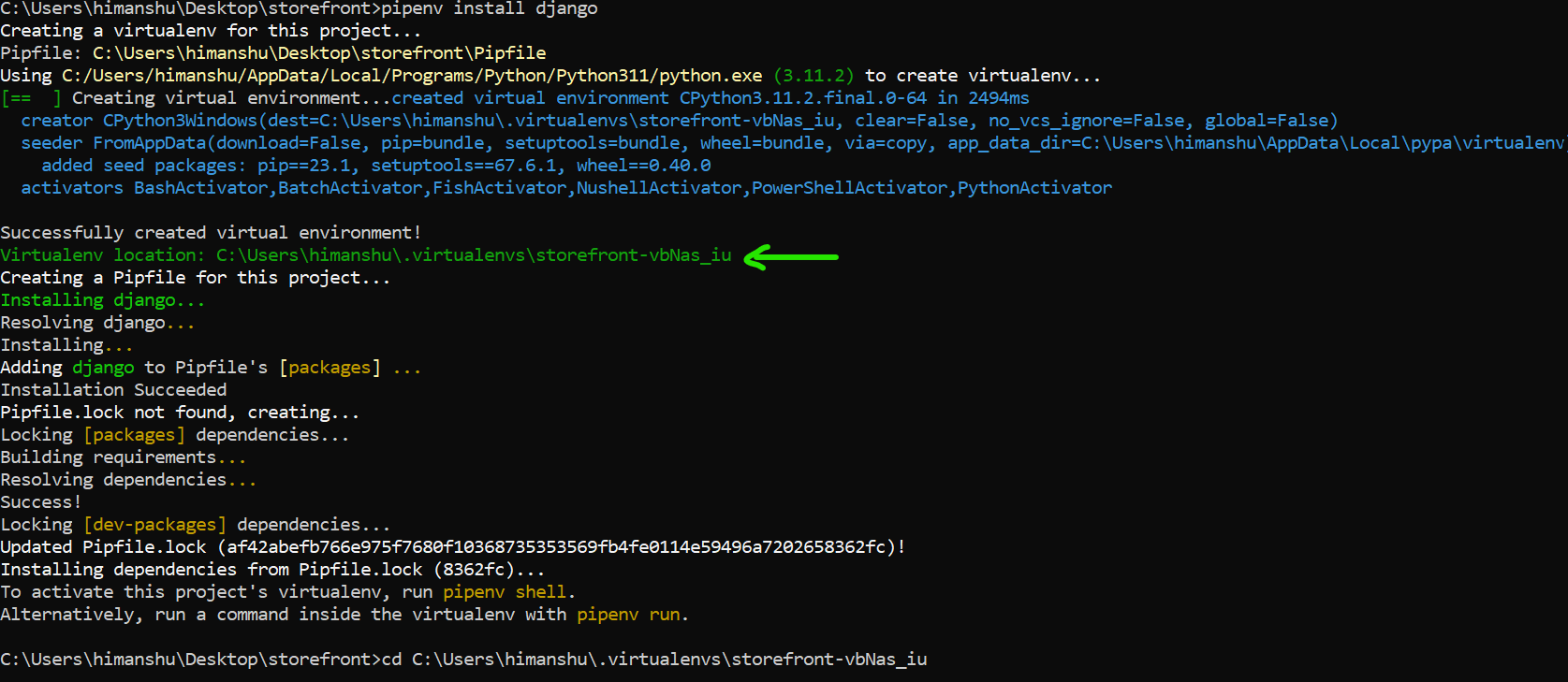
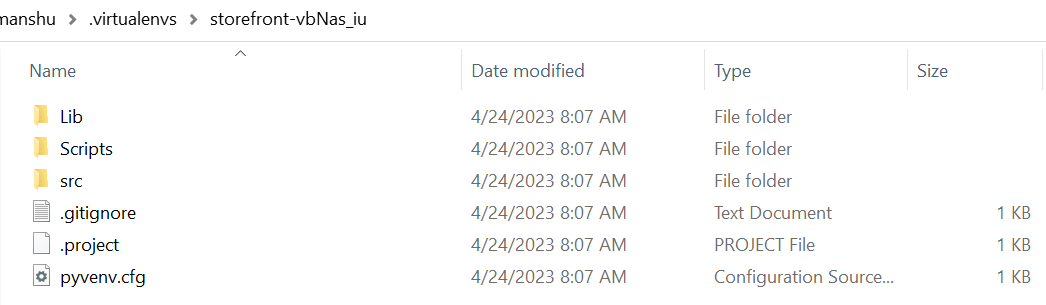


