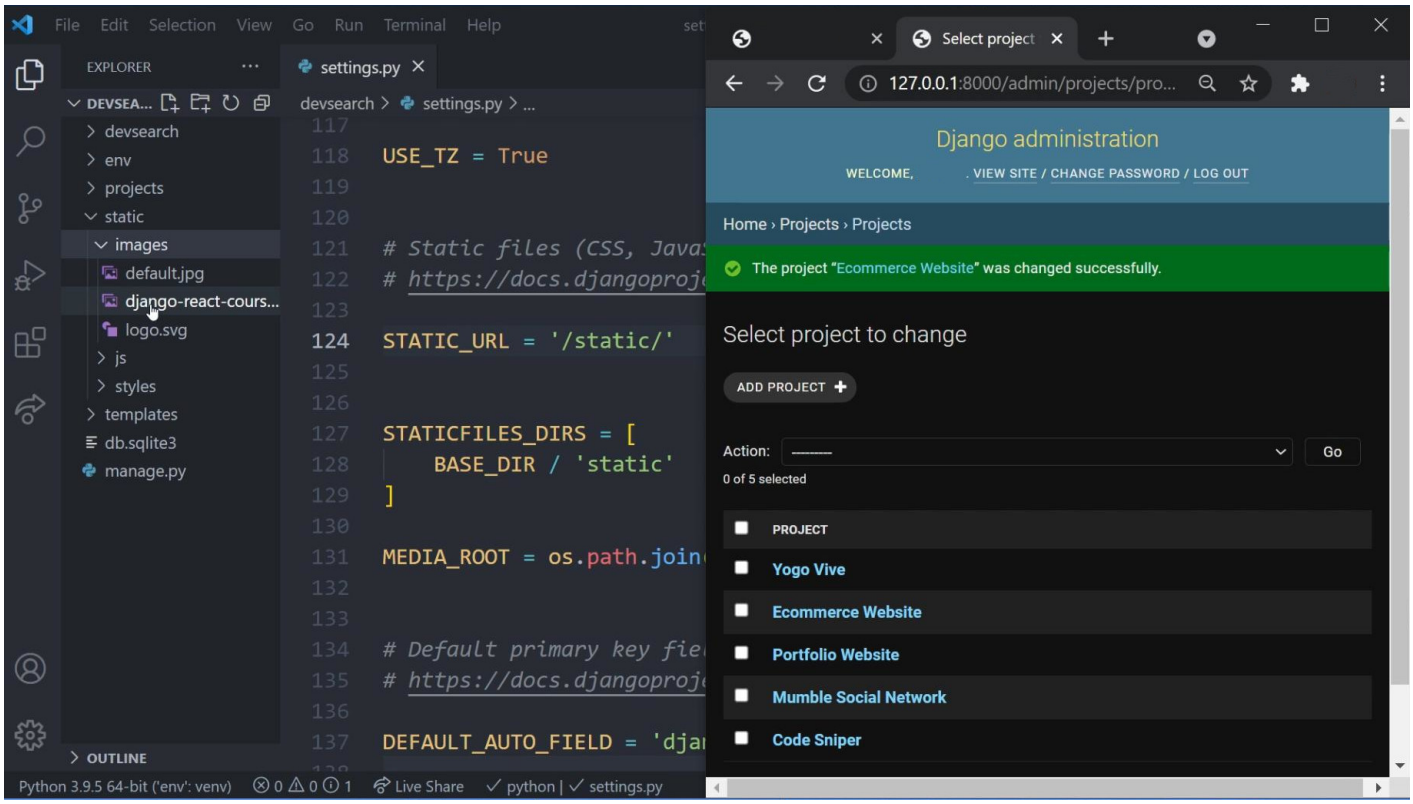
Open up the admin-panel. Open a project and then select an image to be uploaded. Use the resources folder from the git repository:



go to Featured Picture:



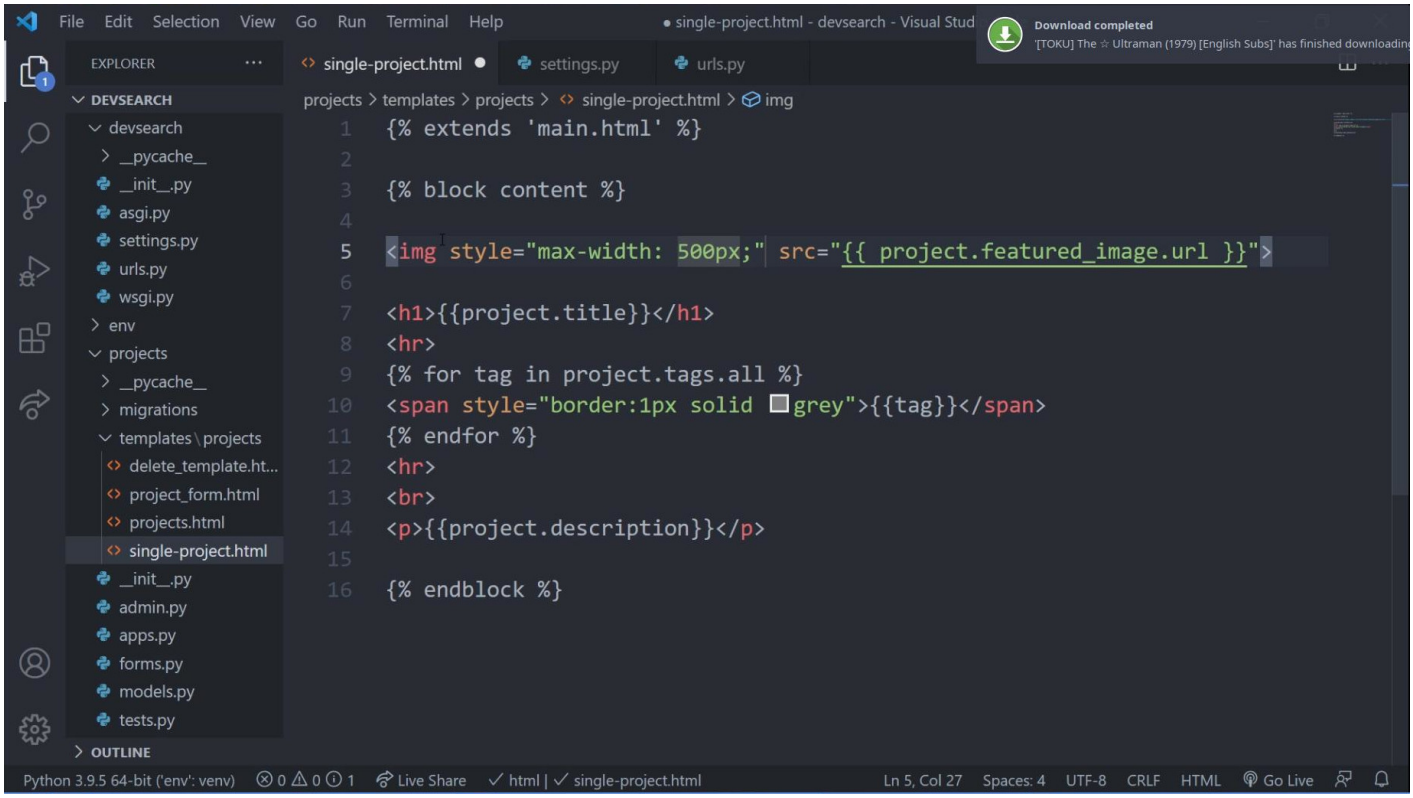
and select an image from the resources folder (django-react-course.jpg) and save it. You should see that the image was saved in ***static/images*** or the folder you had set up in /devsite/settings.py with the **MEDIA_ROOT** constant as shown here:



Rendering Individual Project Images

In this section we'll have to modify the single-project.html template to showcase images of the referred project. The whole process is a little bit tricky, so try to follow the guide as close as possible.

Let's begin. Open up the ***/projects/templates/projects/single-project.html*** file and modify the file as shown:



The *style* parameter limits the image size to a width of 500px (pixels)

The added line of code

```
<img src="{{ project.featured_image.url }}">
```

does several things:

1. Notice that we call the project member ***featured_image*** which is the full file path for where the image is actually stored, in this case, ***/static/images/<project_image>***.
2. We are using a dynamic link to the image's full path by using the ***.url*** member. Though this is done here for expedite purposes only. There are other ways to show the image.

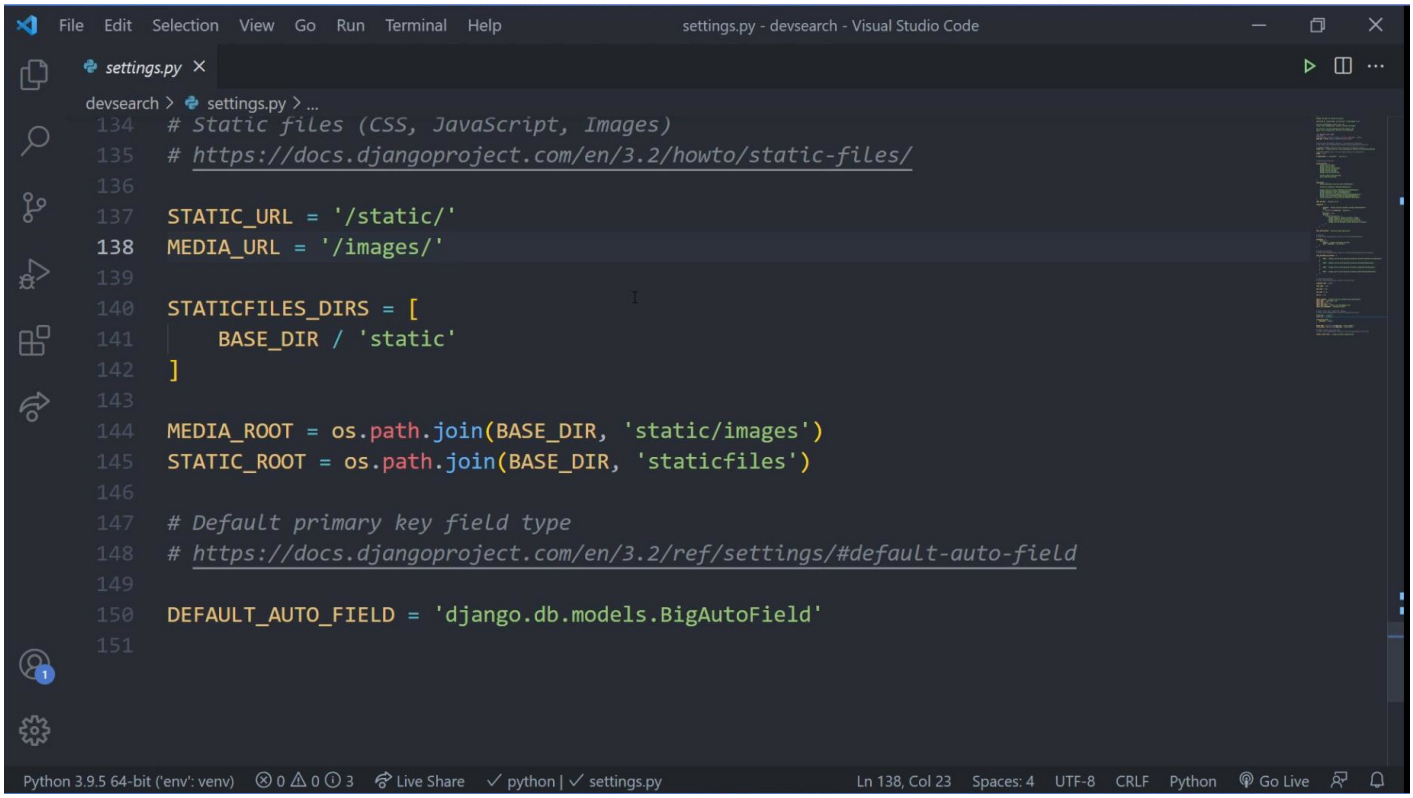
"MEDIA_ROOT" sets the absolute path to the directory where **uploaded** files are stored and setting **"MEDIA_ROOT"** never ever influences media file URL.

