Django Crash Course

In this guide, we are going to build a web app for a weed business. This web app will include an ecommerce section (which is what we are going to be focusing on).

Here's what we are going to cover:

- Creating a Django project
  - Creating a django app in our project
- Models
  - Creating models
  - Interacting with the models in the django shell
  - Writing automated tests for the models
- The Django admin site
  - Registering models
  - Customizing the admin site
  - Users
    - Groups and permissions
- Views and URLS
  - Writing views
  - Linking views to URLs
  - Passing and accessing Python objects to and from HTML templates
  - Checking permissions in views
- Switching databases
  - PostgreSQL

Creating a Django project

To create a Django project, you will need to install the Django package.
It is recommended to install the package (along with any other packages you will use in the project) in a virtual environment (the commands related to creating/activating the virtual environment are for Windows).

1. Create a new folder on your computer called *django_crash_course*: `mkdir django_crash_course`
2. Create a virtual environment inside the django_crash_course directory:
   - `cd django_crash_course`
   - `python -m virtualenv <env_name>` e.g. `python -m virtualenv my_environment`

If you don't have the virtualenv package, install it using `python -m pip install virtualenv`

3. Activate the virtual environment: `my_environment\Scripts\activate`
4. Install Django: `pip install Django`
   To check if Django was installed successfully, run: `python -m django --version`
5. Create your Django project called *iDeal*: `python -m django-admin startproject iDeal`

Ok so now the project is created, but we still have to create a Django app.

Projects vs Apps

- An app is a web application for a particular purpose e.g. we are going to create and ecommerce app. We can create as many apps as we need in our projects but it is important that the apps be independent of each other and serve distinct roles.
- A project is a collection of configurations and apps for a particular website. It can contain multiple apps.

Creating an app

1. We will create the ecommerce app in the *iDeal* directory (the one that has the manage.py file)
   - `cd iDeal`
   - `python manage.py startapp ecommerce`
2. After creating the app, we have to add it to the *INSTALLED_APPS* list in our *iDeal/settings.py* file

```
INSTALLED_APPS = [

  'django.contrib.admin',

  'django.contrib.auth',

  'django.contrib.contenttypes',

  'django.contrib.sessions',

  'django.contrib.messages',

  'django.contrib.staticfiles',

  'ecommerce'  # ⟵
```

```
    ]
```

Models

Creating models

Models are Python classes represent the tables in our database. They allow us to manipulate data from our database without writing SQL queries for the specific DBMS. Django uses an SQLite database by default, we will cover using a PostgreSQL database later.

Open the `ecommerce/models.py` file in a text editor and paste the following code into it

```
from django.db import models



class Collection(models.Model):

    name = models.CharField(max_length=20, unique=True)

    time_created = models.DateTimeField(auto_now_add=True)



    class Meta:

        verbose_name = "Collection"

        verbose_name_plural = "Collections"

class Product(models.Model):

    name = models.CharField(max_length=100)

    price = models.FloatField(default=0)

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

    cover_image = models.FileField(null=True)

    quantity = models.IntegerField(default=0)
```

```python
        # validate the price and quantity using the clean method and write a
test



class CollectionProduct(models.Model):

    collection = models.ForeignKey(Collection, on_delete=models.CASCADE)

    product = models.ForeignKey(Product, on_delete=models.CASCADE)



class Variant(models.Model):

    product = models.ForeignKey(Product, on_delete=models.CASCADE)

    name = models.CharField(max_length=30)



    # validate product and variant name are unique together



class Image(models.Model):

    product = models.ForeignKey(Product, on_delete=models.CASCADE)

    file = models.FileField()



class Order(models.Model):

    time_made = models.DateTimeField(auto_now_add=True)

    discount = models.FloatField(default=0) # given in fractions, from 0 - 1
```

```
    # create a method for getting the total order price

    # validate the discount > 0 and < 1



class OrderProduct(models.Model):

    order = models.ForeignKey(Order, on_delete=models.CASCADE)

    product = models.ForeignKey(Product, on_delete=models.CASCADE)

    variant = models.ForeignKey(Variant, on_delete=models.SET_NULL,
null=True)

    quantity = models.IntegerField()



    # validate quantity > 0
```