**Project setup**:

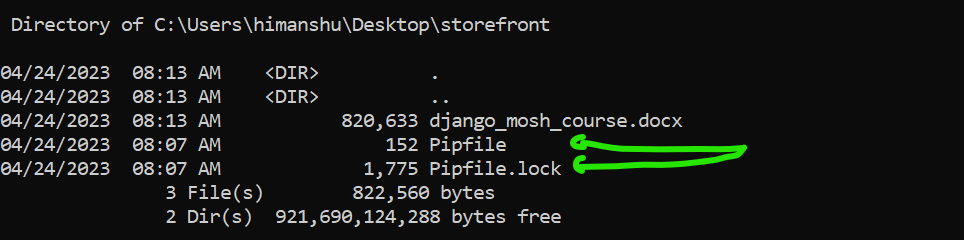
1. Create a folder called storefront, cd into it and run below commands
2. python3 -m pip install pipenv
3. pipenv install django



1. Copy the virtual environment C:\Users\himanshu\.virtualenvs\storefront-vbNas\_iu and go inside this folder.



1. pipenv created a virtual environment and installed django in it but in addition it also created two more files in our storefront directory.



1. Pipfile is like package.json for JS projects.

[[source]]

url = "https://pypi.org/simple"

verify\_ssl = true

name = "pypi"

[packages]

django = "\*"

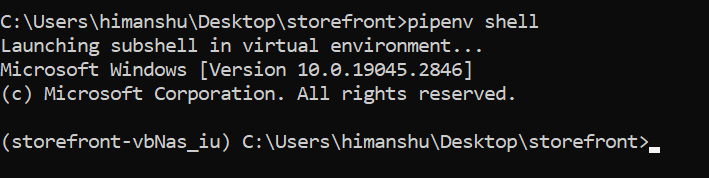
[dev-packages]

[requires]

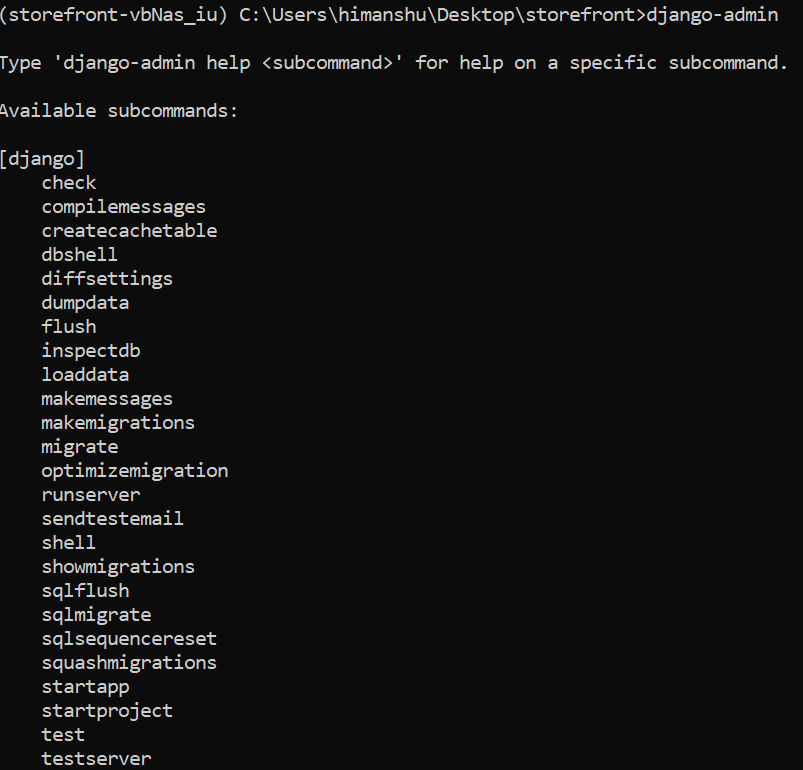
python\_version = "3.11"

django = “\*” means latest version of Django package that our application is depended upon.

