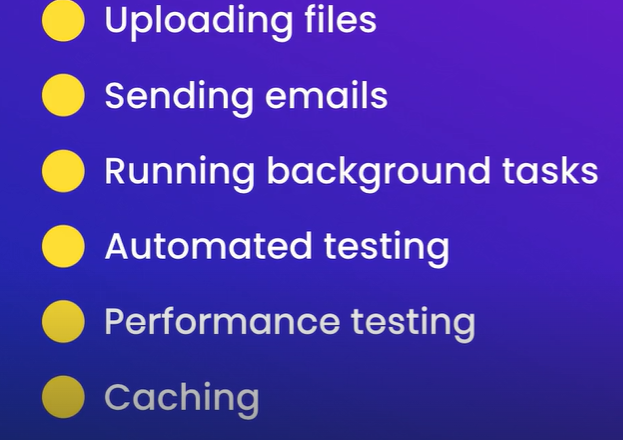


**Welcome**:

In the first two parts of the series, we build a complete RESTful API for an online store.

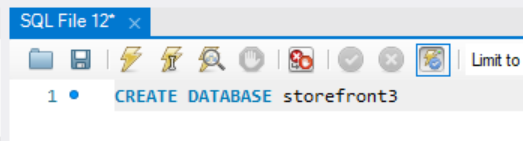
In this part we will continue on some advanced concepts like,



So by the end of this course we will able to build production grade Django applications.

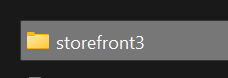
**Setting up the Project**:

Open up MySQL and create a new database called storefront3.



So we are going to create a separate database for this part of the course.

Now below the video we have some source code in a folder called storefront3.



Open this folder with VS code and first thing we need to do is change the database password.

DATABASES = {

    'default': {

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

        'NAME': 'storefront3',

        'HOST': 'localhost',

        'USER': 'root',

        'PASSWORD': 'MyPassword' 🡪C*hange this password to login into SQL*

    }

}

Now we need to install our project dependencies, by using

pipenv install

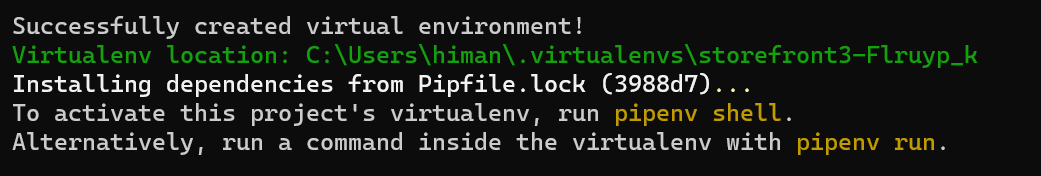
**Side note**: I changed the python version to 3.11 in pipfile, previously it was 3.9 so I was seeing an error while running pipenv install command.

[dev-packages]

autopep8 = "\*"

[requires]

python\_version = "3.11" 🡪 *Here*



Our project dependencies are now installed, now we need to activate our virtual environment.

And for that we need to run pipenv shell.

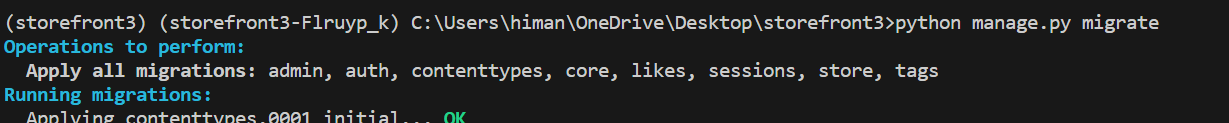


Now open this in VS code using code . and once inside *ctrl + shift + P* and python (*select interpreter*).

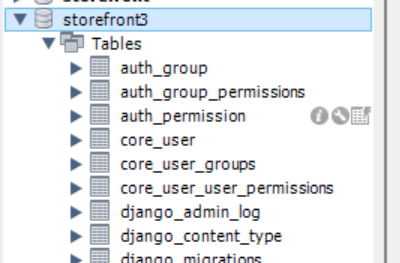
C:\Users\himan\.virtualenvs\storefront3-Flruyp\_k\Scripts\python 🡨 This is the interpreter path enter it.

Next step is to run migrations,

python manage.py migrate



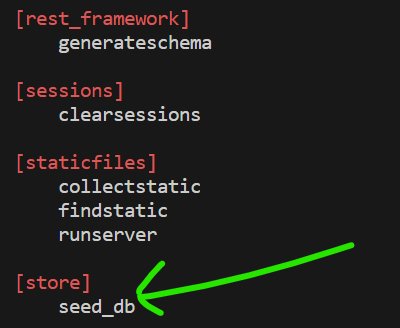
Now all our database tables are created.



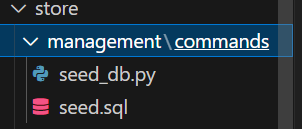
So its time to populate these tables.

Previously we used a seed.sql file to populate our database, this time we are going to use a different approach (*a custom command to populate our database*).

If we just run python manage.py without any arguments we can see all commands that are available to us.



So here we can see that in the store app, we have a command called *seed\_db*. Let’s see its implementation,



Here in the store app we have a folder called *management* and inside it a subfolder called *commands*, this is where we add our custom commands.

*Django automatically looks for any custom commands inside this folder*.

In this folder we have a file called seed\_db.py (*indicates the name of our custom command*).

from django.core.management.base import BaseCommand

from django.db import connection

from pathlib import Path

import os

class Command(BaseCommand):

    help = 'Populates the database with collections and products'

    def handle(self, \*args, \*\*options):

        print('Populating the database...')

        current\_dir = os.path.dirname(\_\_file\_\_)

        file\_path = os.path.join(current\_dir, 'seed.sql')

        sql = Path(file\_path).read\_text()

        with connection.cursor() as cursor:

            cursor.execute(sql)

In this file, we have a class that extends *BaseCommand* which is defined in django.core.management.base.

Here we set the *help* attribute to a description and we override the *handle* method.

In the *handle* method first we print “populating the database”, then we get the current directory and using ***os****.path.join* we join it with seed.sql (*in this commands folder we have our seed file*).

Now using our custom command, we are going to execute this sql file.

So *we compute the full path to our SQL file and then using the* *Path* *class, we read the entire text in this file which gives us our sql instruction*.

file\_path = os.path.join(current\_dir, 'seed.sql') 🡪 *in these two lines*

        sql = Path(file\_path).read\_text() 🡪 *sql instruction*

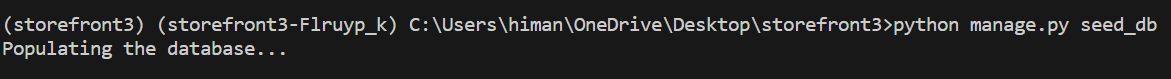
Then we open the connection to our database and execute this SQL instruction.

        with connection.cursor() as cursor: 🡪 *Here*

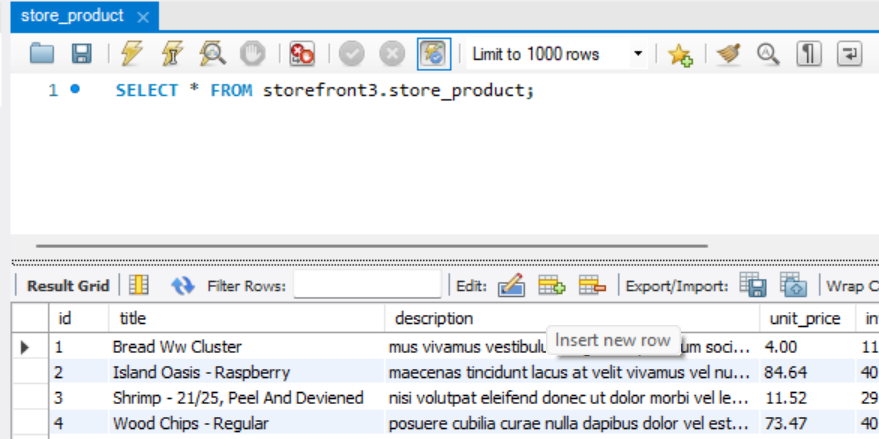
            cursor.execute(sql)