"MEDIA_URL" sets the directory(middle) part of media file URL between the host part and the file part of media file URL as shown below and setting **"MEDIA_URL"** never ever influence to the absolute path to the directory where uploaded files are stored:

```
            Host        Directory      File
              |             |           |
        <-------------> <----------> <-------->
https://www.example.com/media/images/orange.jpg
```

So, another step that has to be taken is to set up the **MEDIA_URL** in the ***/devsite/settings.py*** file. The **MEDIA_URL** just points to the middle part of the directory. It must end in a slash if set to a non-empty value. You will need to configure these files to be served in both development and production environments. If **MEDIA_URL** is set to an empty string, will in some cases cause errors or bugs.

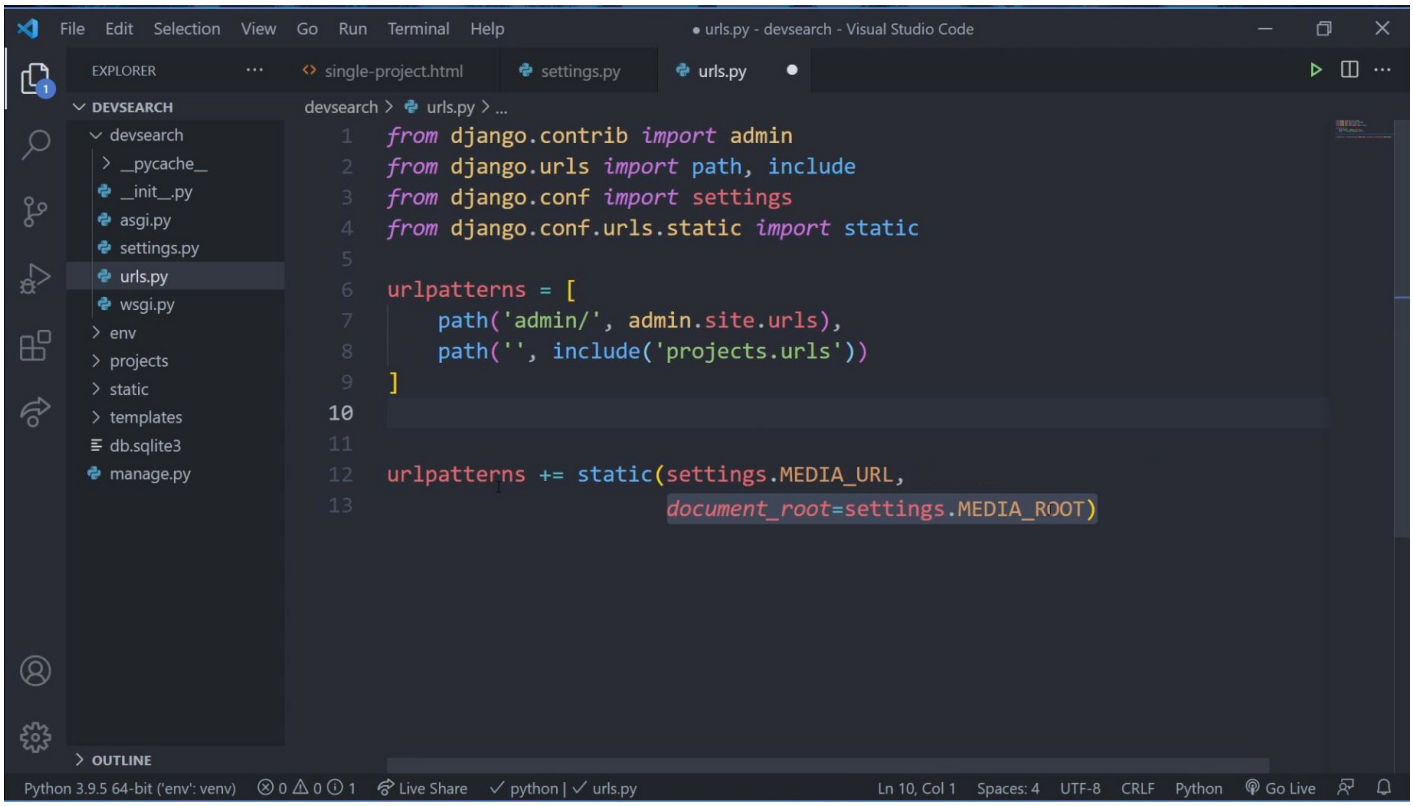
Check this post in stackoverflow for further explanation.



Alternative Ways to Show the Image

Modify the Django Project **urls.py** File

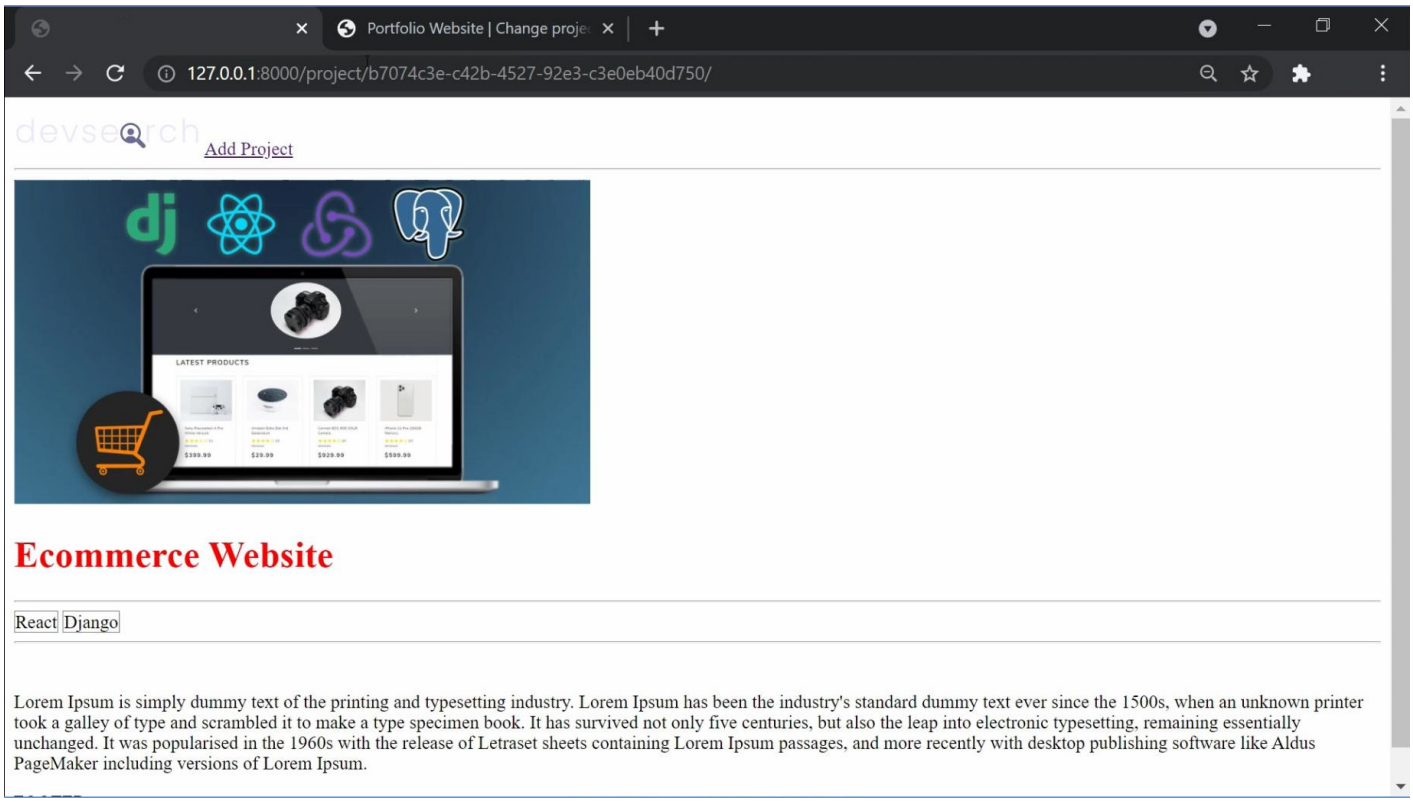
Next, we nee to do some modifications to the ***/devsite/urls.py*** file in order to find the individual project images.