1. Now back in our project directory run pipenv shell



Note: When you type django-admin, you will see a lot of commands like this



django-admin is a command line utility that comes with Django

1. Run django-admin startproject storefront . It will create a directory inside our project that is the core of our application.

Note: manage.py

This acts as a wrapper around django-admin. So from now on instead of using django-admin in command line we will use manage.py. Reason is that manage.py takes the settings of this project into account.

1. In order to run server

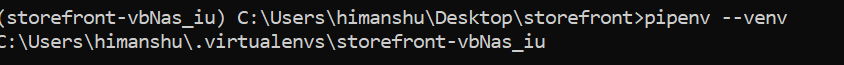
python manage.py runserver 9000 (here we can give port number as another command line argument, but if we do not supply it default is 8000)

Django application will start on <http://127.0.0.1:8000/>

**Setting up interpreter in VS code**:

1. Get the exact path of our virtual environment using

pipenv –venv



1. Ctrl+Shift+p to open command palette.
2. Select interpretor and Copy/paste the path of the environment

Note: Inside setting.py module, we have a bunch of different settings including INSTALLED\_APPS

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

]

sessions app is legacy, session is temporary memory on the server for managing user’s data. **DO NOT** delete this app.

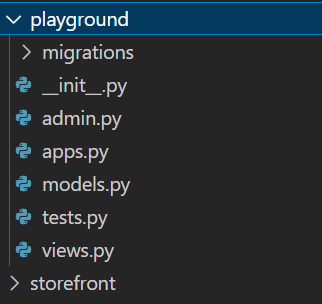
"django.contrib.sessions",

messages app used for displaying one time notification to user.

staticfiles app used for displaying static files like images, css files

**Creating a new app**:

1. Run this command python manage.py startapp playground.



Our new app called *playground* is now created.

1. Now, register the app in the list of installed apps

INSTALLED\_APPS = [

    "django.contrib.admin",

    "django.contrib.auth",

    "django.contrib.contenttypes",

    "django.contrib.messages",

    "django.contrib.staticfiles",

    "playground",

]

**Writing Views**:

1. As we know HTTP is a request-response protocol. So every data exchange involves a request and a response. This is where we use views.py module in Django.

Or more accurately, it is a *request handler*! (in some frameworks it is called an *action*…)

1. So we write a function, *say\_hello* which should take a request object and returns a response.

“*In real world scenario, we can do many things from this function like pull data from database, transform data, and send emails and so on*”

For now just return a response.

from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.

def say\_hello(request):

    return HttpResponse("Hello World")

Note: We import HttpResponse class from django.http module and return an instance of this class from our view function.

**Mapping URLs to views**:

Here is the problem statement, whenever we go to <http://127.0.0.1:8000/playground/hello> our view function should be called and return “Hello World” to the user.

1. Go to our playground app folder and create a new file called urls.py. In this module we are going to map our URLs to our view functions.
2. Import path function from django.urls

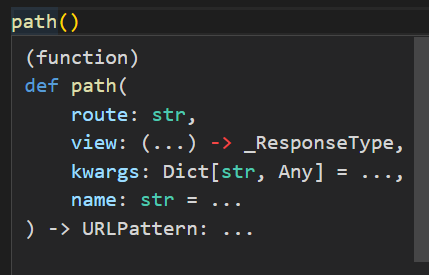
from django.urls import path

1. From current app Import views module to reference our view functions

from . import views

1. Define a variable called urlpatterns=[]. It contains array of urlpattern objects.
2. We use path() to create a urlpattern object.

Note: When we hover over path function we see first argument is route which is a string and second argument is the view function which returns a -> URLPattern object



1. So, we give two arguments to our path function, the URL which is ‘*playground/hello*’ and reference to say\_hello function(not calling it…) *views.say\_hello*.