Now back to the terminal, let’s run,

python manage.py seed\_db

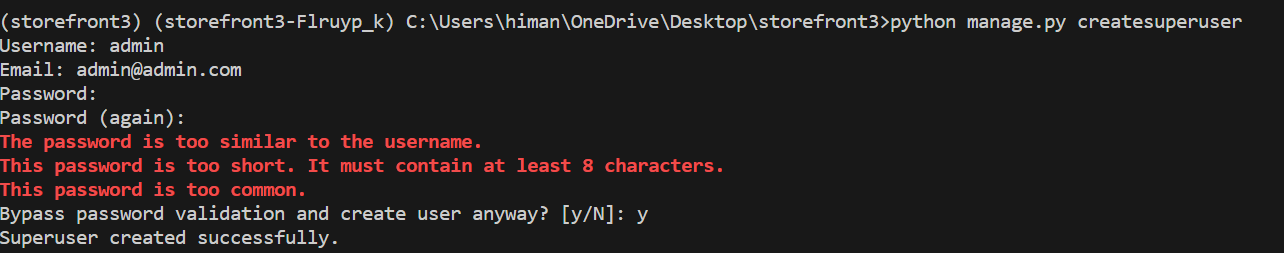


Our database is populated,



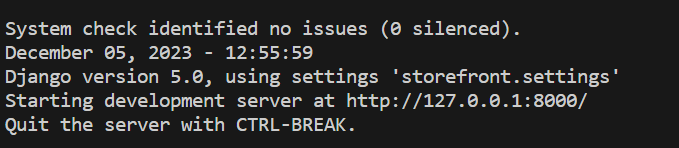
Next we need to create a superuser

python manager.py createsuperuser

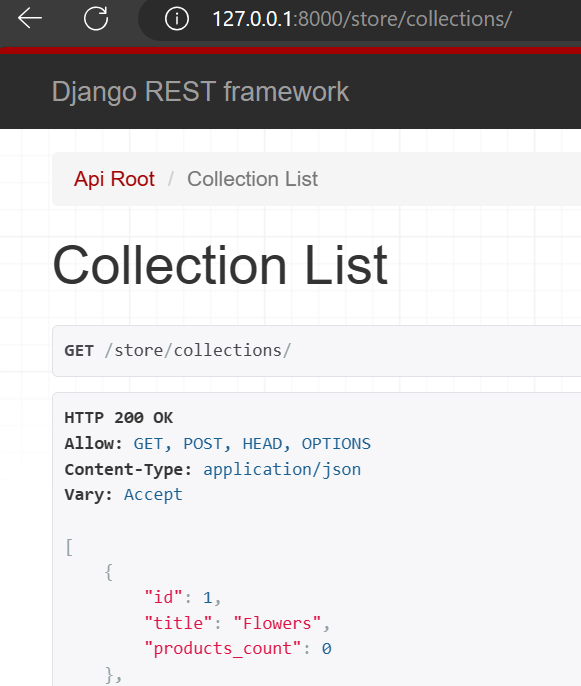


Now we have an admin user, final step is to run our webserver.

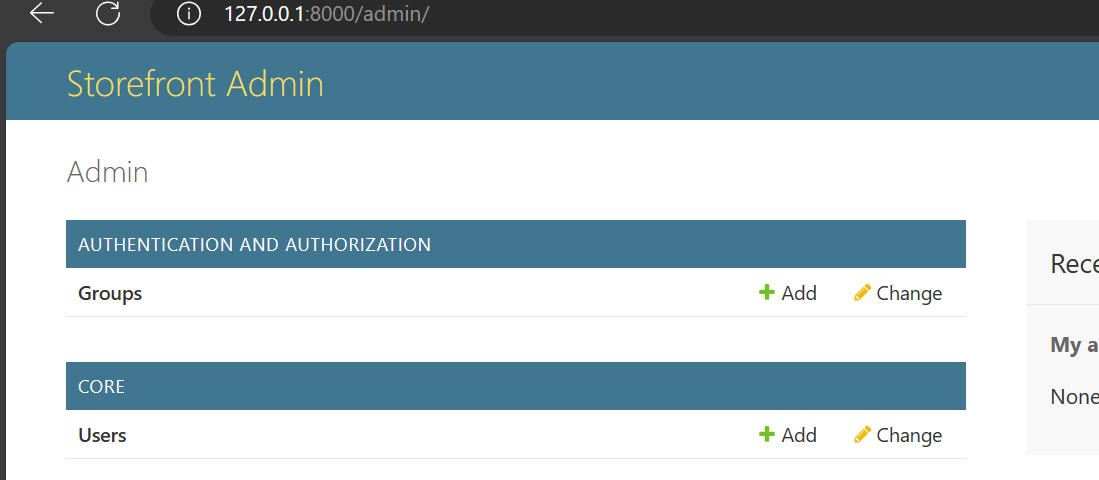
python manage.py runserver



Let’s head over to an endpoint.



And login with admin account,



Everything is working as expected.

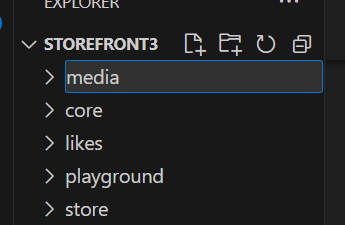
**Uploading Files**:

In this section we will talk about uploading files. So together we will build a RESTful API that client apps can call for uploading images.

**Managing Media Files**:

The first thing we need to do to implement upload feature is deciding where these user uploaded files should be stored and how we are going to serve them.

So let’s create a new folder in the root called *media*.(*according to Django media refers to user uploaded files*)



Let’s imagine all user uploaded files are going to end up in this folder. Now we need to tell Django about this folder.

So let’s go to our settings module, here we have a setting called *STATIC\_URL* 🡪*static in Django world refers to static files of our application like CSS, JavaScript, Images and so on*…

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/3.2/howto/static-files/

STATIC\_URL = "/static/"

These are the files that are bundled in our application. Media refers to the user uploaded files.

So right here (*with STATIC\_URL*) since these concepts are closely related, we should define a setting called ***MEDIA\_URL***, and *this is the endpoint at which we want to expose our media or uploaded files*.

STATIC\_URL = "/static/"

MEDIA\_URL = '/media/' 🡪 *Make sure to end up with /*

Now we should also want to *tell Django where these media files should be stored in the file system* and for that we will set ***MEDIA\_ROOT*** to the full path to the media directory.

So first on the top we need to import *os* module,

import os

from pathlib import Path

from datetime import timedelta

Then we called *os.path.join()*,

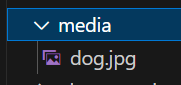
STATIC\_URL = "/static/"

MEDIA\_URL = "/media/"

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

In *os* module we have a setting for a variable called *BASE\_DIR*(*that represents the current directory*) and append it with ‘media’ (*BASE\_DIR/media*).

Now to test this, we will add an image to our media directory, *let’s imagine this image is uploaded by a user*. So let’s see if we can successfully serve it.



So we add an image(*dog.jpg*) inside media folder. Now we need to go to our URLs module and define a route for serving these media files.

Go to *urls* module of the storefront (*project*) folder and on the top from django.conf we import *settings* object and also *static* function from django.conf.urls.static.

from django.conf import settings

from django.conf.urls.static import static

Now in the urlpatterns we are going to concatenate the list with result of the static function.

urlpatterns = [

    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)),

] + static() 🡪 *Here*

We give the static function two arguments our MEDIA\_URL and MEDIA\_ROOT.

urlpatterns = [

    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)),

] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

Here *document\_root* is the kwarg in static function.

With this new line,

static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

We are telling Django that *we want to expose an endpoint that is defined in MEDIA\_URL and any requests that go to this endpoint should be routed to the file system at settings.MEDIA\_ROOT*.

This strategy is good for development but for production we will take different strategy later.

A better way to write this code is like this, instead of modifying this urlpatterns list right here, we are going to write a condition.

urlpatterns = [

    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)),

]

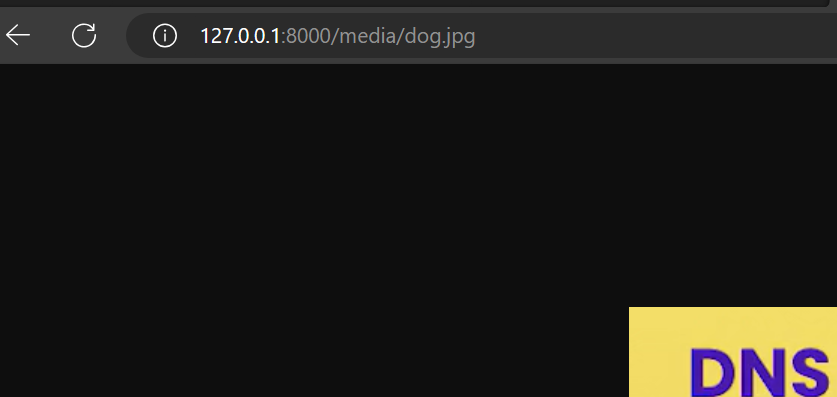
if settings.DEBUG: 🡪 *here*

    urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

If *settings.DEBUG* is true then we are going to take urlpatterns and add our new path to it

Note: In development *DEBUG* is turned on by default and in production it is turned off.