The comments indicate ways in which we can improve our models

After creating our models, to add these tables to our database we have to:

- Create a database migration: `python manage.py makemigrations <app_name>`, for our case; `python manage.py makemigrations ecommerce`
  Migrations contain the code that applies the changes you make in the models of an app to the database. The migration files are stored in the `<app_name>/migrations/` directory (DON'T MESS AROUND WITH THESE FILES)
- Apply the migrations: `python manage.py migrate`

Interacting with the models in the Django shell

The Django shell provides a quick and interactive way to test components related to your project. In this section, we will be testing the models we just created.

1. Open the Django shell by running `python manage.py shell`
2. Import our models from the ecommerce app
   `from ecommerce.models import *`
3. Get the list of collections and products, they should both return an empty queryset

```
Collection.objects.all()
Product.objects.all()
```

4. Create a new collection: `c = Collection.objects.create(name='Collection 1')`. Create a new product: `p = Product.objects.create(name='Product 1', price=-900, quantity=-10)`

5. The product gets created successfully whereas it doesn't make sense for it to have negative values for price and quantity. In the next section we are going to look at how to test the models automatically to ensure that they follow some rules.

6. We will look into more sophisticated model actions in the views.

Writing automated tests for the models

Django heavily encourages the philosophy of TDD (Test Driven Development).
This is a coding practice in which we

- Write tests for a particular behavior, these tests fail the first time they are run
- We then go and code out the required behavior that is supposed to make the tests pass
- We run the tests again, this time they should pass. If the tests still fail then we should go back to the previous step.

We observed in the previous section that we could create a product with negative prices and quantities which doesn't make sense.
We want the models to raise a **ValidationError** if the price or quantity is less than 0.
Let's go and write out the tests to check the price and quantity

Open the `ecommerce/tests.py` file and paste the following code into it

```python
from django.test import TestCase

from django.core.exceptions import import ValidationError




from .models import *



class ProductModelTests(TestCase):

    def test_price_less_than_0_raises_error(self):

        """
```

```python
        checks whether the save method raises a ValidationError when the
price < 0

        """

        product = Product(name='Test product', quantity=3, price=-950)


        self.assertRaises( ValidationError, product.save )



    def test_quantity_less_than_0_raises_error(self):

        """

        checks whether the save method raises a ValidationError when the
quantity < 0

        """

        product = Product(name='Test product', quantity=-3, price=950)


        self.assertRaises( ValidationError, product.save )

    def test_quantity_and_price_less_than_0_raises_error(self):

        """

        checks whether the save method raises a ValidationError when the
quantity < 0 and price < 0

        """

        product = Product(name='Test product', quantity=-3, price=-950)


        self.assertRaises( ValidationError, product.save )
```

After writing the tests, we run them using `python manage.py test`. The output should tell us that 3 tests failed.

We now have to rewrite our Product model in such a way that it passes these tests i.e. raises a ValidationError if the quantity < 0 or the product < 0.

We will do this by overriding the clean and save methods of our model. Copy the code below and paste it into the Product class under the comment

```
    # validate the price and quantity using the clean method and write a
test

    def clean(self) → None:

        if self.quantity < 0:

            raise ValidationError("The quantity of the product cannot be
less than 0")

        if self.price < 0:

            raise ValidationError("The price of the product cannot be less
than 0")

        return super().clean()



    def save(self, *args, **kwargs) → None:

        self.clean()

        return super().save(*args, **kwargs)
```

Now running the tests again show us an OK message

To write tests for another model, simply create a new class that inherits from TestCase and define your test methods inside that class.

Let's create one more TestCase for the variant model.