This is our playground app’s very own URL configuration.

from django.urls import path

from . import views

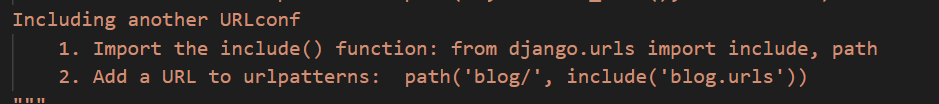
urlpatterns=[

    path("playground/hello/",views.say\_hello)

]

But we need to import this URL configuration into the main URL configuration of this project.

1. For this We need to navigate to urls.py module in our storefront project directory. There are already instructions for including new URL configuration



1. Inside urlpatterns, path function we will mention that any URL that starts with “playground/” should be routed to our “playground.urls” module inside include function

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path("admin/", admin.site.urls),

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

]

Note:

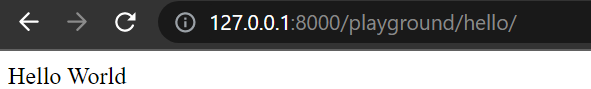
1. If we receive a request from playground/hello, Django knows that all requests that starts with *playground* should be handled by the app inside include function
2. **Always ends routes with /.**
3. Since we already mentioned playground/ in the core urls.py module, we can remove it from playground’s urls.py.

urlpatterns=[

    path("hello/",views.say\_hello)

]

Run the server and see

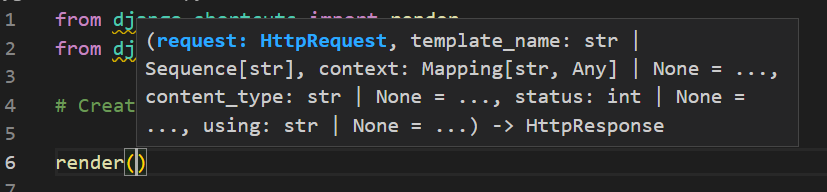


**Using Templates**:

*A Template return HTML content to the client*.

1. Create a folder called *templates* and inside it create a file called *hello.html*. Just write some content there like <h1>Hello World</h1>
2. Back to our views.py module, where we are returning a plain HTTP response, We will use the render function to render a template and return HTML markup to the client.

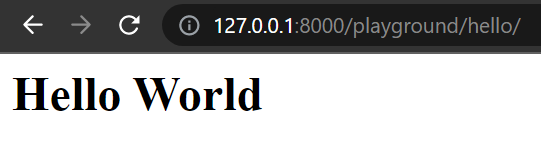
Note: This render function returns a **HTTP response object**, its first argument is **request object** and type of this is HttpRequest and second argument is **template\_name** object of type string.



So…

def say\_hello(request):

    return render(request, "hello.html")



1. Here we are rendering a plain h1 tag with hardcoded value “Hello World”. But we can also dynamically render some value

Notice the third argument of the render method

**context: Mapping[str, Any]**

*This is a* ***context*** *object and* *type of this is a* ***mapping*** *of string to any*. *That means we can pass any mapping object that maps string value to any other type of object*.

1. Here, we can pass a dictionary. Type of key should be a string mapped to a string value, so {‘name’: ‘Himanshu’}

def say\_hello(request):

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

and in hello.html

<h1>HELLO {{name}}</h1>

Name of the Key inside double curly braces.



Note: If we need to write logic like If-ELSE statement inside html file we can use this syntax {%---some code---%}

{% if name %}

<h1>Hello {{name}}</h1>

{% else %}

<h1>Hello World</h1>

{%endif%}

We can easily replace Django default template engine with your preferred template engine.

**Debugging Django applications in VS code**(In Django fundamentals):Need to do

**Using Django Debug Toolbar**:

Installation:

1. Activate virtual environment.
2. Run pipenv install django-debug-toolbar.
3. Add “debug\_toolbar” in the list of installed apps.
4. Add urlpattern in main URLconf module.

urlpatterns = [

    path("admin/", admin.site.urls),

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

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

]

1. Add a middleware, we use middleware to hook into Django’s request-response processing. So add this line

"debug\_toolbar.middleware.DebugToolbarMiddleware",

Inside MIDDLEWARE list of settings.py module.