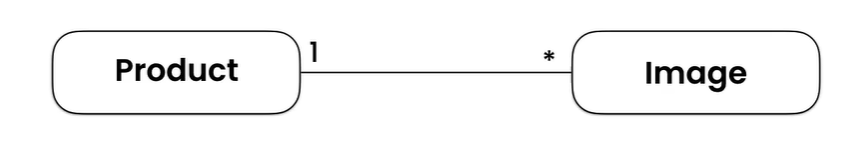
Now let’s test our implementation, at <http://127.0.0.1:8000/media/dog.jpg>



Great! we can successfully serve media files at this address.

**Adding Images to Products**:

So let’s imagine that each product can have zero or more images. So *we need to make a small change to our data model and add a* ***one to many*** *relationship between* ***Product*** *and a new model called* ***ProductImage***.



Let’s go to *models* module of the store app.

Here is our *Product* model,

class Product(models.Model):

    title = models.CharField(max\_length=255)

    slug = models.SlugField()

    description = models.TextField(null=True, blank=True)

    unit\_price = models.DecimalField(

        max\_digits=6,

        decimal\_places=2,

        validators=[MinValueValidator(1)])

    inventory = models.IntegerField(validators=[MinValueValidator(0)])

    last\_update = models.DateTimeField(auto\_now=True)

    collection = models.ForeignKey(

        Collection, on\_delete=models.PROTECT, related\_name='products')

    promotions = models.ManyToManyField(Promotion, blank=True)

    def \_\_str\_\_(self) -> str:

        return self.title

    class Meta:

        ordering = ['title']

Right after this, we will define a new class called ProductImage which will extend *models.Model*.

In this class we need two attributes. First one is product (*which is a foreign key to the Product model*) and second attribute is image (*which is an ImageField*)

class ProductImage(models.Model):

    product = models.ForeignKey(Product, on\_delete=models.CASCADE, related\_name='images')

    image = models.ImageField()

Note: Just like *ImageField*, we also have *FileField*, the difference between them is that *ImageField validates the uploaded image and ensures it’s a valid image*. It also has properties that only apply to images like width & height. So if you want to allow users to upload images, use ImageField but *for any other type of file like documents, pdfs use FileField*.

In the ImageField, we need to set *upload\_to* to a path like store/images/.

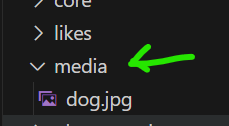
class ProductImage(models.Model):

    product = models.ForeignKey(Product, on\_delete=models.CASCADE, related\_name='images')

    image = models.ImageField(upload\_to='store/images')

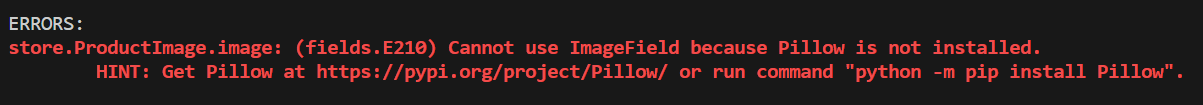
Note: With this implementation, we are not going to store images in a database because that’s going to make our database super big and our queries are going to be slow. So *for performance reasons we will store images in our file system and their path in the database*.

The path ‘store/images’ that we have specified in upload\_to is relative to our MEDIA\_ROOT. In the previous lesson we set the MEDIA\_ROOT to the media folder.

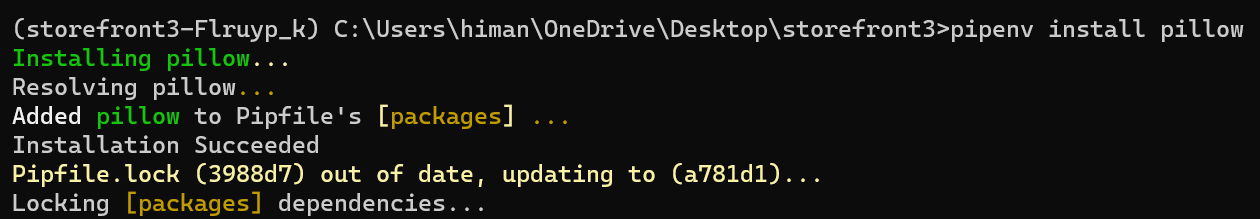


So when the user uploads an image for a product, that image is going to be stored here (*store/images*).

Since we have used an ImageField here, we need to add a library called ***pillow***.

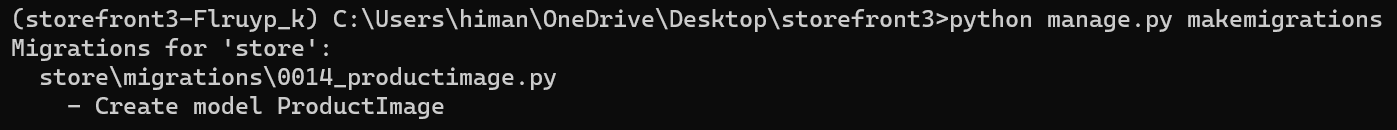


pipenv install pillow

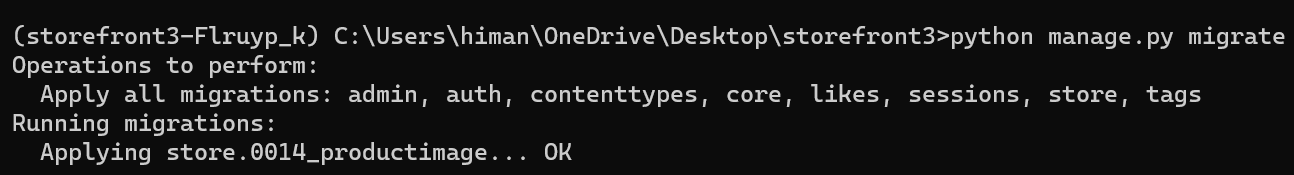


Now the final step is to create a migration and run it.

python manage.py makemigrations



python manage.py migrate



So our data model is ready, Next we are going to build an API for uploading images.

**Building and API to upload images**:

In this lesson, we are going to build an endpoint like this,

/products/1/images/1

So here we have nested resources, therefore we are going to use a nested router to implement this endpoint.

As always, for building an API we need,

🡪 Serializer

🡪 View

🡪 Router

First we go to serializers module of the store app and create a new serializer.

class ProductImageSerializer(serializers.ModelSerializer):

Here we define a Meta class and set the model to ProductImage and fields to id and image.

class ProductImageSerializer(serializers.ModelSerializer):

    class Meta:

        model = ProductImage

        fields = ["id", "image"]

We don’t want to return product id here, because its already available in the URL (/products/1/images/1) 🡨 *product\_id = 1* here.

Next we need to implement a view, so we go to the views module of the store app and define a new class called *ProductImageViewSet* which extends *ModelViewSet*.

class ProductImageViewSet(ModelViewSet):

    serializer\_class = ProductImageSerializer

    queryset = ProductImage.objects.all()

Now there is a problem here, we don’t want to return all product images in the database, we only want to return images for a particular product.

So instead of setting the *queryset* attribute, we are going to override *get\_queryset* method.

class ProductImageViewSet(ModelViewSet):

    serializer\_class = ProductImageSerializer

    def get\_queryset(self):

        return ProductImage.objects.filter(product\_id = )

Here we need to filter ProductImage based on product\_id. We get it from self.kwarg[‘product\_pk’]

Note: In the URL where we have /product/1/images/1. Our URL parameter is product\_pk and second one is pk.

/products/1(***product\_pk***)/images/1(***pk***)

So we grab the product id from the url,

class ProductImageViewSet(ModelViewSet):

    serializer\_class = ProductImageSerializer

    def get\_queryset(self):

        return ProductImage.objects.filter(product\_id = self.kwargs['product\_pk']) 🡪 *Here*

Here is our viewset, next we define our route.

So we go to the urls module of the store app.

products\_router = routers.NestedDefaultRouter(

    router, 'products', lookup='product')

products\_router.register('reviews', views.ReviewViewSet,

                         basename='product-reviews')

Here we have a router for our products endpoint. Earlier we registered a nested endpoint inside products called *reviews*.