In the *ecommerce/tests.py* file, add this line to the top where the imports are: `from django.db.utils import IntegrityError`
Paste the following code into the `ecommerce/tests.py` file under the ProductModelTests class.

```python
class VariantModelTests(TestCase):

    def test_product_and_variant_name_unique_together(self):

        """

        Create and save a variant with a product and a name

        Create another variant with the same product and name as the first

        The save method of the second variant should raise an IntegrityError

        """

        product = Product.objects.create(name='Test product', quantity=3,
price=1000)

        v1 = Variant.objects.create(name='Testv', product=product)


        v2 = Variant(name='Testv', product=product)


        self.assertRaises( IntegrityError, v2.save )
```

Running `python manage.py test` again should show 1 failure.
To make this test pass, we will use a Meta class in our variant model. Paste the following code into the Variant class, under the comment.

```python
        # validate product and variant name are unique together
    class Meta:

        unique_together = [

            ["product", "name"]

        ]
```

Make migrations, apply the migrations and run the tests again. This time, they should all pass.

Here are some more tests that you can write:

- In the CollectionProduct model, test whether the collection and product attributes are unique together
- In the Order model, check that the discount cannot be < 0 and > 1
- In the OrderProduct model, check
    - that the quantity > 0
    - that the quantity is less than or equal to the quantity of the Product
    - that the Product's quantity is reduced after the OrderProduct is saved

The Django admin site

Django provides a pretty neat admin interface that allows you to interact with your models without having to write any extra code.
The admin's recommended use is limited to an organization's internal management tool. It's not intended for building your entire frontend around.

To access the admin site we need a superuser account. We can create one using the `python manage.py createsuperuser` command.
Just run the command, fill in your username and password (you can leave the Email address empty).

Once you've created the user, run the server and navigate to `<SERVER_IP>:<PORT>/admin/` e.g. (localhost:5000/admin)[localhost:5000/admin].
The interface should look like this when you've logged in:
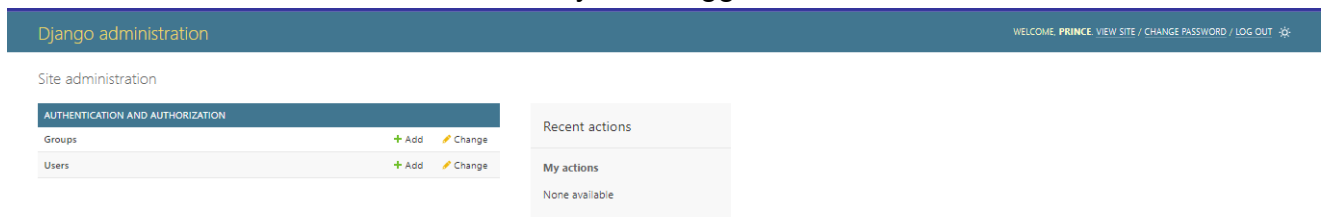
Registering models

Django doesn't know which models you want to access from the admin interface. To make a model accessible from the admin interface, we have to register it.
We register the models for the different apps we have, in our project we have just 1 app and we want to access all of its models from the admin site.

Paste the following code into `ecommerce/admin.py`

```python
from django.contrib import admin


from .models import *


admin.site.register(Collection)

admin.site.register(Product)

admin.site.register(CollectionProduct)

admin.site.register(Variant)

admin.site.register(Image)

admin.site.register(Order)

admin.site.register(OrderProduct)
```
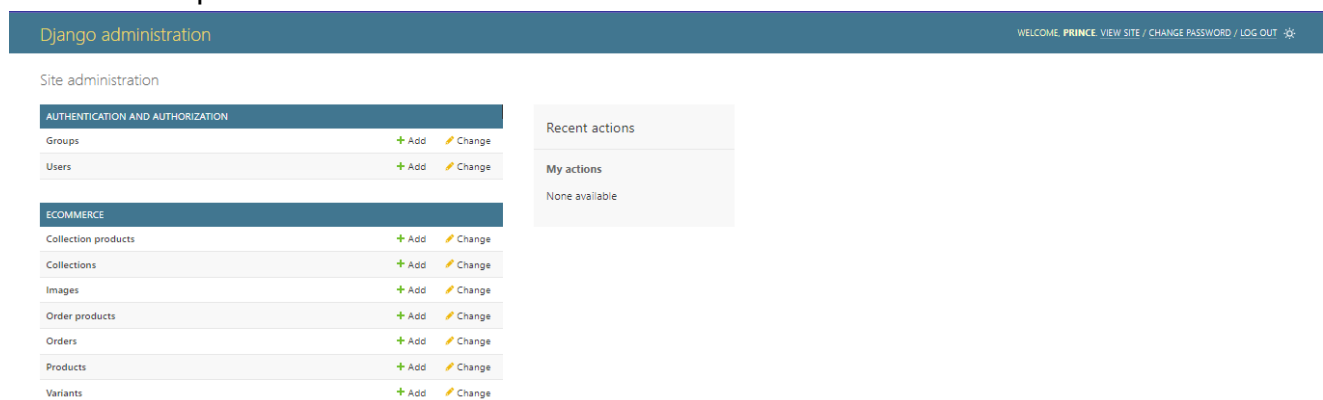
Saving the file and refreshing (or rerunning if you stopped the server) the admin page should show this output.