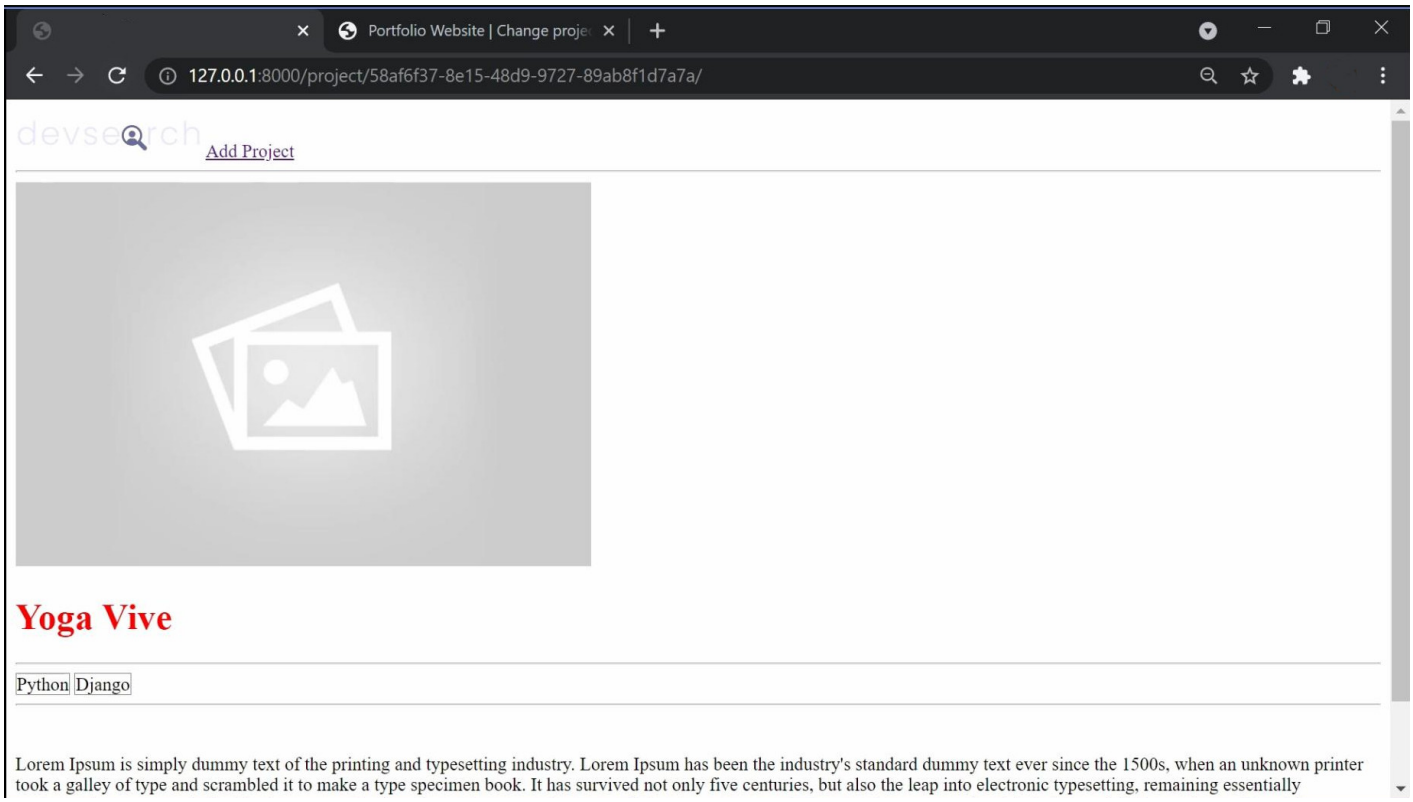
What the changes are doing is finding the constants **"MEDIA_URL"** and **"MEDIA_ROOT"** in order to associate the individual image file path in the ***/static/images*** folder and convert those file paths into a url and making the files visible through the associated url file path.

The conversion from file path to url is done by the static function imported from ***django.conf.urls.static***

The end result should look something like this:



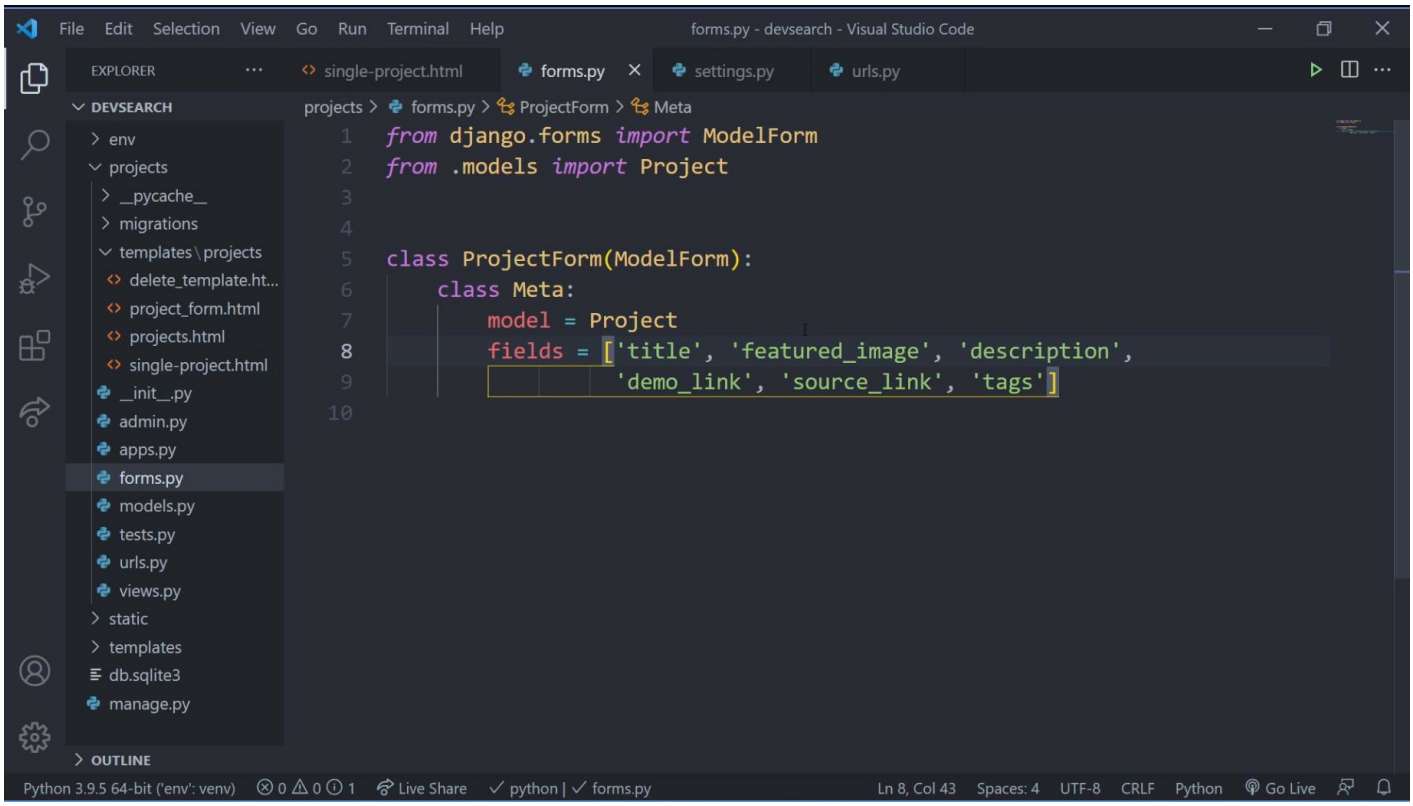
and one with the default image:



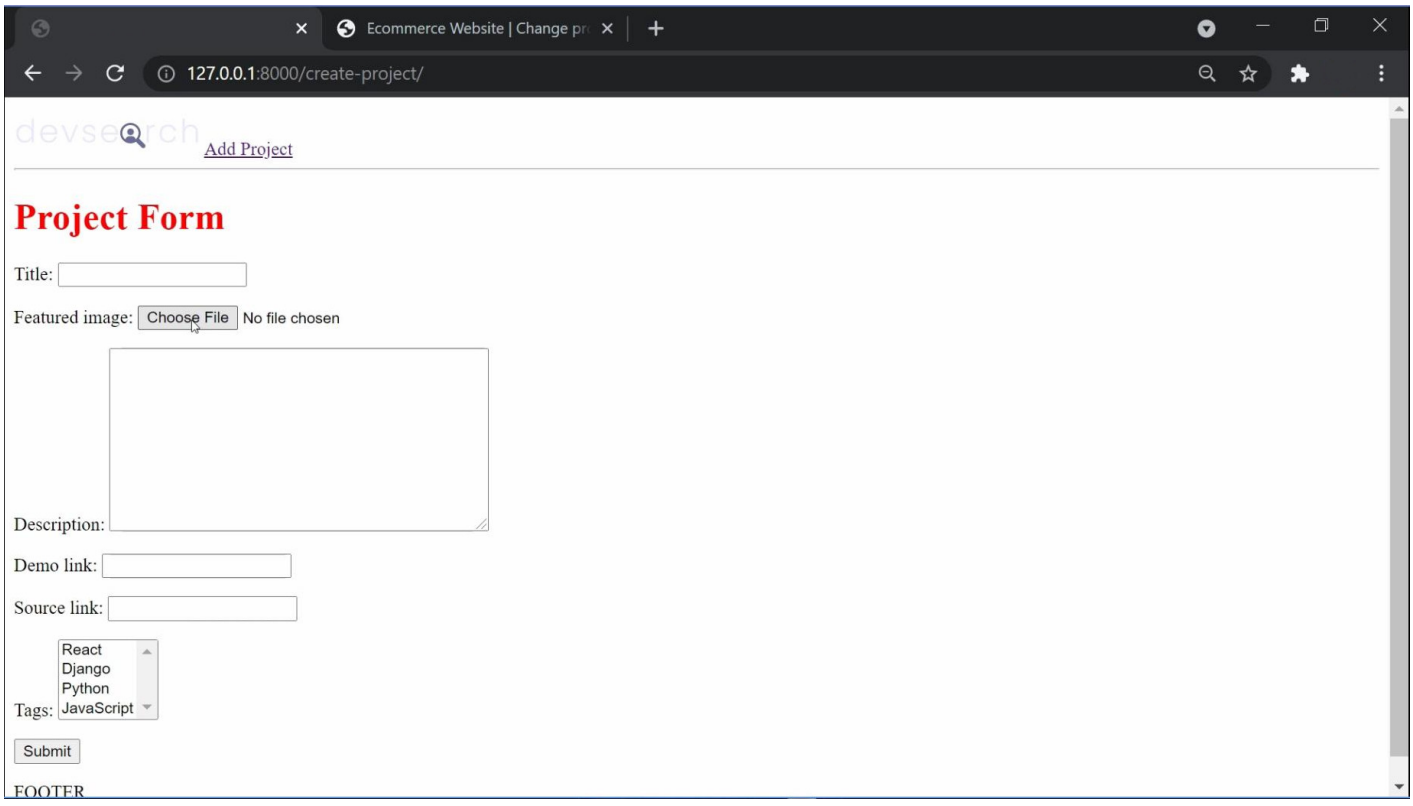
Please remember that we uploaded the image via the administration panel at 127.0.0.1/admin/ with modifications directly to the database. However, we should let any user to upload what ever image desires for her/his project. We will modify our forms in order to allow this type of CRUD operation.