So Let’s register the new endpoint,

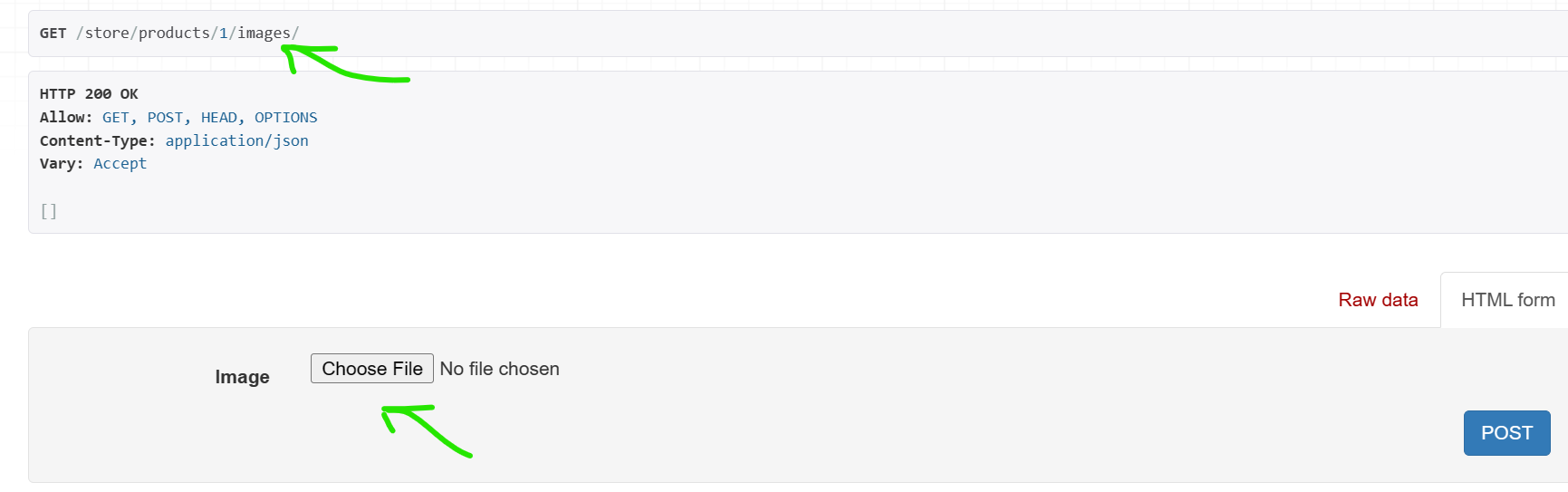
Since we have overridden the get\_queryset method, here we need to set the basename (product-images).

products\_router = routers.NestedDefaultRouter(router, "products", lookup="product")

products\_router.register("reviews", views.ReviewViewSet, basename="product-reviews")

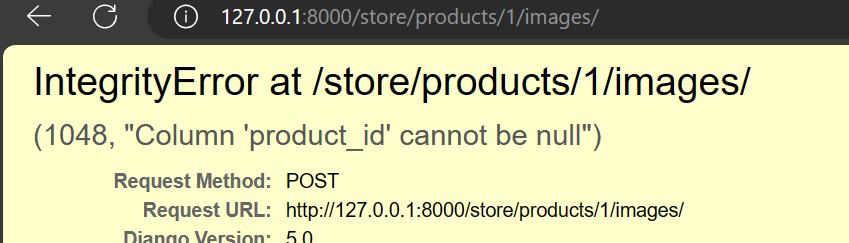
products\_router.register("images", views.ProductImageViewSet, basename="product-images") 🡪 *Here*

Let’s go to products/1/images/ endpoint,



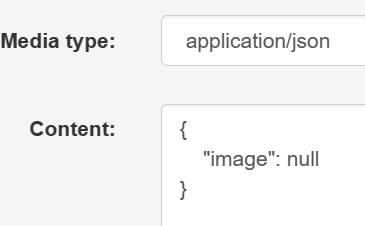
Here in this HTML form we can upload an image. Let’s upload the image.

As soon as we hit POST, we get an IntegrityError.



“Column” product\_id cannot be null.

Its because in this form we are only supplying the image not the product\_id. Look at the raw data.



In this JSON object, we only have the image property, not the product\_id. But we don’t want to include product\_id in this object because its already available in the URL.

So when creating a ProductImage object, we should extract product\_id from URL and use it to save the ProductImage.

In the ProductImageViewSet we have access to our URL parameters, just like we used product\_pk to filter images. So *we are going to extract* ***product\_pk*** *one more time and using a* ***context object*** *pass that to our serializer then in the serializer we are going to grab that from the context and use it to create a ProductImage object*.

class ProductImageViewSet(ModelViewSet):

    serializer\_class = ProductImageSerializer

    def get\_serializer\_context(self): 🡪 *Here*

        return {"product\_id": self.kwargs["product\_pk"]}

    def get\_queryset(self):

        return ProductImage.objects.filter(product\_id=self.kwargs["product\_pk"])

Now let’s go to our serializer and this class we should override the create method and set the product\_id to self.context(‘product\_id’)

class ProductImageSerializer(serializers.ModelSerializer):

    def create(self, validated\_data):

        product\_id = self.context["product\_id"] 🡪 *get product\_id from context*

        return *dictionary*

Then instead of relying on default implementation, we are going to explicitly create ProductImage object. First we set product\_id and then we pass all the validated data.

class ProductImageSerializer(serializers.ModelSerializer):

    def create(self, validated\_data):

        product\_id = self.context["product\_id"]

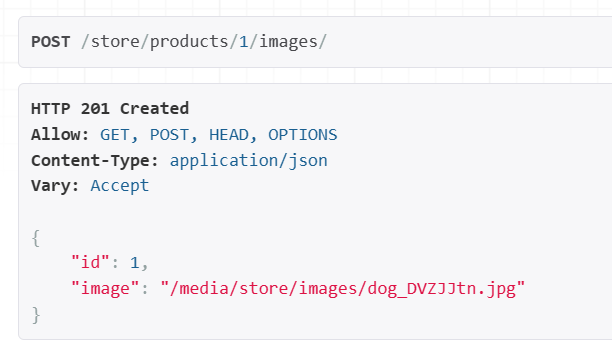
        return ProductImage.objects.create(product\_id=product\_id, \*\*validated\_data) 🡪 *Here*

    class Meta:

        model = ProductImage

        fields = ["id", "image"]

Let’s test our implementation again. By uploading same image.



Great! Now we have this image stored at this address and id of this image is 1.

Note: Django automatically appended some random letters to the image name (*dog\_DVZJJtn.jpg*), if we upload the same image twice to avoid overwriting an existing file.

**Returning Images from the API**:

So we implemented the uploading feature. Now let’s say the client wants to render the list of products by hitting /products endpoint.



Currently we don’t have the images here and we don’t want client to hit separate endpoints for getting the images of each product, that’s way to expensive and slow.

So when returning the list of products, we want to return the images as well, therefore we go to our *ProductSerializer* class.

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = [

            "id",

            "title",

            "description",

            "slug",

            "inventory",

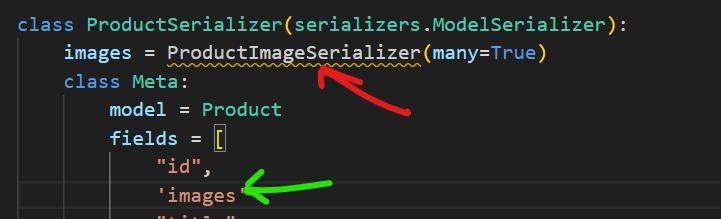
            "unit\_price",

            "price\_with\_tax",

            "collection",

        ]

Here are the fields we are currently returning. We will add a new field called ‘images’ and define it like this.



Note: We are getting an error here because ProductImageSerializer is defined after ProductSerializer, so just cut the code from there and put it before.