Before going to the next step, try adding a Product, an Image, a Collection Product and an Order (with corresponding Order products).
If you're up for it, try modifying any of the records you created above using the admin interface.

Customizing the admin site

The admin site works alright but it's not very user-friendly. Here are some issues that one notices when interacting with it:

- Our records don't have explicit names e.g. the Products are called `Product object (x)` where x is a number. This doesn't make much sense so we have to modify it.
- To add an Image to a Product, we need to first add the Product before adding the image. Ideally we would like to be able to add images while creating the product
- To add a Variant, we need to create a product first and then add the variant to the Product. To add multiple variants, we need to do so one by one which can be time-consuming.
- To add a Collection product, we need to create a collection, a product and then the collection product. Bear in mind that a product can belong to more than one collection. So with the current user interface, we need to add each of these Collection Products one by one which is not a very good user experience.
- To add an Order Product, we would have to first add the order and then manually add the Order products to the order. Ideally we want to be able to add Order products while creating the order.

1. Giving explicit names to records

The first issue is the easiest and doesn't concern the admin site directly but is a useful tip to know. To modify the names displayed for our records, we simply need to override the _str_ method of the model. The code below shows an example using the Product model, paste it in the Product class in the ecommerce/models.py file.

```python
def __str__(self) -> str:

    return f"{self.name}: {self.price} XAF"
```

Save the file and look at the list of products in the admin page, the string displayed should be updated too.