1. Final step is adding our IP address in the internal IPs setting. For local development we use ‘127.0.0.1’. Add below code in settings module

INTERNAL\_IPS = [

    # ...

    "127.0.0.1",

    # ...

]

1. With all our settings done, we still do not see any debug\_toolbar on our hello.html page. Because toolbar only returns when we return a proper HTML document. So we will add a html and body tag in our hello.html.

<html>

  <body>

    {% if name %}

    <h1>Hello {{name}}</h1>

    {% else %}

    <h1>Hello World</h1>

    {%endif%}

  </body>

</html>

1. Now when we refresh the page we see a toolbar like this…



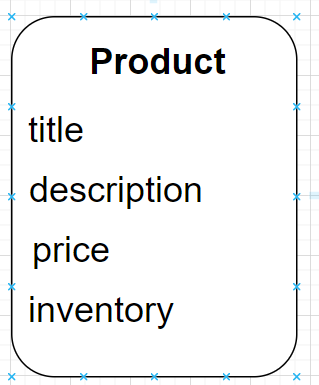
**Section—2: Models**:

**Data Modeling**:

First step in every Django project is figuring out the pieces of data that we want to store.

Take an ecommerce application for example

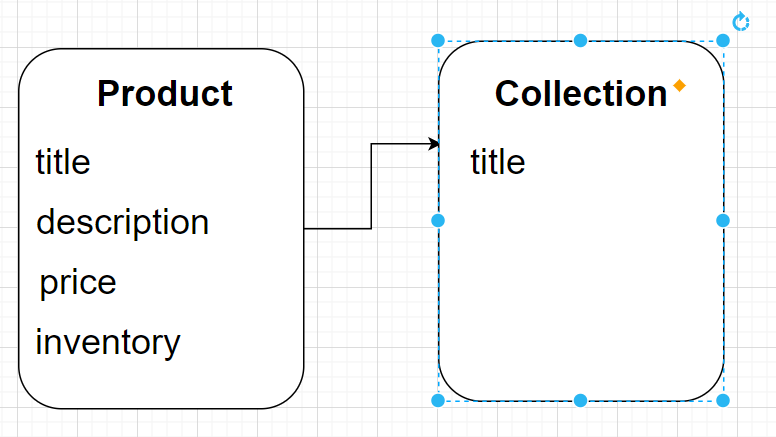
Here we have a concept of *Product* which is just an entity

🡨It has these attributes and many more…

Suppose our products are divided into different categories like shoes, beauty products, fruits and so on.

So we need another entity called *collection* with an attribute called *title*.

Now, we need to add a relationship or an association between these entities. So we can navigate from one end and navigate to another.



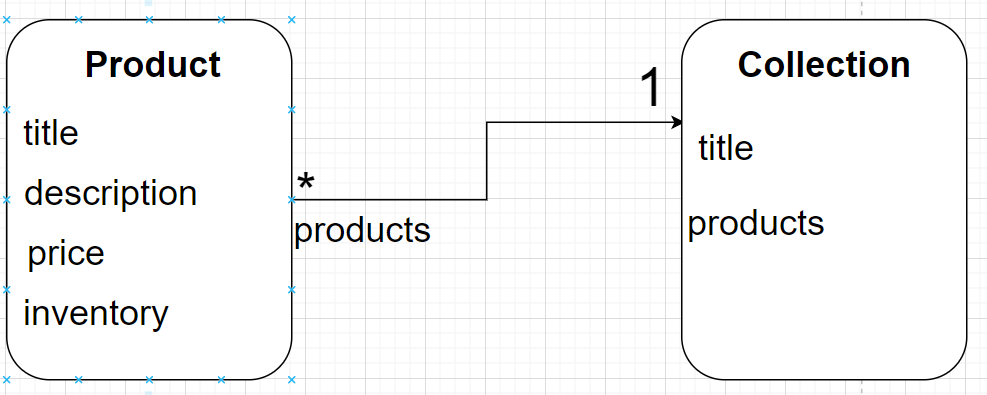
For example, We can get a collection and find all the products in that collection.