We set *many = true* but we also want to set *read\_only* to true because *when creating the product we don’t want to pass multiple images, we only want to pass properties related to a product object*.

class ProductSerializer(serializers.ModelSerializer):

    images = ProductImageSerializer(many=True, read\_only=True)

    class Meta:

        model = Product

        fields = [

            "id",

            "images",

            "title",

            "description",

            "slug",

            "inventory",

            "unit\_price",

            "price\_with\_tax",

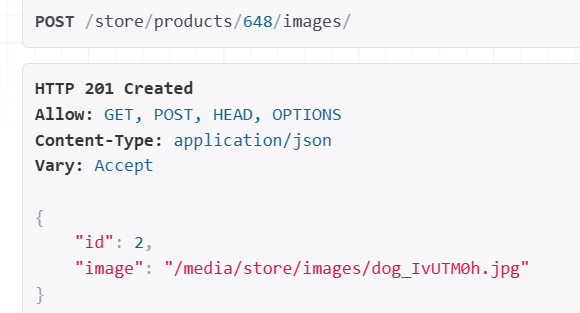
            "collection",

        ]

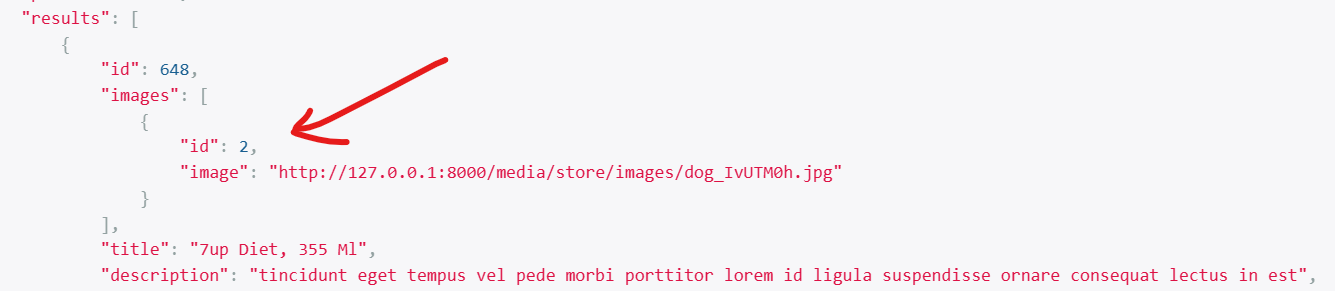
Now each product has images property,



Let’s add an image in product\_id 648 to test this,

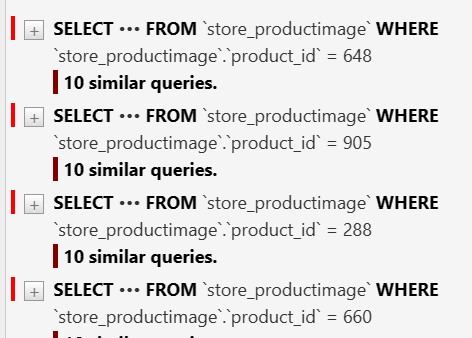


If we go to product endpoint again,



This product has one image with this ID, and if we click it our image opens.

There is one issue here, if we open Django debug toolbar.



We can see multiple duplicate queries because *for each product Django is going to the database to fetch its images so we need to eager load our product with their images*.

To do this go to ProductViewSet from where we are fetching our products.

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.prefetch\_related("images").all()

    serializer\_class = ProductSerializer

Here we call *prefetch\_related*(‘images’)

Our duplicate queries are removed now.

**Validating Uploaded Files**:

In the ProductImage class, look at the ImageField.

class ProductImage(models.Model):

    product = models.ForeignKey(

        Product, on\_delete=models.CASCADE, related\_name="images"

    )

    image = models.ImageField(upload\_to="store/images")

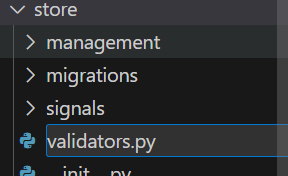
Earlier we learned that this *ImageField* validates the uploaded file and raises an error, if it’s not a valid image.

So if we try to upload a document like pdf, we are going to get an error and that’s why we had to install *pillow*, because *pillow is an image processing library for python*.

So under the hood this ImageField class uses pillow for validating the incoming image.

But what about the file size? currently there is no built in validation for that. So we need to create it from scratch.

In the store app, add a new file called *validators*.py



Here we are going to define the validator function.

We can call it *validate\_file\_size* and give it *file* as parameter.

def validate\_file\_size(file):

Here we can define a variable called *max\_size\_kb* and set it to 50 with below condition.

from django.core.exceptions import ValidationError

def validate\_file\_size(file):

    max\_size\_kb = 50

    if file.size > max\_size\_kb \* 1024:

        raise ValidationError(f"Files cannot be larger then {max\_size\_kb}KB!")

Now we have a validator function, go to our ProductImage model and set a keyword argument called *validators*(*a list of one or more validators*) inside *ImageField*.

class ProductImage(models.Model):

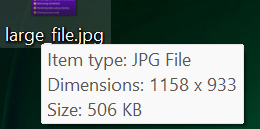
    product = models.ForeignKey(

        Product, on\_delete=models.CASCADE, related\_name="images"

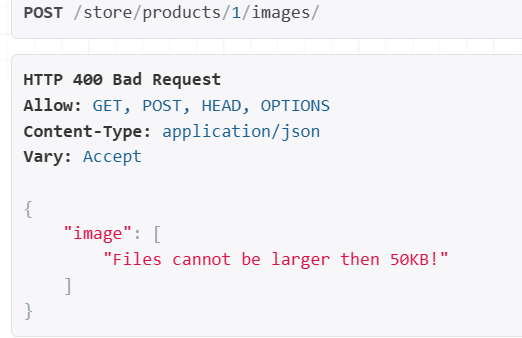
    )

    image = models.ImageField(upload\_to="store/images", validators=[validate\_file\_size]) 🡪 *here*

To test if this validation is working, we need to upload a file size greater than 50KB.



We get a validation error,



In Django we can also validate the file extension and that is useful or essential specially when we are using *FileField* (*might want to restrict client to upload pdfs only*), ImageField does that under the hood so need to customize there.

This is how we restrict file extension. In django.core.validators we have a *FileExtensionValidator*.

class ProductImage(models.Model):

    product = models.ForeignKey(

        Product, on\_delete=models.CASCADE, related\_name="images"

    )

    image = models.FileField( 🡪 *Change here to FileField*

        upload\_to="store/images",

        validators=[FileExtensionValidator(allowed\_extensions=["pdf"])],

    ) 🡪 *Not .pdf only pdf*

If we try to upload the same image again we see,



Change back to previous,

class ProductImage(models.Model):

    product = models.ForeignKey(

        Product, on\_delete=models.CASCADE, related\_name="images"

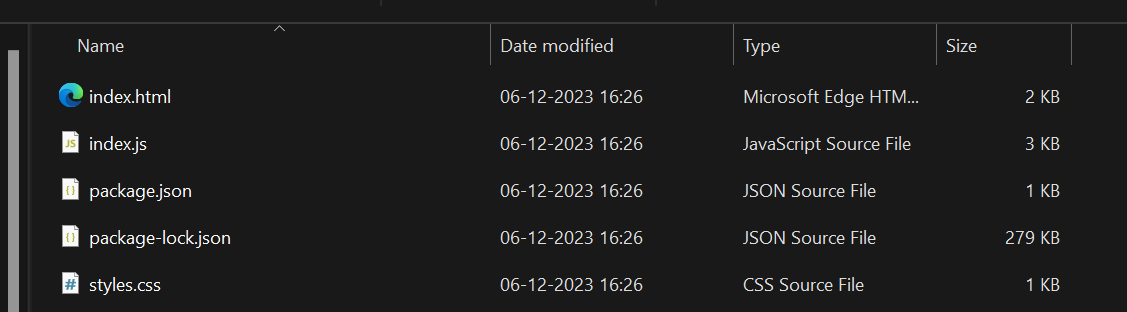
    )

    image = models.ImageField(upload\_to="store/images", validators=[validate\_file\_size])

**Setting up the client app**:

In The uploading files section of the course we have a *client app*

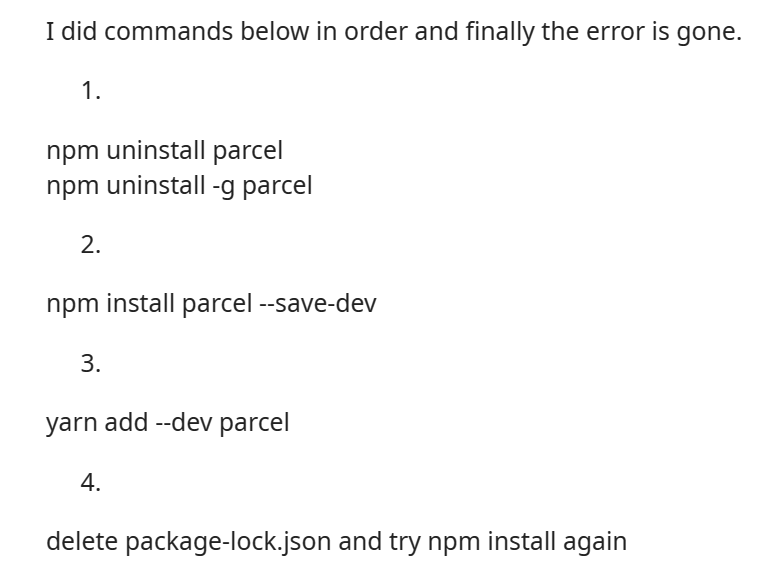
This is a very basic JavaScript project for uploading files.