2. Adding images when adding a Product

To do this, we need to create 2 new classes

- `ImageInline` that subclasses the `admin.StackedInline` class. This class will enable us to add images inline with a product.
- `ProductAdmin` that subclasses the `admin.ModelAdmin` class. This class will define how our Product model will be displayed and manipulated on the admin site.
  Paste the code below into the `ecommerce/admin.py` (Replace the old code with this one)

```python
from django.contrib import admin


from .models import *


class ImageInline(admin.StackedInline):

    model = Image

    extra = 3


class ProductAdmin(admin.ModelAdmin):

    inlines = [ImageInline]
    list_display = ("name", "price", "quantity")


admin.site.register(Collection)

admin.site.register(Product, ProductAdmin)

admin.site.register(CollectionProduct)

admin.site.register(Variant)

admin.site.register(Order)

admin.site.register(OrderProduct)
```
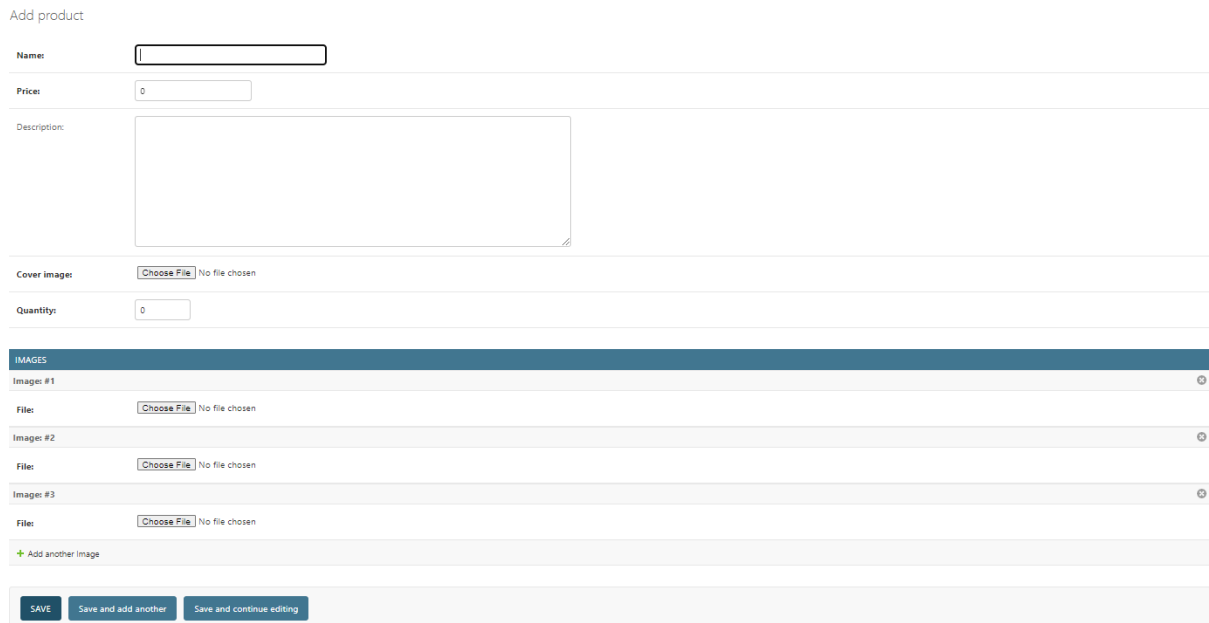
Ok, so what have we done?

- We removed the line that registers the Image model
- This tells Django that 'Image' objects are edited on the Product admin page. By default, provide enough fields for 3 choices (`extra = 3` in the ImageInline class)
- The list_display tells Django to display the name, price and quantity on the Products list page
  Save the file and try adding a Product from the admin site.

This is what the Add product page now looks like, we can specify 0 or more images to add alongside the Product.

Apply the same technique to solve the issue of the variants, collectionproducts and orders:

- Create a VariantInline class with extra = 3 like the ImageInline class
- Create a CollectionProductInline class with extra = 3 like the ImageInline class
- Create an OrderProductInline class with extra = 3, create an OrderAdmin class and add the OrderProductInline class to its list of inlines.

In the end, the `ecommerce/admin.py` file should look like this

```
from django.contrib import admin




from .models import *




class ImageInline(admin.StackedInline):
```

```python
    model = Image

    extra = 3


class VariantInline(admin.StackedInline):

    model = Variant

    extra = 3


class CollectionProductInline(admin.StackedInline):

    model = CollectionProduct

    extra = 3


class OrderProductInline(admin.StackedInline):

    model = OrderProduct

    extra = 3


class ProductAdmin(admin.ModelAdmin):

    inlines = [ImageInline, VariantInline, CollectionProductInline]

    list_display = ("name", "price", "quantity")


class OrderAdmin(admin.ModelAdmin):

    inlines = [OrderProductInline]
```

```
admin.site.register(Collection)

admin.site.register(Product, ProductAdmin)

admin.site.register(Order, OrderAdmin)
```

3. Customizing the admin look and feel

Right now our admin site looks really generic. We would like for starters to show the name of our organization on the title instead of 'Django administration'

We can do this by modifying the Django admin template.