Modifying the Project Form to Allow Custom Images

First, we need to modify the ***/projects/forms.py*** file to add the new field we added to upload the image ***featured_image*** in the nested **class Meta**:

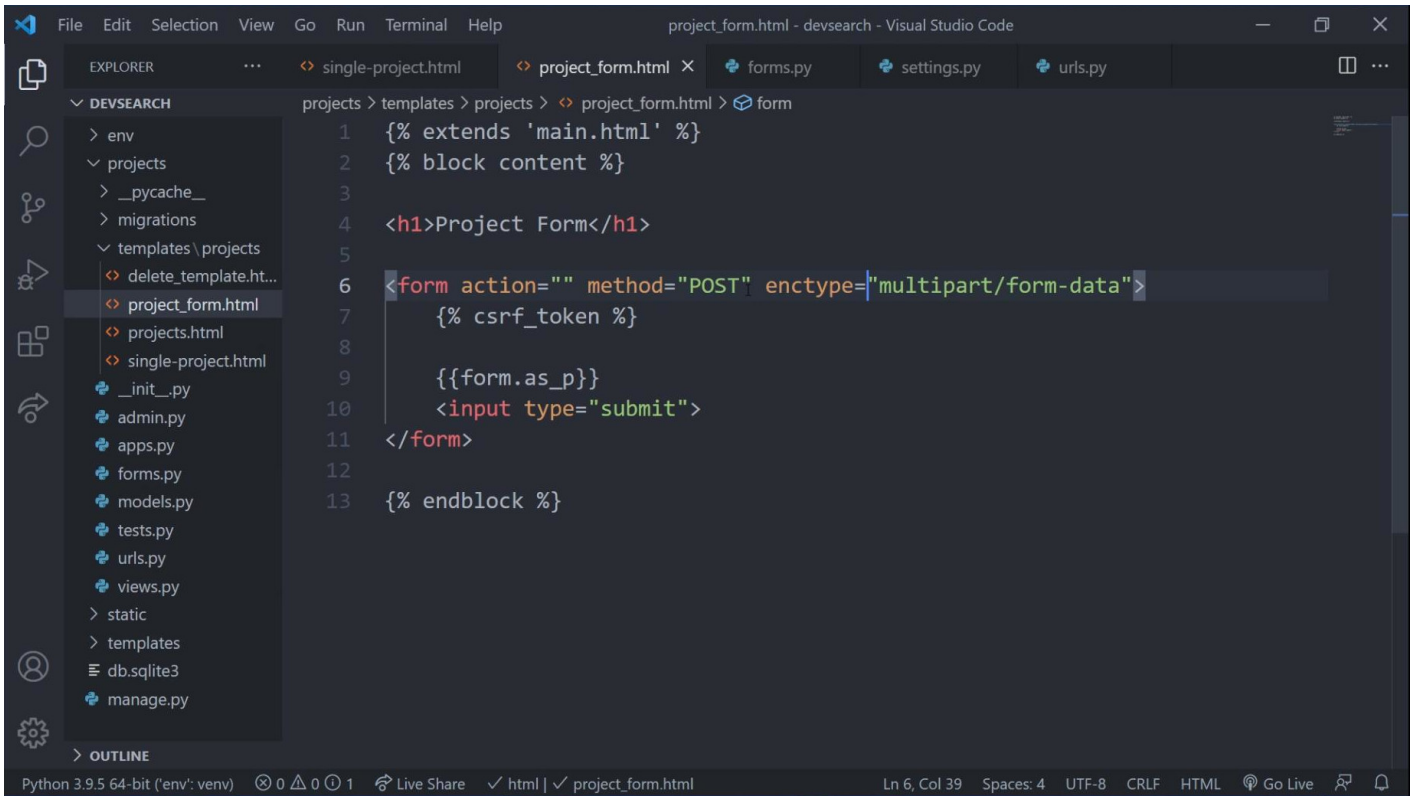


So our form now has the new field included:



Still, the form won't process the image selection process just yet. You can actually go and select an image, but it won't place it in the ***/static/image*** folder.

We need to modify the ***/projects/templates/projects/project_form.html*** template:

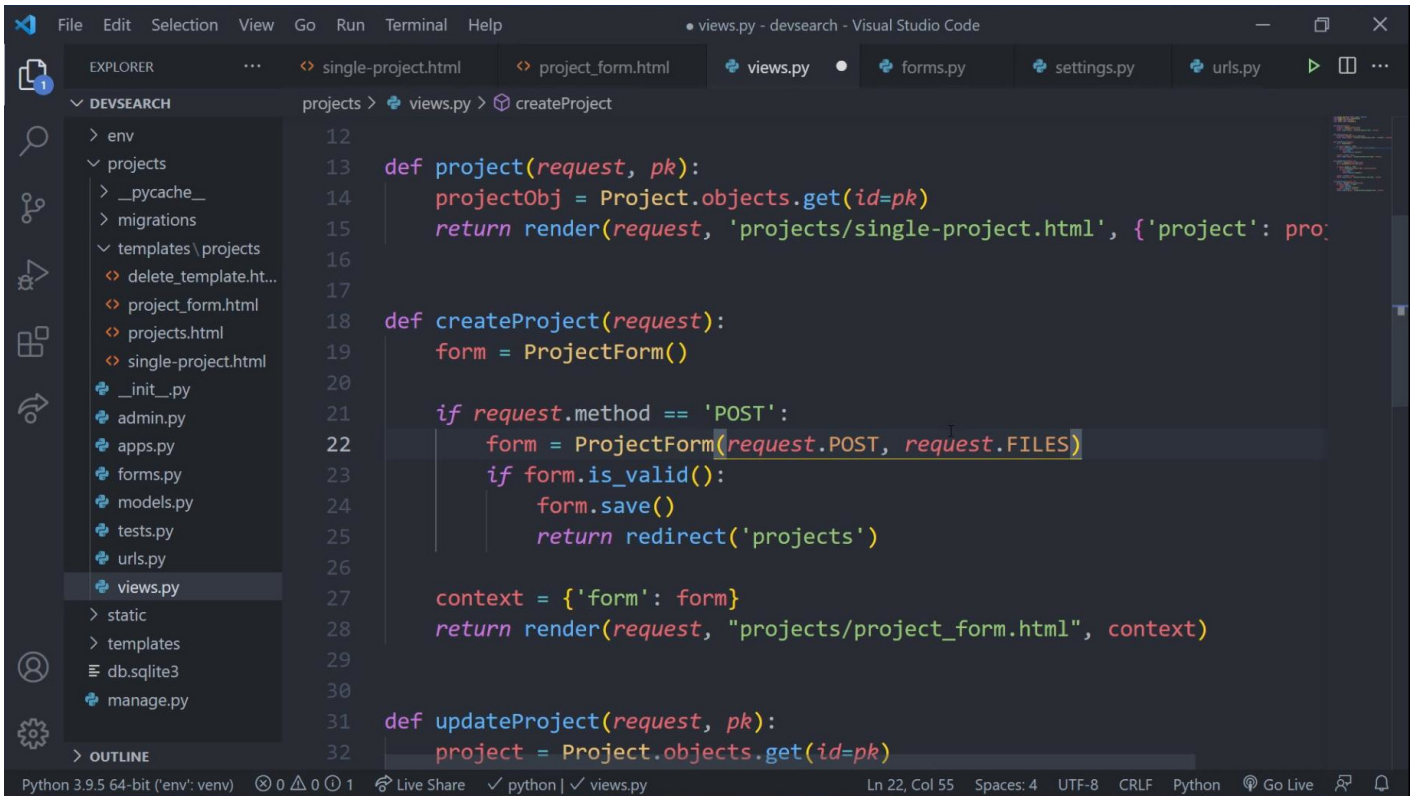


the line for form has changed to:

```
<form action="" method="POST" enctype="multipart/form-data">
```

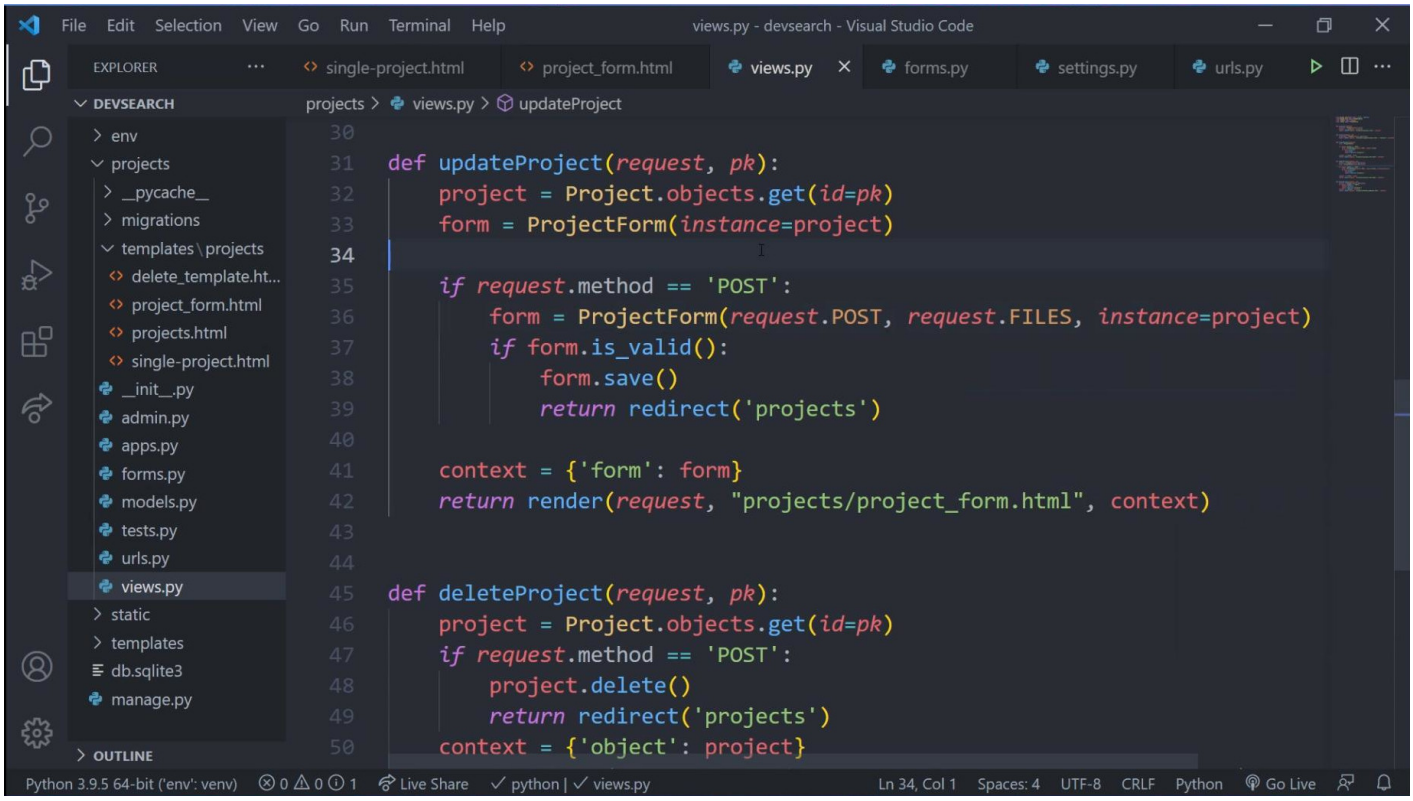
The ***multipart/form-data*** instructs the form to break up the data into specific fields that we can recover later on the ***views.py*** file.