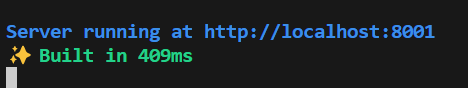
Let’s open this project in a new instance of VS code.

Then run npm install to install project dependencies.

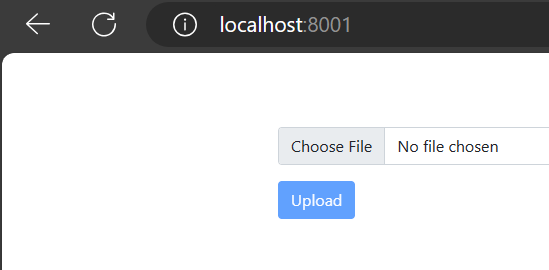
If it does not work, do these changes…



After npm start,



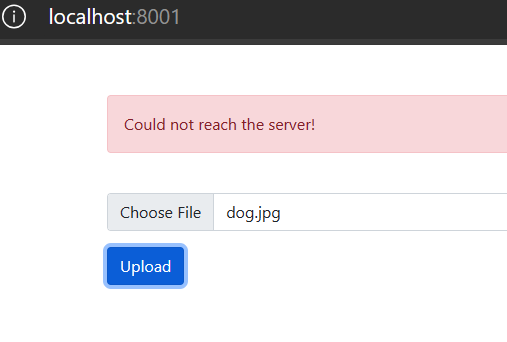
We can see our application running on localhost:8001 port,



But currently this feature is not working because there is a problem that we are going to talk about in next lesson.

**Enabling CORS**:

Let’s see what happens if we try to upload an image using our client app.

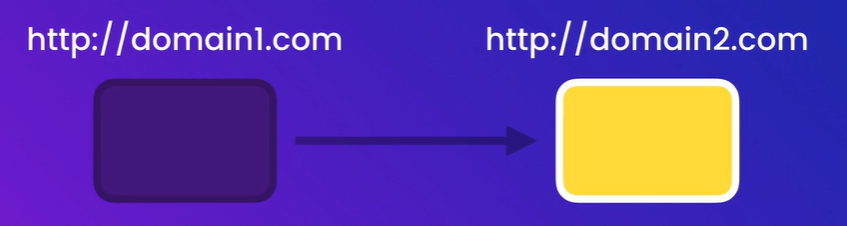


We get an error saying could not reach the server!

This is a common security feature that is implemented in all browsers, it’s called ***CORS***.



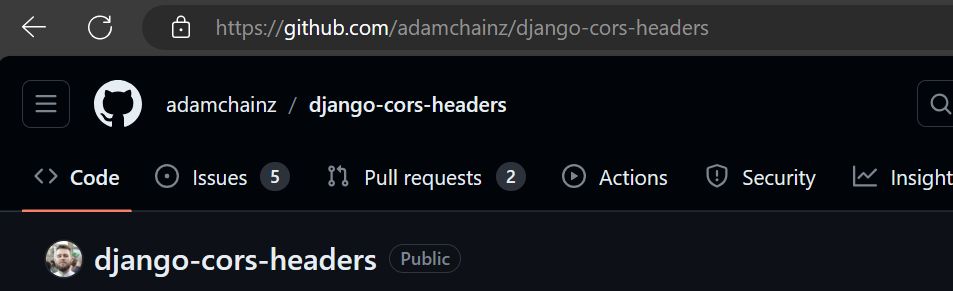
In practical terms, *this policy prevents an app hosted on one domain from sending a request to an app hosted on other domain*.



Our client app is hosted on port 8001 and trying to send a request to our store app on port 8000. So *even though both these apps are hosted on the same machine our browser sees them hosted on different domains*. That’s where CORS policy kicks in and prevents this request from reaching the server.

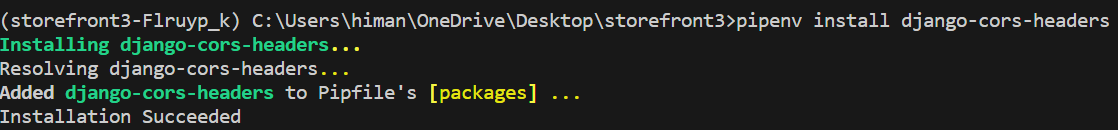
To solve this problem, we need to go to our backend and do a bit of configuration so that requests coming from this address (*localhost:8001*) are not blocked.

For this we will use a library called ***django-cors-headers***.



Steps to install and configure this library:

1. First pipenv install django-cors-headers.



Note: All the below configuration will be done in settings.py

1. Go to list of installed apps and register a app called ‘*corsheaders’*.

INSTALLED\_APPS = [

    "django.contrib.admin",,

    "django.contrib.messages",

    "django.contrib.staticfiles",

    "django\_filters",

    'corsheaders' 🡪 *Here*

1. Go to middleware and add ‘*corsheaders.middleware.CorsMiddleware*’

MIDDLEWARE = [

    "corsheaders.middleware.CorsMiddleware", 🡪 *Here*

    "debug\_toolbar.middleware.DebugToolbarMiddleware",

1. Next we need to set what origins can send requests to our backend for that we use a setting called *CORS\_ALLOWED\_ORIGINS*.

CORS\_ALLOWED\_ORIGINS = [

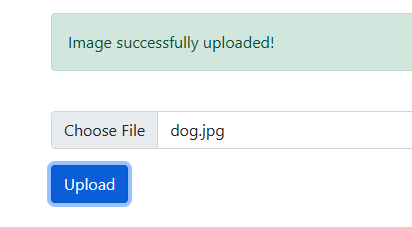
    "http://localhost:8001",

    "http://127.0.0.1:8001"

]

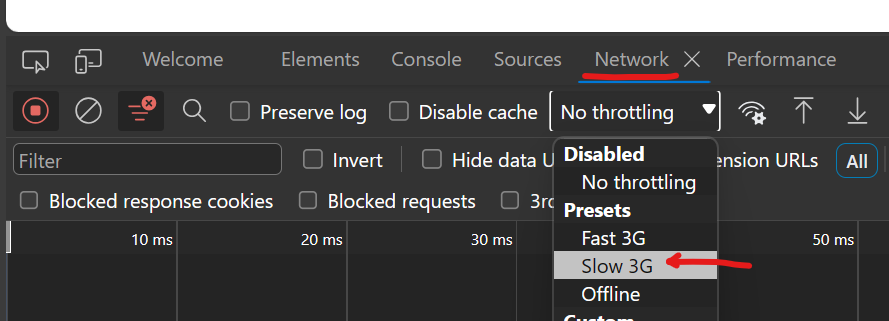
We set this setting to a list of strings specifying domain name or IP addresses.

So we are done here, now let’s try uploading our file.



So we have successfully uploaded the image.

Note: While uploading the file, we see a blue progress bar filling up as upload progresses, if need to see this slowly in action we can upload a larger file and enable *throttling*.

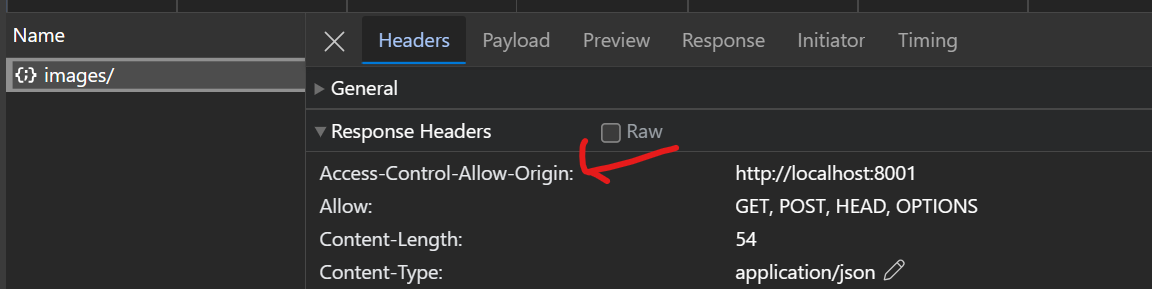


Go to network🡪Throttling🡪Slow 3G

One more thing we should see is the list of requests we send to backend.

The moment we press upload button, an image request sent to the server.