Create a `templates` directory in the same location where your `manage.py` file is found.
Open your `iDeal/settings.py` file and add a `DIRS` option to the `TEMPLATES` setting.
Replace the `TEMPLATES` setting with this code:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"], # ⟵
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    },
]
```

DIRS is a list of filesystem directories to check when loading Django templates.

Now create an `admin` directory in the `templates` directory you created earlier (this is where we are going to store our custom admin templates).
Copy the template `admin/base_site.html` from within the default Django admin template directory in the source code of Django itself (django/contrib/admin/templates) into that directory.

If you have difficulty locating the Django source files, run the following command `python -c "import django; print(django.__path__)"`. Running it on my machine gave me this output (I'm using a virtual environment for this)

`['D:\\IAI Teaching Documents\\Level 3\\Django Crash Course HTML, CSS & JS\\crash_course\\environment\\lib\\site-packages\\django']`

Replace the code in the `admin/base_site.html` file that you copied to your directory with the one below

```
{% extends "admin/base.html" %}



{% block title %}{% if subtitle %}{{ subtitle }} | {% endif %}{{ title }} |
{{ _('iDeal admin') }}{% endblock %}



{% block branding %}

<div id="site-name"><a href="{% url 'admin:index' %}">{{ _('iDeal
administration') }}</a></div>

{% if user.is_anonymous %}

  {% include "admin/color_theme_toggle.html" %}

{% endif %}

{% endblock %}



{% block nav-global %}{% endblock %}
```

Users

We already created one superuser before and we can create more users with less permissions.
On the admin site, there's a section for 'Authentication and Authorization' that contains Groups and Users.

Go to the Users page to see the list of users and click the 'Add User' button. This will take you to a page where you have to fill the username and password. Put whatever you want

here.

You can also edit the user after creating to put in more details such as the first and last name as well as the permissions.

When taken to this page:

- Fill in the user's first and last name
- Check the 'Staff status' checkbox (only staff users can access the admin)
- Go down to the User permissions section and select any permissions you want the user to have.
- Permissions are written in the following manner
  `App_name|model_name|permission_name` e.g. `Ecommerce | Product | Can add product` means the user can add a product.
- Let's give our user the following permissions
  - CRUD on product (add, view, change, delete)
  - CRUD on Collections
- Once you've selected your desired permissions, Click on the arrow pointing to the right between the 2 sections to move the permissions over to the Chosen user permissions. After doing so, you should have something similar to this:
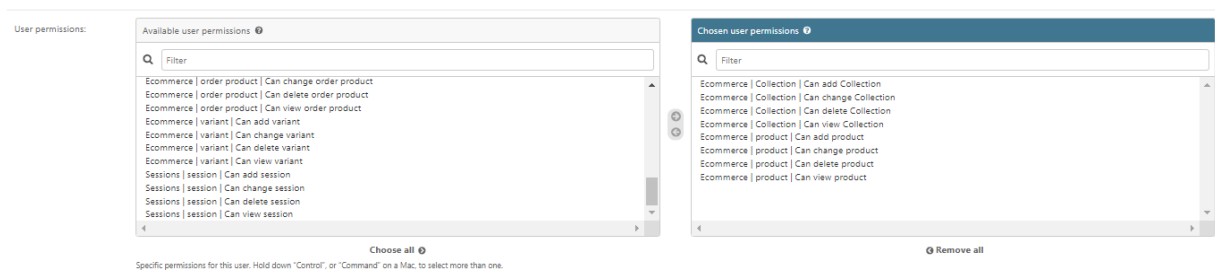
- Save the user.
- Log out and log back in with the new user, your interface should look like this:
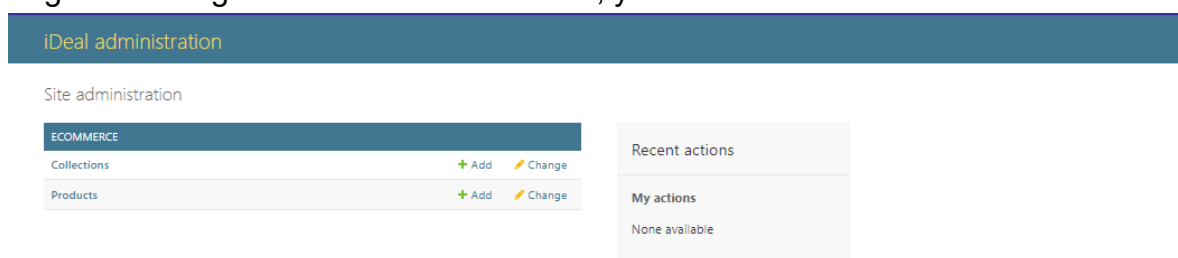
- Notice how this user can only view the models that he has permissions on.

Groups

So we created a user and assigned permissions above, but if we had to create many users and assign them the same permissions, it gets pretty tiresome.

So instead, we would create a Group with permissions and then assign users to that group. Each user in a group has all the permissions that the group has. A user can belong to more than one group.

To create a group, we go to the Add Group page and define the group name, as well as the group's permissions (similar to how we defined them in the Users) and save.

Log back in as the superuser and do the following:

- Add a group called Product Manager
- Give it the following permissions
    - CRUD on Products
    - CRUD on Collections

- Modify the user you created in the previous section by
  - Adding the Product Manager group to their list of groups
  - Removing all of the permissions that it has, so that the only permissions it will have are those assigned to the group(s) he belongs to.

- Add another user to this same Group
- Log back in as one of the users that you created and try adding a Product.
- Notice that all of our inline models are gone (Image, CollectionProduct, Variant). This is because this user doesn't have the permission to add them.

- So log back in as the superadmin and modify the group such that it has permissions to manipulate the above models.

Views and URLs

Ok, so we've customized the admin part of our application which is intended to be used by the administrators/employees of the organization but what about the end users?
The end users cannot be allowed to use the admin site, so we have to create a separate frontend for them. We will create the different pages of this frontend by writing views in our `ecommerce` app.

A view function, or simply a view, is a Python function or method that receives a web request and returns a web response. These views are responsible for processing user input, interacting with the model, and returning the appropriate response, often rendered using a template (HTML file).

Views are connected to specific URLS via the URLconf (URL configuration). This is typically defined in the urls.py file.

Let's create a simple index view. To do so:

- Go to the `iDeal/urls.py` file and paste the code below:

```
from django.contrib import admin

from django.urls import include, path



urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include('ecommerce.urls'))
```

```
]
```

This tells Django to check in the `ecommerce/urls.py` file for any URL that does not start with admin.

- Create a function, `index` in the `ecommerce/views` file that simply returns a Hello message. Replace the code in the `ecommerce/views.py` file with the one below:

```
from django.shortcuts import render

from django.http import HttpResponse


def index(request):

    return HttpResponse("Hello, World!")
```

- For this view to be accessible to a user, we need to connect it to a URL. We had already told Django to search for URLS (except for the admin) in the ecommerce.urls file, now we have to create that file and its content. Create a urls.py file and paste the following code in it:

```
from django.urls import path



from . import views



urlpatterns = [

    path('', views.index, name='index')

]
```

- Now going to `localhost:8000` should display the "Hello, World!" message.

Setting up file uploads

We wrote our models earlier and some of them included fields for uploading files. When we upload files using these models, the files are stored in the project directory which is not

ideal.

Also we currently cannot access these files in our views.

In this step we are going to configure our Django project so that it uploads the files to a particular directory and can serve static files.

- Add a new folder to your project directory called `uploads`. You might want to add this folder to the .gitignore as it can get pretty large.
- Go to the `iDeal/settings.py` and add these 2 lines at the end:

```
MEDIA_ROOT = BASE_DIR / "uploads/"

MEDIA_URL = '/media/'
```

- Go to the `iDeal/urls.py` file and replace the urlpatterns list with this one

```
urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include('ecommerce.urls'))

] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

- Ok, almost done. Now we need to specify in the models where the uploaded files should be saved. Go to your `ecommerce/models.py` file and in each `FileField` pass another parameter called `upload_to` e.g. in the Product class, `cover_image = models.FileField(null=True, upload_to='uploads')`. Do the same for the Image model
- Now if you try to upload from the admin, it gets saved to the uploads folder.