Next, in the ***/projects/views.py***, we need to modify our createProject and updateProject classes:



and see that we added a new argument to the createProject form when a "POST" request is made:

```
form = ProjectForm(request.POST, request.FILES)
```

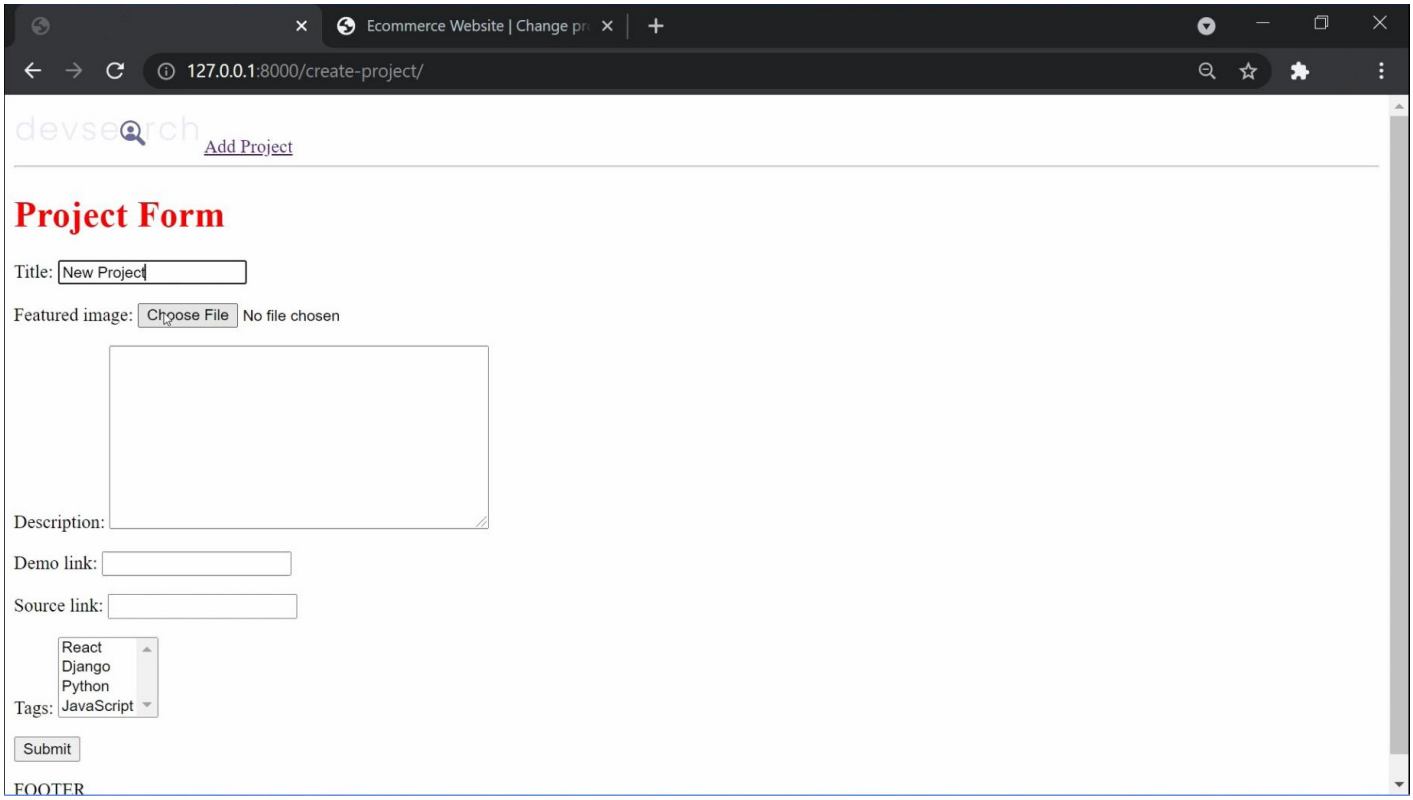


and the modification is:

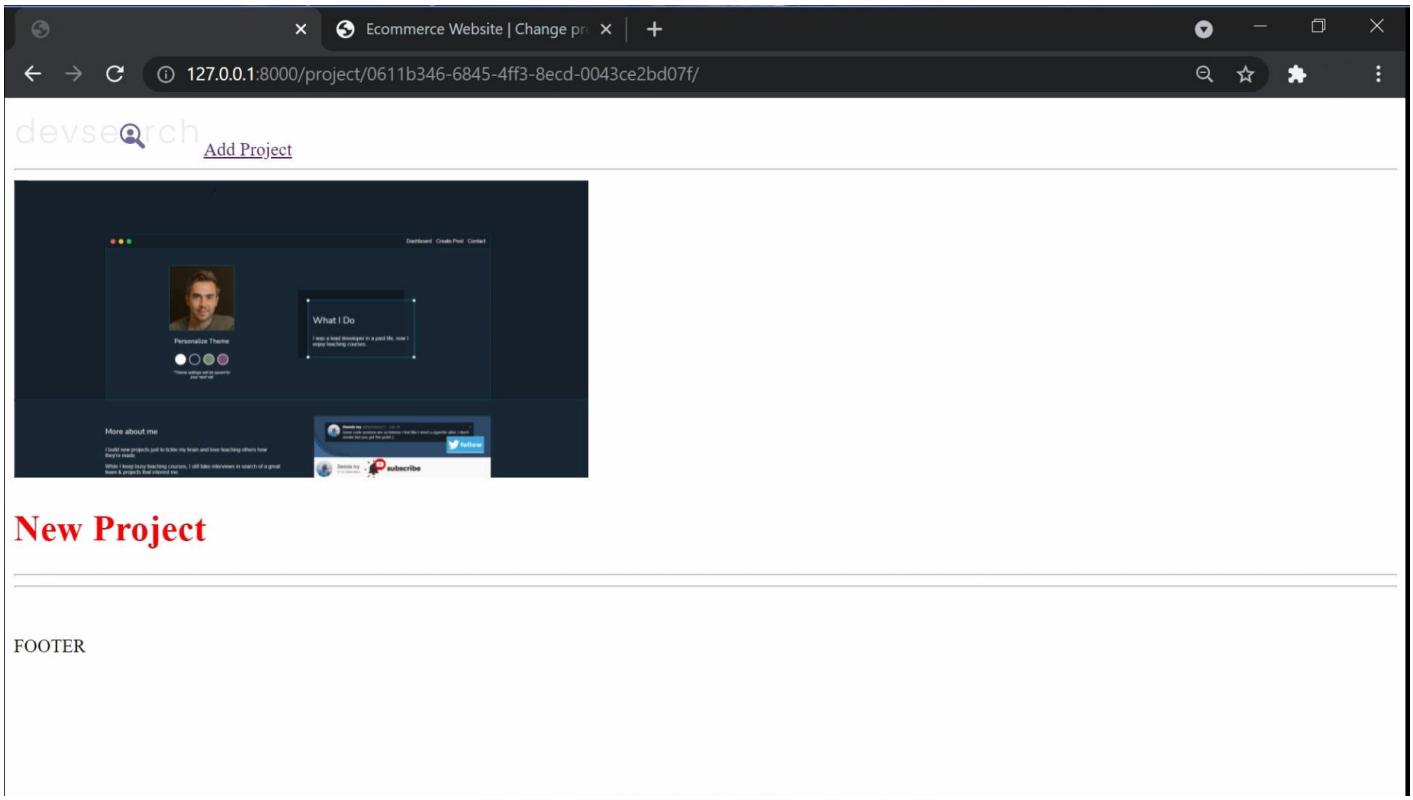
```
form = ProjectForm(request.POST, request.FILES, instance=project)
```

And that should be all the necessary steps to include images in each new projects and update images to existing projects.

Go ahead and test the form by creating a new project:



and submit a new image in the form:



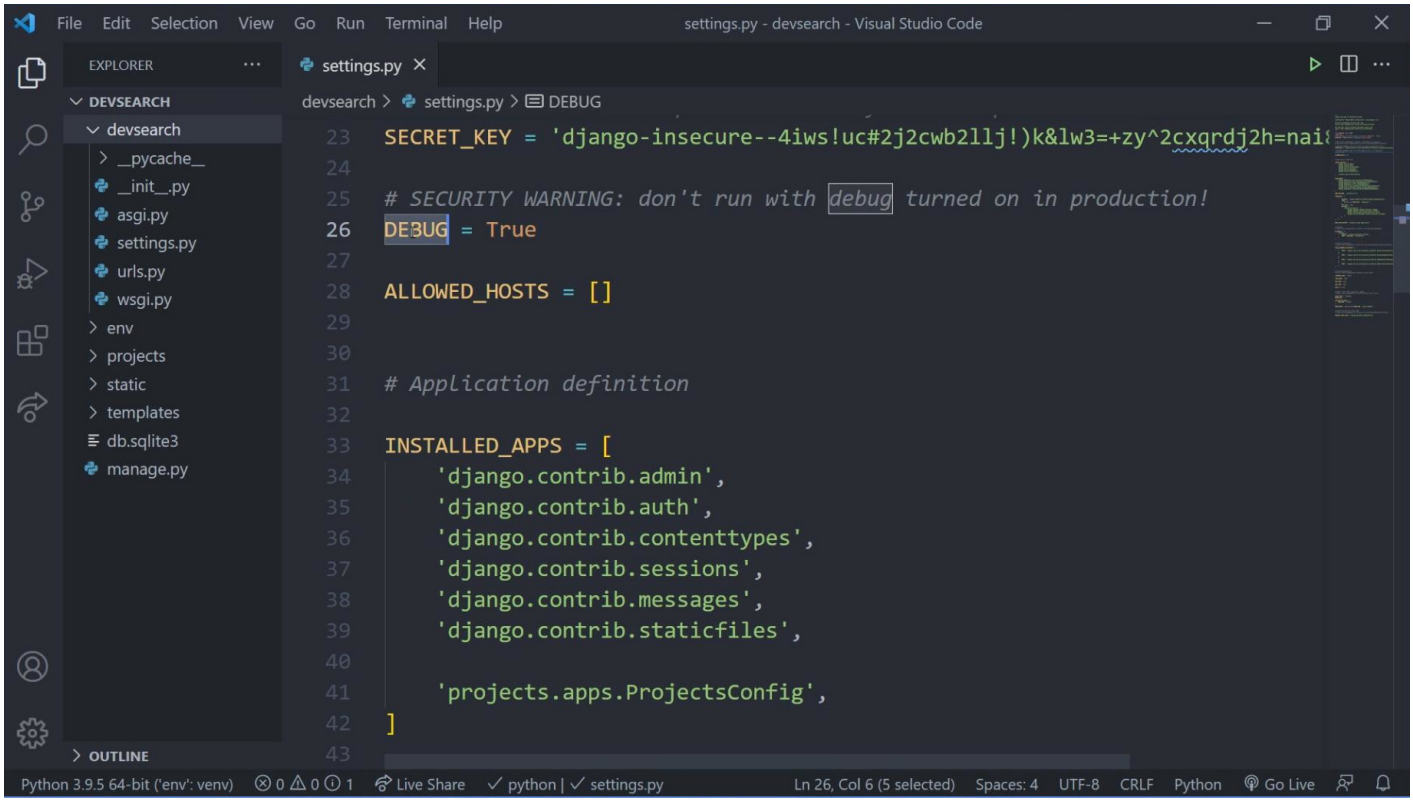
And finally, try to update the images for existing projects in your projects page. **Go ahead and test it.**

This should be all about setting images on individual forms. We acomplished the following:

1. Set up an image or icon on the navbar using static files.
2. Set up dynamic images for individual project and their pages.
3. Modified the forms in order to enter images and store them in specific folders. You can set up any folder to do this, it not necessarily done using the /static/images folder. You could use a folder inside the application folder, something such as ***/projects/images*** and you would have to modify the **MEDIA_ROOT** and **MEDIA_URL** constants in the ***/devsite/settings.py*** file.