In its response headers we see Access-control-Allow-Origin which is set to localhost:8001

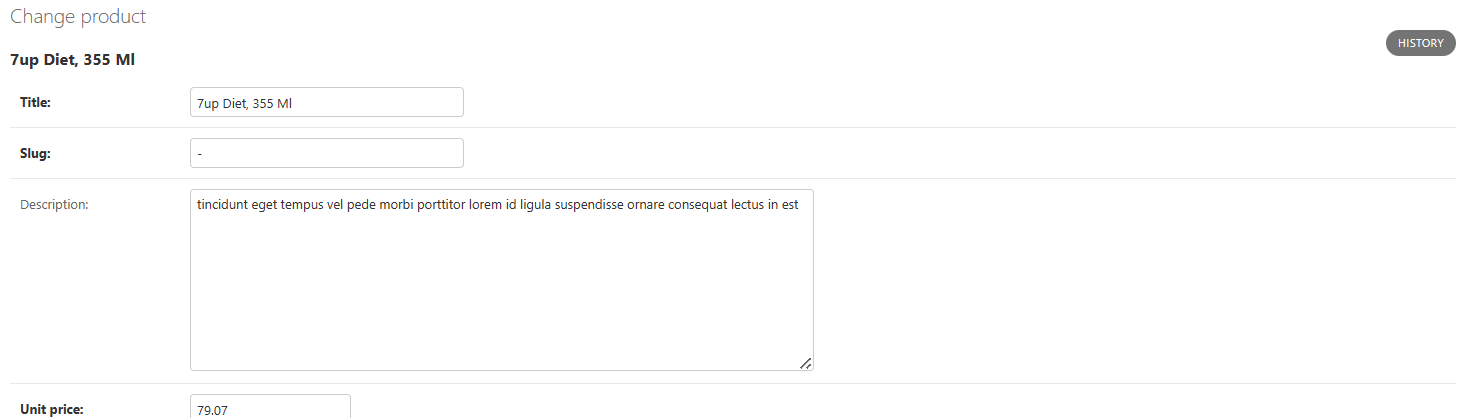


This is coming from the library we just installed.

So *we put this header in the response and return it to the client. The client will read this header and find out that it is allowed to send a request to our backend and then, it will upload the file*.

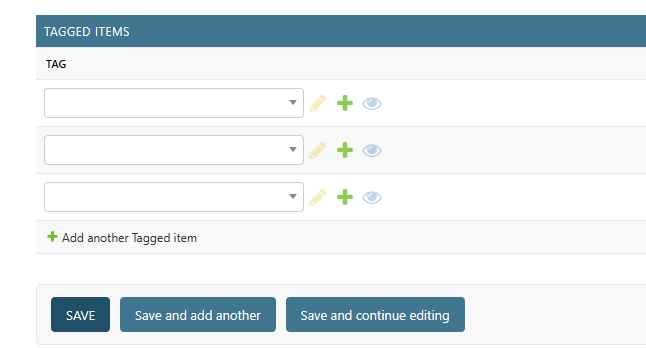
**Managing Images in the Admin**:

Last thing we will talk about in this section is managing product images in the admin interface.



If we look at any product in our admin interface, we can’t see any way to manage our product images here.

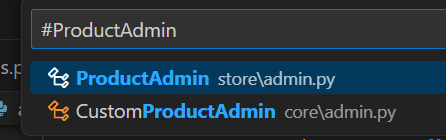
So similar to tags,



We want to show our product images right here.

Let’s do a quick recap on how we implemented this section.

Search for ProductAdmin,



We have two classes. We have one class defined in the store app and another one defined in core app.

The first one (*ProductAdmin*) is reusable part, so any project that uses the store app is going to use its admin module.

The other one (*CustomProductAdmin*) is defined in the core app(*very specific to this project*). It basically combines features from different apps.

class TagInline(GenericTabularInline):

    autocomplete\_fields = ['tag']

    model = TaggedItem

class CustomProductAdmin(ProductAdmin):

    inlines = [TagInline] 🡪 *Observe this...*

admin.site.unregister(Product)

admin.site.register(Product, CustomProductAdmin)

In this *CustomProductAdmin* class we have set the *inlines* to a list and in this list we have *TagInline* defined just above it.

So to show the Product images first *we need to define an inline class* (*ProductImageInline*) *and then we need to register under inlines*.

*But Where is the right place to define this inline class*?

We don’t want to define it inside core app because this app is very specific to this project. We should define it in the store app because the ProductImage class is defined there.

So any project that uses the store app should also have the capability to manage product images in the admin interface.

Let’s go to ProductAdmin class,

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    autocomplete\_fields = ['collection']

    prepopulated\_fields = {

        'slug': ['title']

    }

    actions = ['clear\_inventory']

    list\_display = ['title', 'unit\_price',

                    'inventory\_status', 'collection\_title']

    list\_editable = ['unit\_price']

    list\_filter = ['collection', 'last\_update', InventoryFilter]

    list\_per\_page = 10

    list\_select\_related = ['collection']

    search\_fields = ['title']

    def collection\_title(self, product):

        return product.collection.title

    @admin.display(ordering='inventory')

    def inventory\_status(self, product):

        if product.inventory < 10:

            return 'Low'

        return 'OK'

    @admin.action(description='Clear inventory')

    def clear\_inventory(self, request, queryset):

        updated\_count = queryset.update(inventory=0)

        self.message\_user(

            request,

            f'{updated\_count} products were successfully updated.',

            messages.ERROR

        )

Right before this ProductAdmin class, we are going to define an *inline* class.

class ProductImageInline(admin.TabularInline):

    model = models.ProductImage

Next we need to register this in the ProductAdmin class,

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    autocomplete\_fields = ['collection']

    prepopulated\_fields = {

        'slug': ['title']

    }

    actions = ['clear\_inventory']

    inlines = [ProductImageInline] 🡪 *Here*

Now there is a problem here. Let’s go back to our CustomProductAdmin,

class CustomProductAdmin(ProductAdmin):

    inlines = [TagInline]

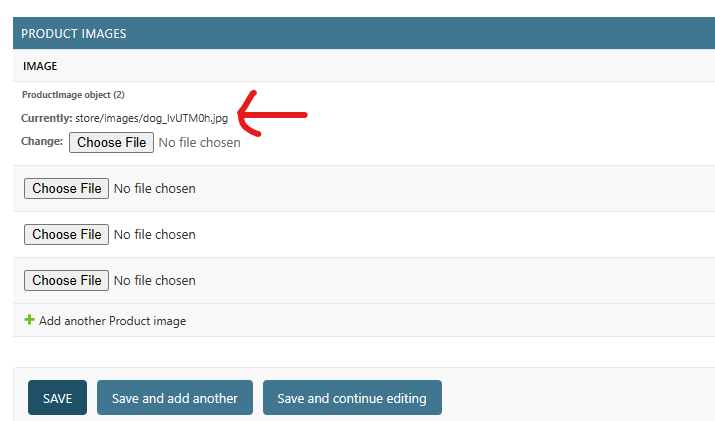
This class extends our ProductAdmin and here we overwriting *inlines* attribute. So whatever we set there is going to be lost here.

So in the core app, we should also add ProductImageInline in this list,

class CustomProductAdmin(ProductAdmin):

    inlines = [TagInline, ProductImageInline]

Now let’s see what happens,



In the admin panel under products, we can see a section with the list of product images. But we can only see the path.

*Would it not be nicer, if we could see thumbnail of each image*?

Back to the ProductImageInline class, here we need to set a attribute called *readonly\_fields* to a list and in this list we add a *thumbnail*.

class ProductImageInline(admin.TabularInline):

    model = models.ProductImage

    readonly\_fields = ['thumbnail']

Now thumbnail is not one of the fields of ProductImage class, so we need to define it as a method here.

class ProductImageInline(admin.TabularInline):

    model = models.ProductImage

    readonly\_fields = ['thumbnail']

    def thumbnail(self, instance): 🡪 *Here*

We give it two parameters. First one is self (*since it’s a method of a class*) and second parameter is an instance of the ProductImage class.

So *we are going to take that product image and convert it into HTML image*.