Now, Let us assume that in our application, a product can belong to one and only one collection and a collection can have multiple products. This is *one to many relationship*.

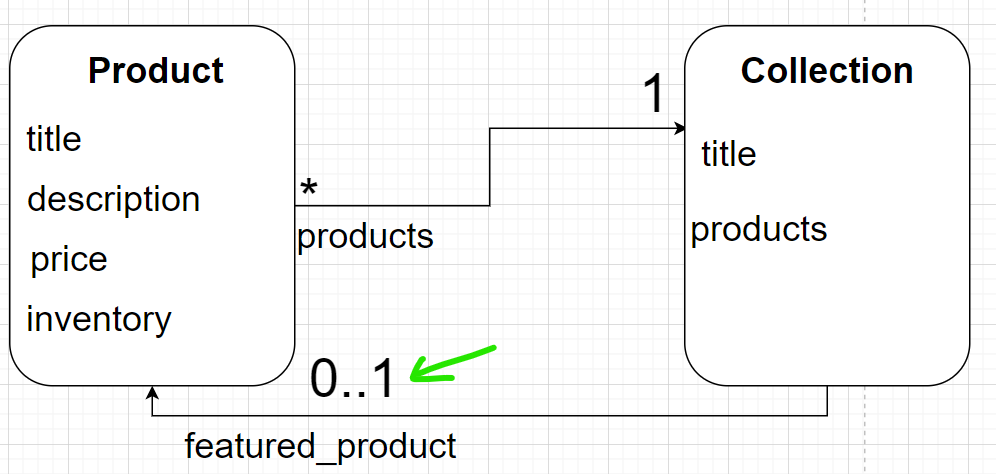
Relationships can be:

1. One to one **1\_\_\_\_\_\_\_\_\_\_1**
2. One to many **1\_\_\_\_\_\_\_\_\_\_\***
3. Many to many **\*\_\_\_\_\_\_\_\_\_\_1**

When we code these entities or classes, in the *collection* class we are going to have an attribute called *products*



Sometimes we can have multiple associations between two entities. Imagine that a collection can optionally have a *featured\_product*, a product whose picture we want to show to the user.

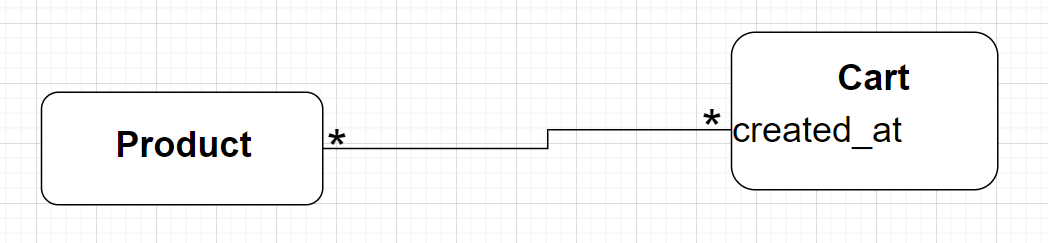


0..1 means a collection can have zero to one product and we are going to call that attribute *featured\_product*.

Note: Every entity has an id attribute and Django automatically create that for us.