About Static Files in Production

All what we've done up to here has been done with the DEBUG constant (flag) set to True. That is, we've been working on debug mode. In the ***/devsite/settings.py*** file:



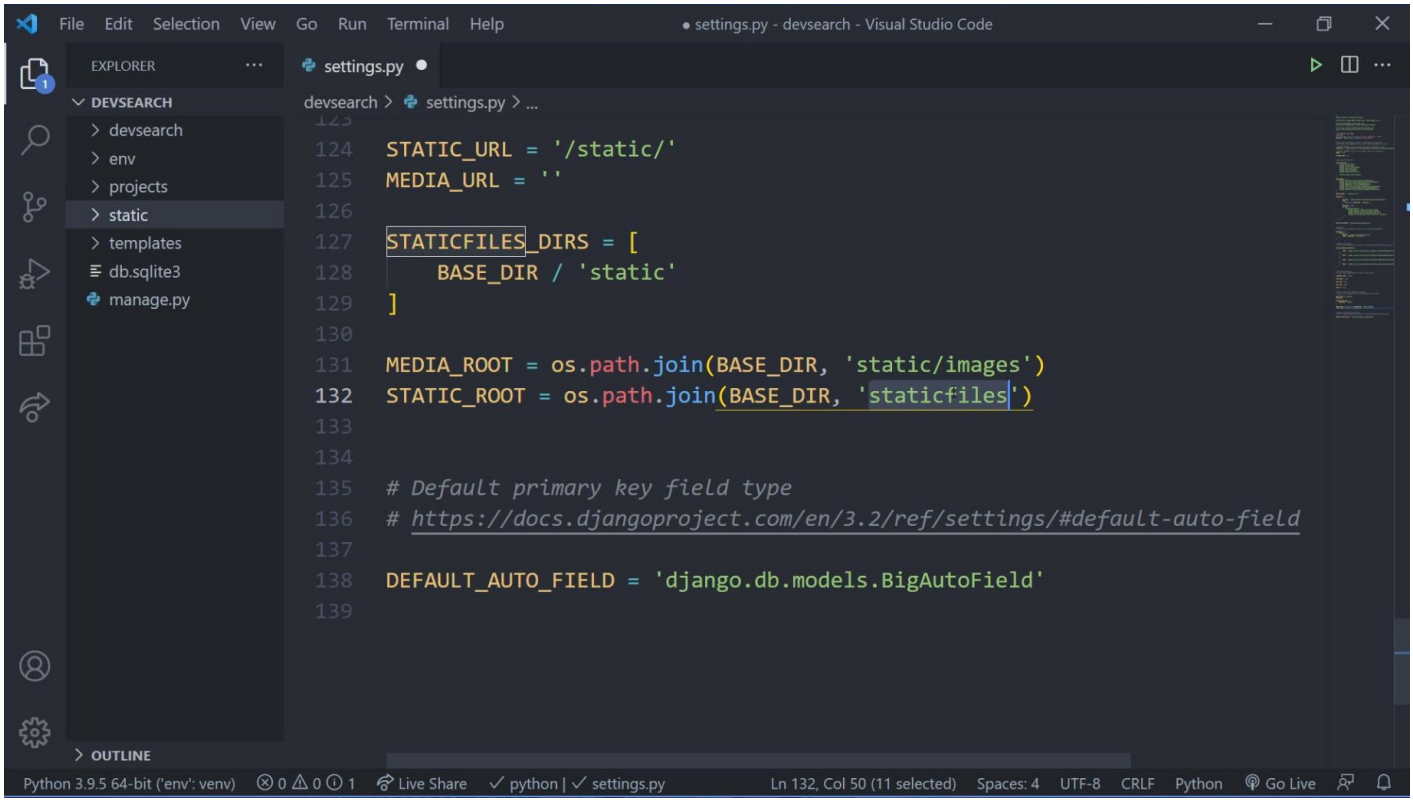
However, once we migrate our site onto Production mode (DEBUG = False), the /static/ folder **WON'T BE AVAILABLE**, i. e., all the styling files (CSS, JavaScript, JAVA, etc) won't be accessible for the Production site.

How Do We Run The Production Site Then?

Very much like when we migrate our databases, we have to migrate all our static files into a new folder, basically, a copy of the contents of the static folder we've been using during development. We could be tempted to copy it manually, and we could, but Django can do that for us instead.

Procedure

1. Set up a constant flag named ***STATIC_ROOT*** in the ***/devsite/setting.py*** file:

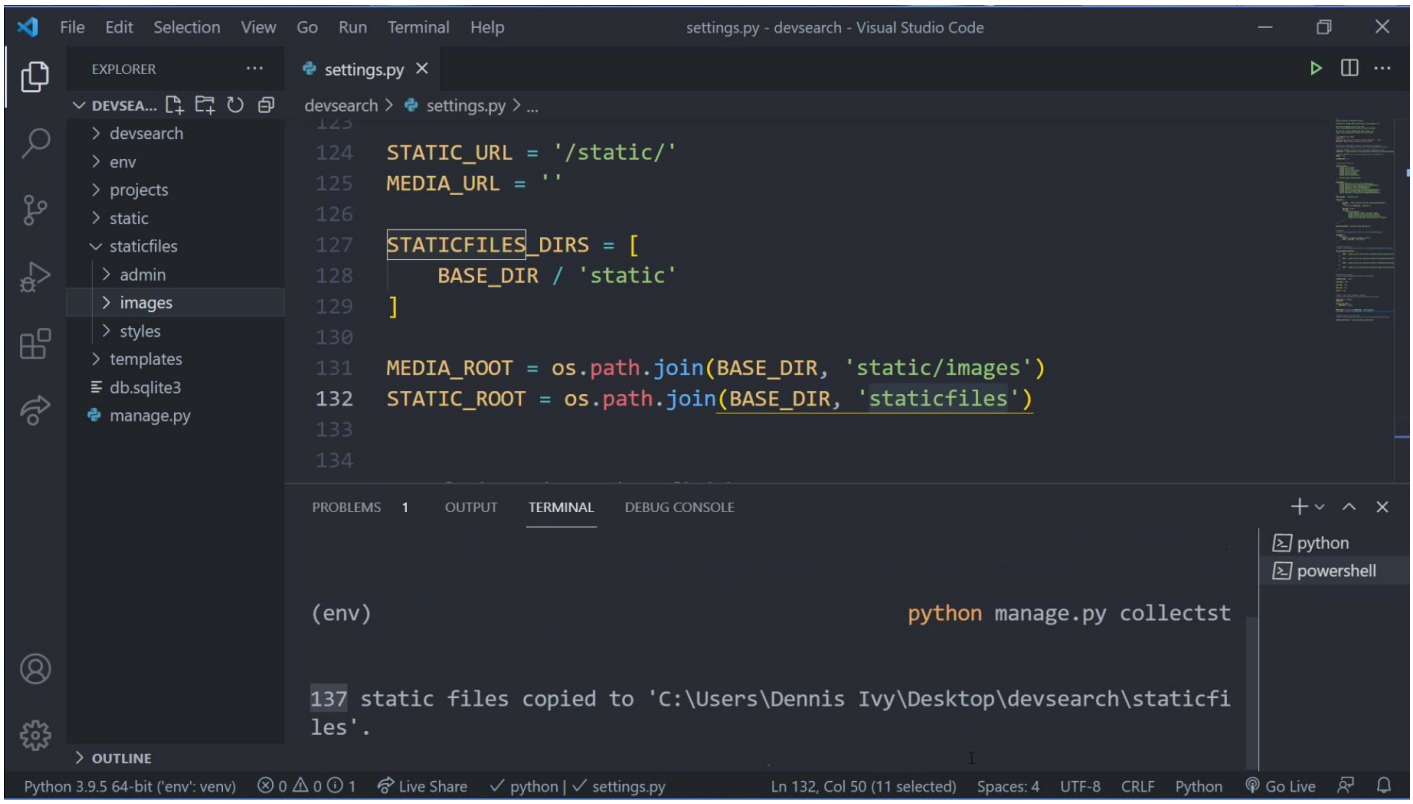


which is pretty similar to **MEDIA_ROOT**. Here we chose ***'staticfiles'*** as the name of the static folder for the site in production mode. It can be any other given name than ***'staticfiles'***, it doesn't really matter what name we assign to it, but preferably it has to be something related to it.

2. Open the terminal window in your IDE and run the command:

```
python manage.py collectstatic
```

And as you can see, it's a very similar procedure to migrate your model modifications to the database tables.



You can see that the staticfiles folder is created automatically and all the contents from the static folder are bundled up together and copied to the new folder. There's also a new addition with the admin folder. It contains all the styling files for the admin panel.