Now a product can have zero or more images, so we need to count for that here. We can say if *instance.image.name* is not empty (*this instance has two fields* ***product*** *and* ***image****, look in ProductImage class*) return an HTML image tag like this,

class ProductImageInline(admin.TabularInline):

    model = models.ProductImage

    readonly\_fields = ['thumbnail']

    def thumbnail(self, instance):

        if instance.image.name != '':

            return '<img src=""/>'

Here source (src) of the image is the URL of the image, so convert the return string into f – string or formatted string and here we render *instance.image.url*.

class ProductImageInline(admin.TabularInline):

    model = models.ProductImage

    readonly\_fields = ["thumbnail"]

    def thumbnail(self, instance):

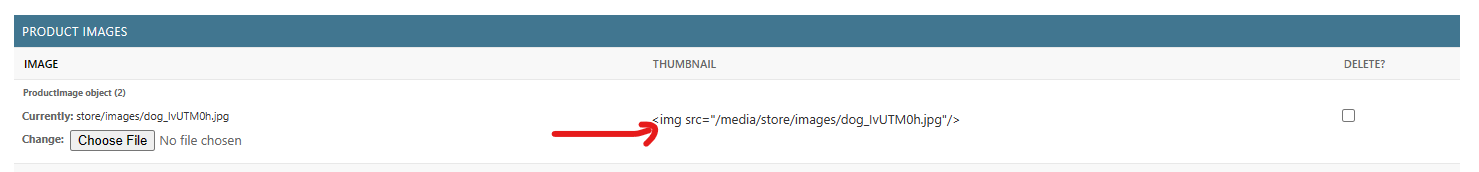
        if instance.image.name != "":

            return f'<img src="{instance.image.url}"/>'

        return ""

If we don’t have an image we just return an empty string.

We refresh the page,



Actual image is not here but we see <img> tag rendered as a string.

To solve it we need to import a *format\_html* function from Django,

from django.utils.html import format\_html, urlencode

We will *pass our image tag as an argument to the format\_html function*.

class ProductImageInline(admin.TabularInline):

    model = models.ProductImage

    readonly\_fields = ["thumbnail"]

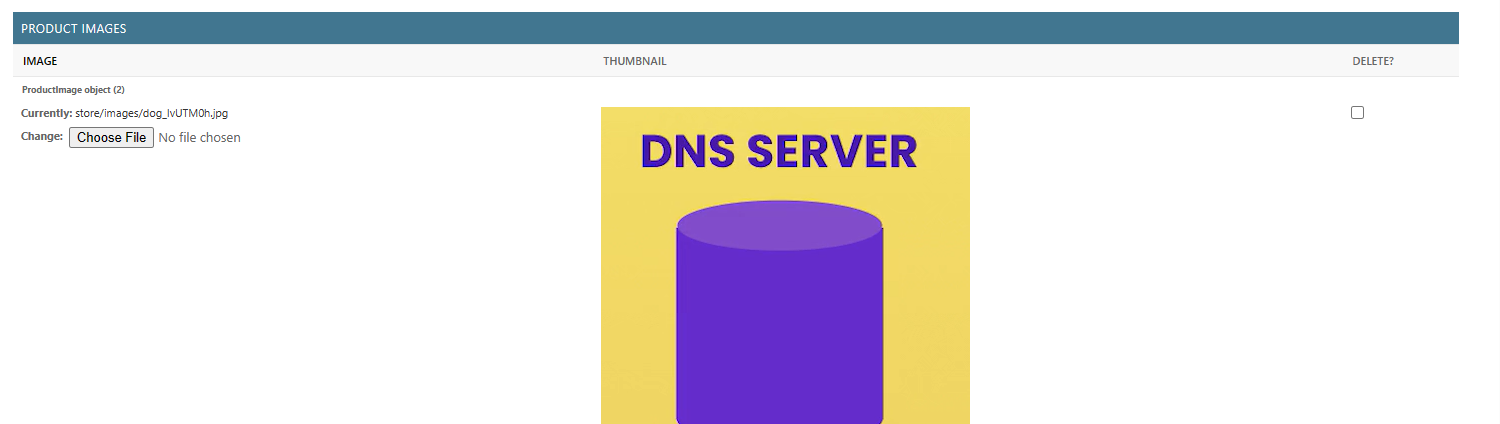
    def thumbnail(self, instance):

        if instance.image.name != "":

            return format\_html(f'<img src="{instance.image.url}"/>')

        return ""

Our image is visible now, but it’s way too big…



This is where we will use CSS to apply a bunch of attributes to this image. So add a class called “thumbnail” to the image tag.

class ProductImageInline(admin.TabularInline):

    model = models.ProductImage

    readonly\_fields = ["thumbnail"]

    def thumbnail(self, instance):

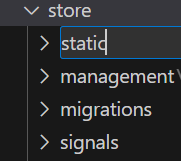
        if instance.image.name != "":

            return format\_html(f'<img src="{instance.image.url}" class="thumbnail"/>') 🡪 *Here*

        return "

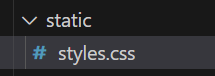
Now, we will define the styles for this image.

Create a folder called *static* inside our store app.



This is a special folder for Django. *When our project starts Django web server goes through every installed app and it will collect all the static files from this folder*.

In this folder we can add a file called *styles.css*.



We define our thumbnail class here,

.thumbnail{

    width: 100px;

    height: 100px;

}

We need to tell admin to import this CSS file on this page and for that we need to go to our original ProductAdmin class.

Here we are going to define a *Media* class which is an inner class (*just like Meta class*). With this *we can specify the static assets that should be loaded on the ProductAdmin page* (*so we can load CSS or JavaScript files*).

    class Media:

        css = {

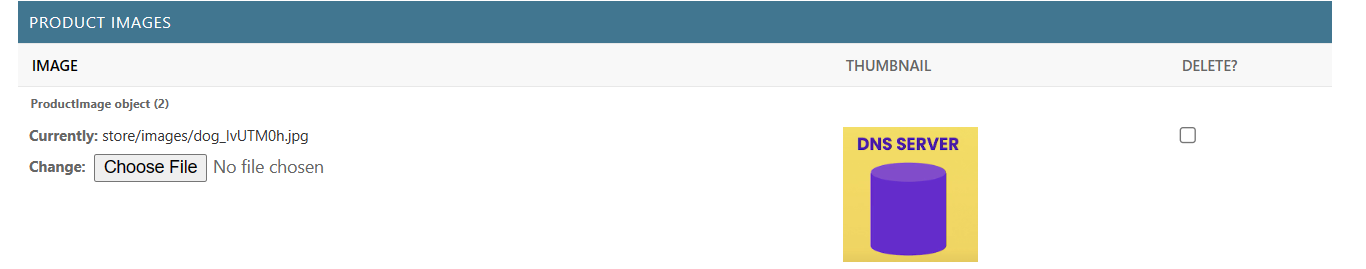
            'all':['styles.css']

        }

Here we set CSS to a dictionary, in which we add one key – value pair. *Key is ‘all’ and value is list of CSS files*.

Note: In CSS we have the concept of media type like ‘*screen’* with which we can apply *styles that only apply to screens* or ‘*print’* with which we can apply *styles when printing a page* but when we use ‘*all’*, these *styles will be applied everywhere*.

So this is how we can load our stylesheet or CSS file on the *ProductAdmin* page.



New width and height are applied to the thumbnail image. But if we pay close attention, we can see the image a little out of proportion.

To solve this we apply a new property to our thumbnail called *object-fit* and set it to *cover*.

.thumbnail{

    width: 100px;

    height: 100px;

    object-fit: cover;

}

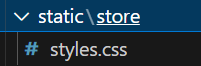
OK…image looks better now!

But there is one problem in this implementation.

When our server starts, *Django web server is going to look at static folder of every installed app and its going to collect their files. That means if we have another file with the same name in another app then one of the file is going to override other file*.

That is why we need to *namespace* these files.

So inside the static folder, we create a new folder called store and cut -- paste our styles.css file into it.



Let’s update the media path in Media class.

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    autocomplete\_fields = ["collection"]

    prepopulated\_fields = {"slug": ["title"]}

    actions = ["clear\_inventory"]

    inlines = [ProductImageInline]

    list\_display = ["title", "unit\_price", "inventory\_status", "collection\_title"]

    list\_editable = ["unit\_price"]

    list\_filter = ["collection", "last\_update", InventoryFilter]

    list\_per\_page = 10

    list\_select\_related = ["collection"]

    search\_fields = ["title"]

    def collection\_title(self, product):

        return product.collection.title

    @admin.display(ordering="inventory")

    def inventory\_status(self, product):

        if product.inventory < 10:

            return "Low"

        return "OK"

    @admin.action(description="Clear inventory")

    def clear\_inventory(self, request, queryset):

        updated\_count = queryset.update(inventory=0)

        self.message\_user(

            request,

            f"{updated\_count} products were successfully updated.",

            messages.ERROR,

        )

    class Media:

        css = {"all": ["store/styles.css"]} 🡪 *Here*