**E commerce data model**:

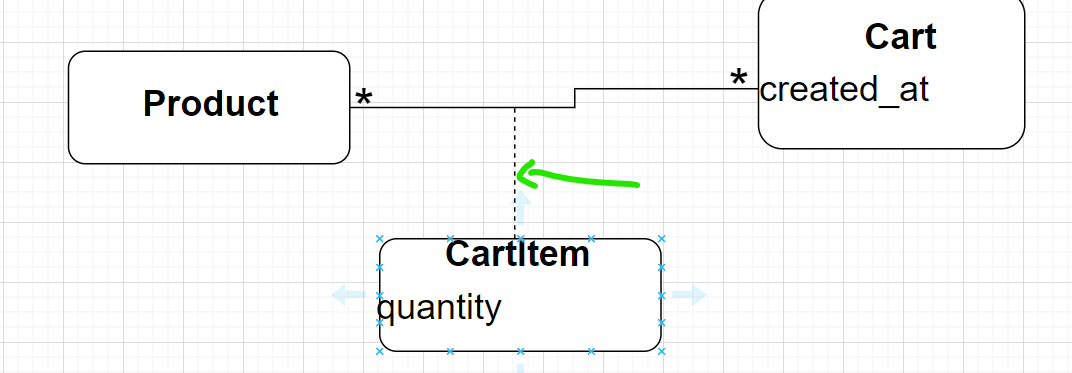
In ecommerce app we have concept of the shopping cart. So we will create an entity, model or a class called *cart* with an attribute called *created\_at*(to know when the cart was created so that we can remove carts that are 30 days old)



Since a cart can have multiple products and a product can be in multiple carts we have many to many relationship.

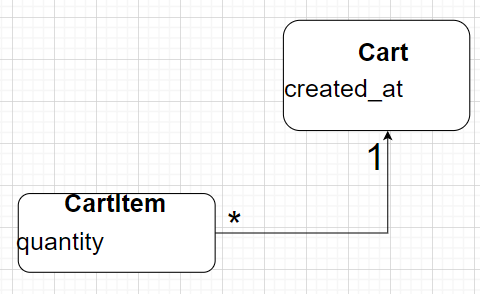
Now, sometimes the relationships between two entities can have attributes. For example if a product is in shopping cart, we need to know how many instances of that product we have inside cart.

So this relationship itself have an attribute called *quantity* which is going to be inside another entity called *CartItem*.

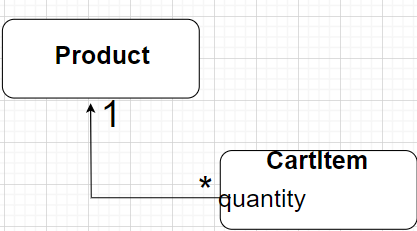


Notice, with dashed line we use to connect *CartItem* to relationship between *Product* and *Cart*. This is called an association class.

Instead of using association class, we can add a relationship between *cart* and *CartItem* because a cart might contain multiple items and an item belongs to one and only one cart.



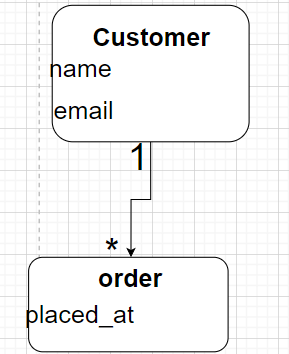
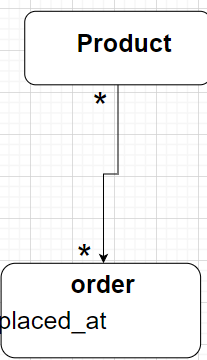
Similarly we can add a relationship between *product* and *CartItems* because each cart item represents a single product and a product might be referenced by many *CartItems*.



So here we have two, one – to – many relationships.

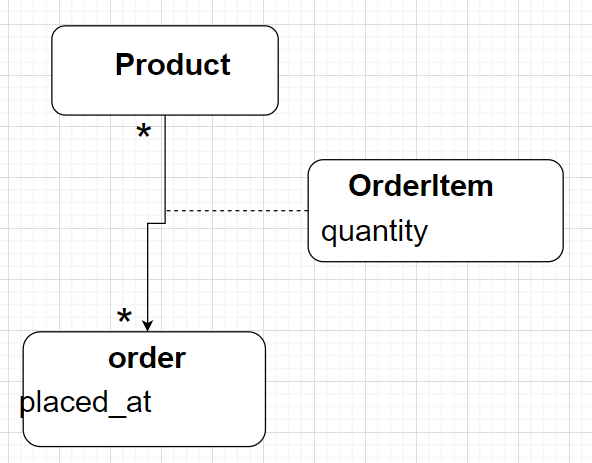
In summary, we can either add a many– to –many relationship with an association class or two, one – to – many relationships. It is a matter of personal preference.