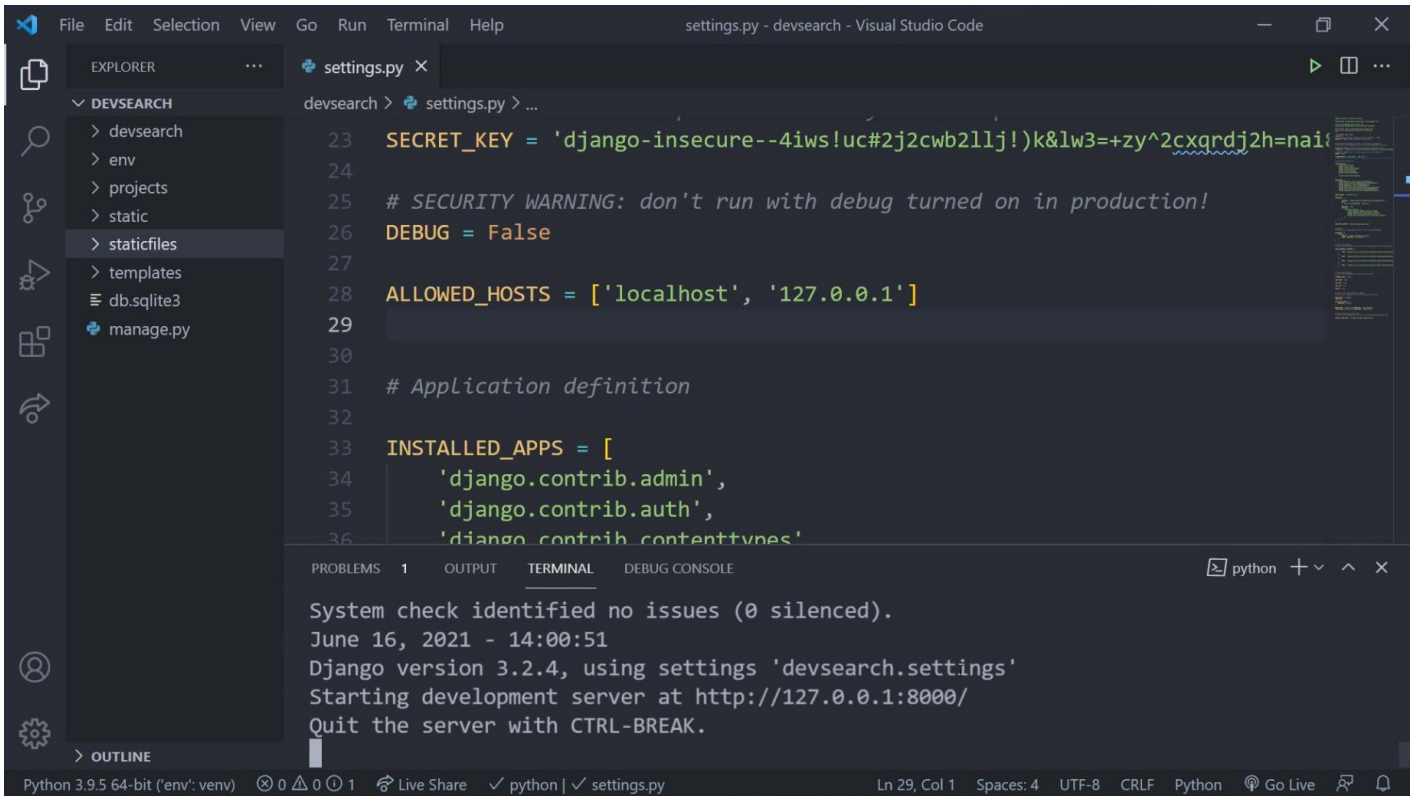
If you have to do modifications to the contents of the static folder, then, like migrating any modification to the database, you'll have to migrate the contents of the static folder to the staticfiles folder for the Production site.

What's Running The Site in Production Mode?

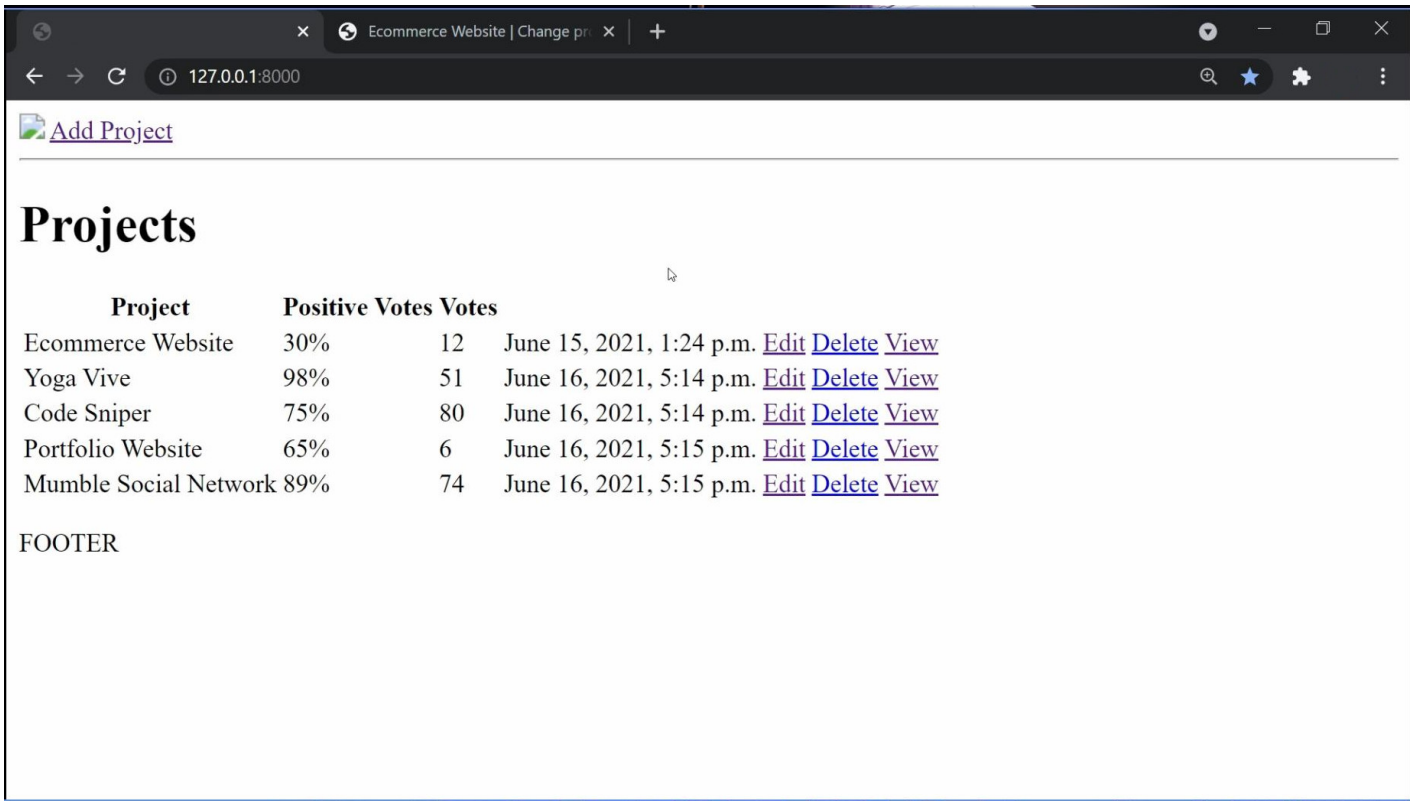
The debug mode provide us with sensitive information about possible errors during development such as if we tipe a route that doesn't exist, then the site running Django will show us the documented errors produced while trying to find the non existing page.

Production mode (DEBUG = False) cancels all warning and error messages.

To run in Production mode we have to add a list of ***ALLOWED_HOSTS*** to the setting.py file:

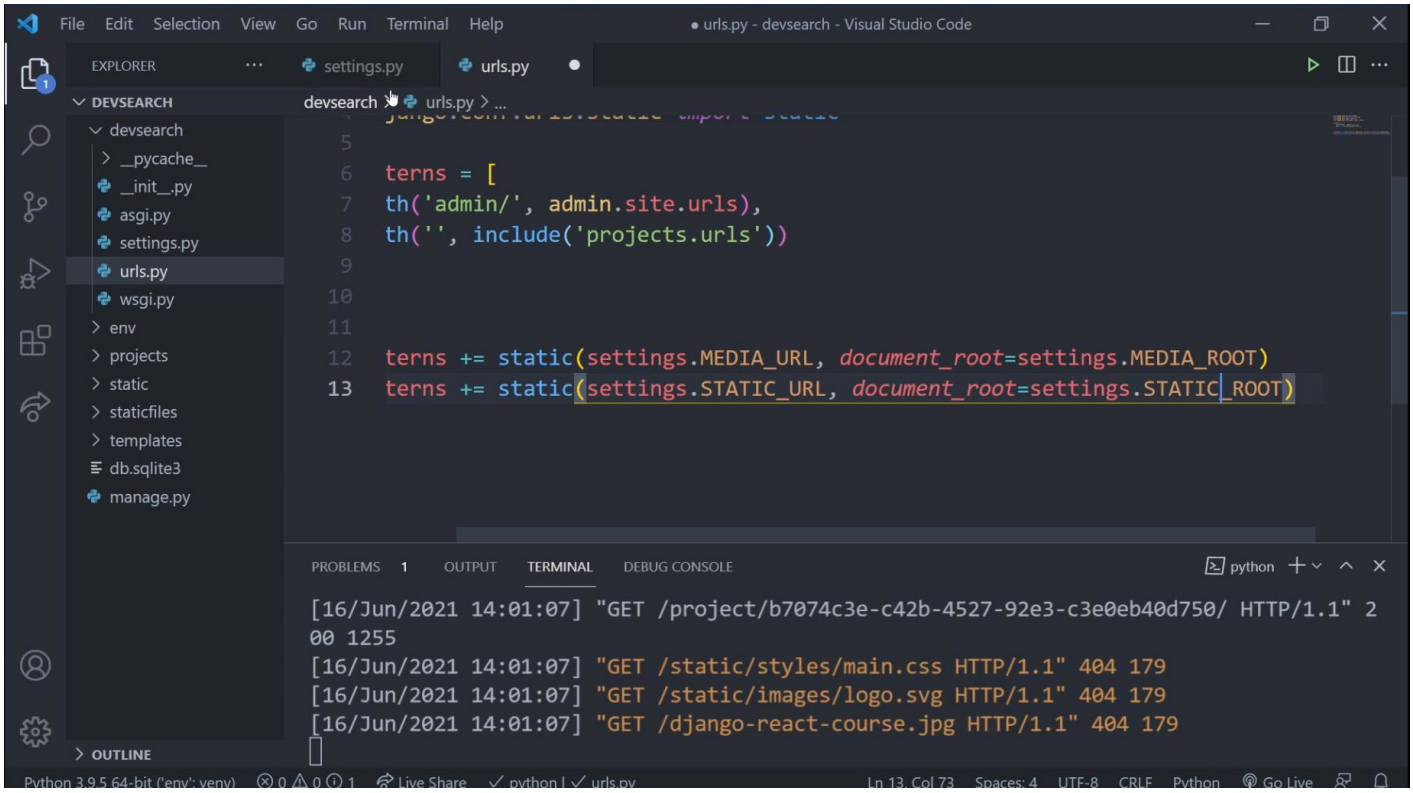


If you try to run the site right now you would find that all your static images and styling can not be seen:



The reason being that we need to include a url path to the ***staticfiles*** folder and add a middleware package that serves the static files, but not the user uploaded content.

The modifications to the urls:



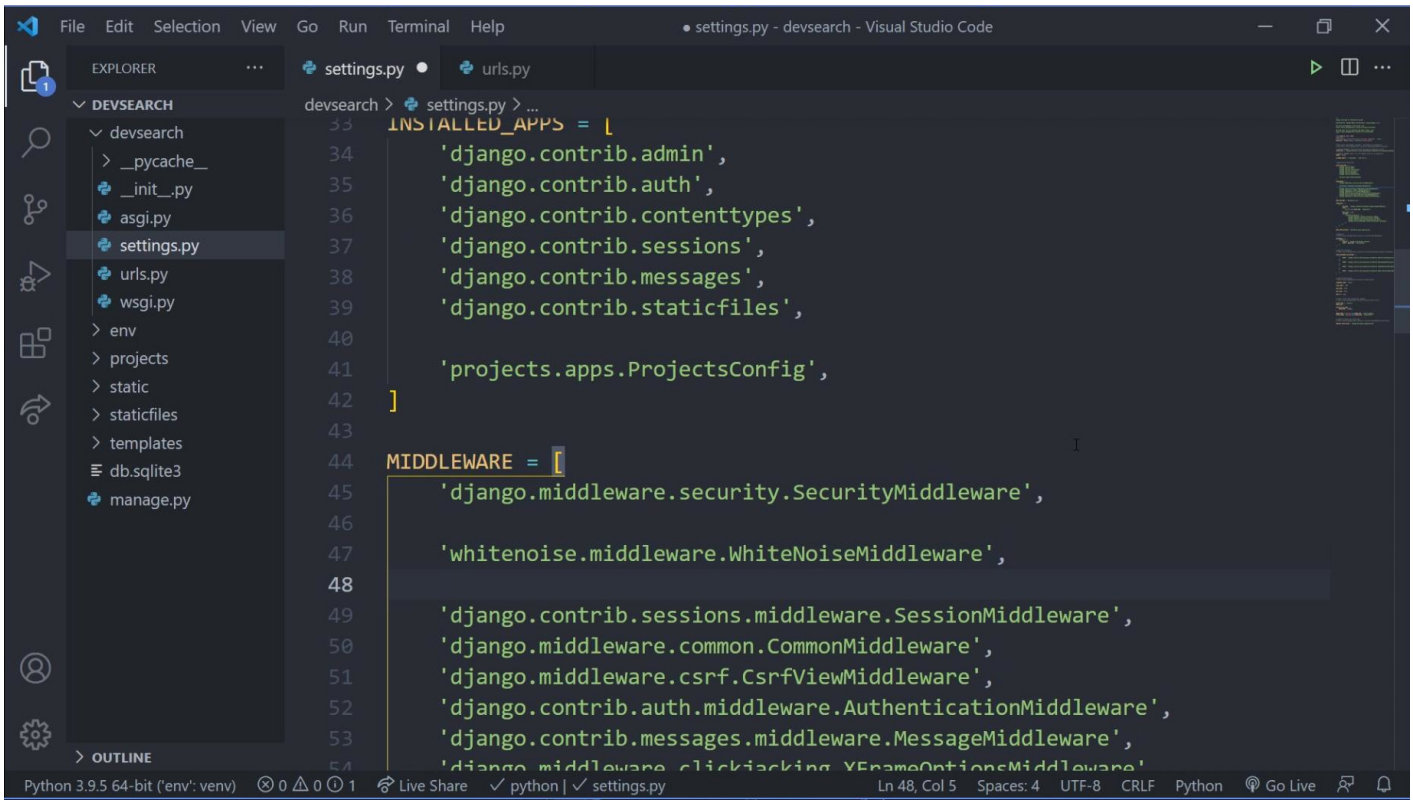
where we added the line:

```
urlpatterns += static(settings.STATIC_URL,
document_root=settings.STATIC_ROOT)
```

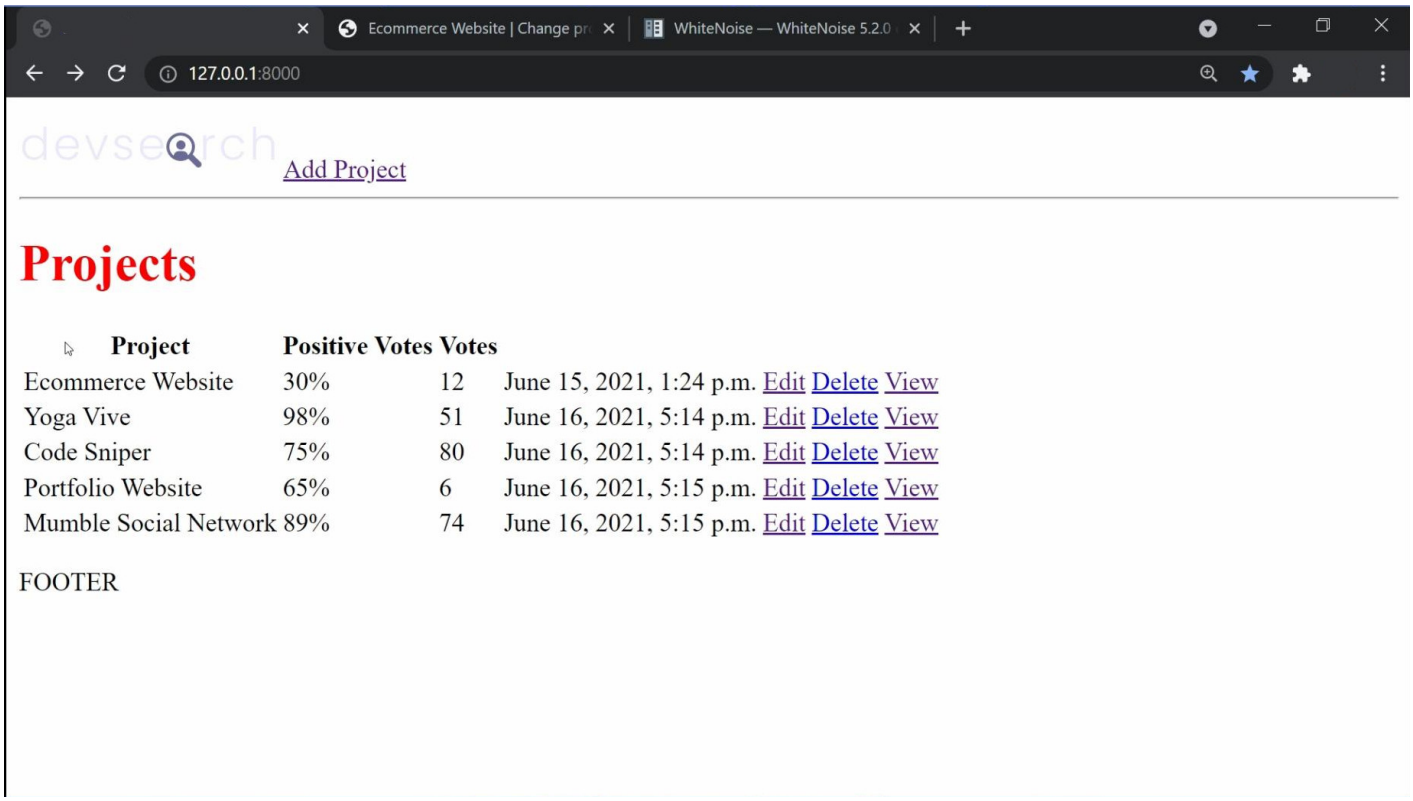
The middle ware is "whitenoise" and we have to install it in our virtual environment.

```
python -m pip install whitenoise
```

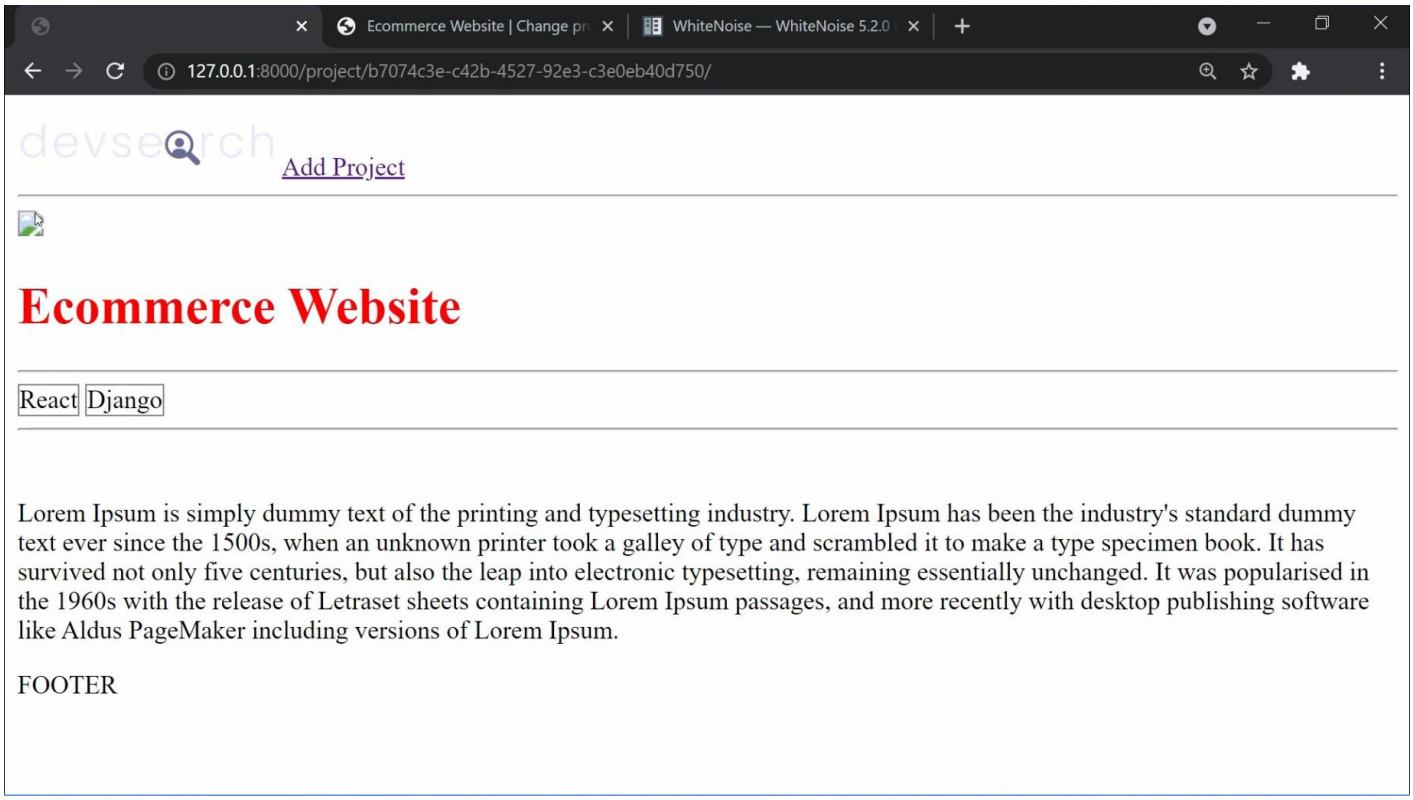
and then do some modifications to the MIDDLEWARE list in the settings.py file.



and we should be able to see:



but we won't be able to see the user uploaded content:



This is an issue that will be solved later on.