Similarly for our user or customer we will create an entity called customer. Each customer might have several orders but each order belongs to only one customer.

But order can have multiple products and a product can be in multiple orders. So this way we have many – to – many relationship between product and order.

So similar to shopping cart, this relationship itself needs attributes. So here we create a new entity called *OrderItem* with an attribute called *quantity*, that is our association class.



**Organizing models in Apps**:

As we know that a Django project contains one or more apps and each app provides a different functionality. So *each app is going to have its own data model*.

One way to organize our models is to create a single app called store and to drop all the entities here and then bundle/distribute this app by pip. This way anyone can

pipenv install \_\_app\_\_ in their project and get all the models and functionality around them.

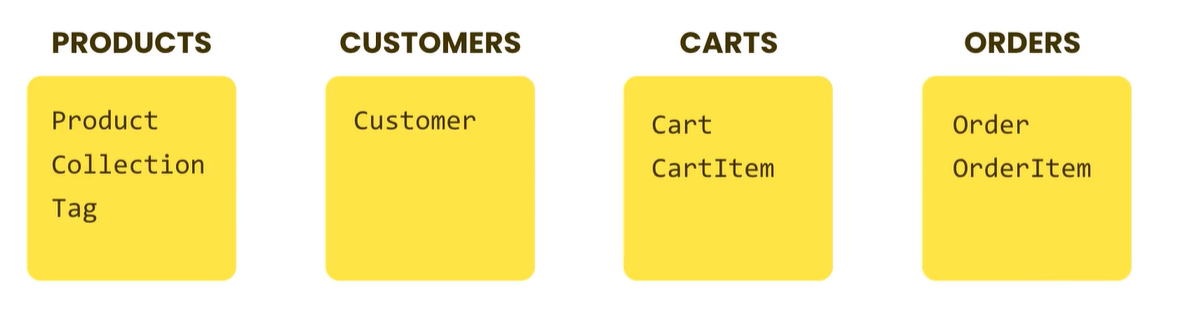
🡨But here is a problem with this method, once the app gets bigger, it start to get bloated with too many models, views and stuff. This is called monolith(large, heavy piece of stone)

Other approach…

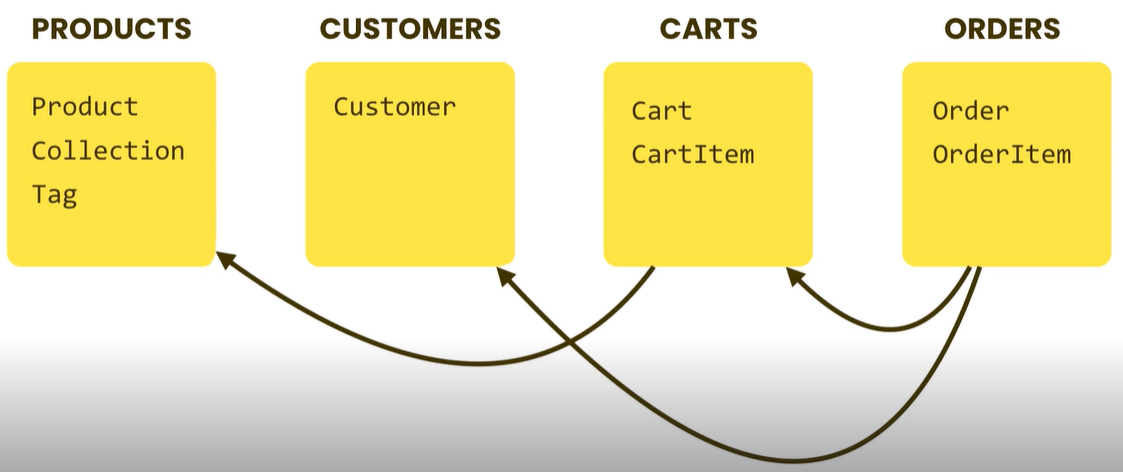
*Each APP should Do one thing and do it well*

-- UNIX philosophy

We can breakdown this project into 4 small apps like this.

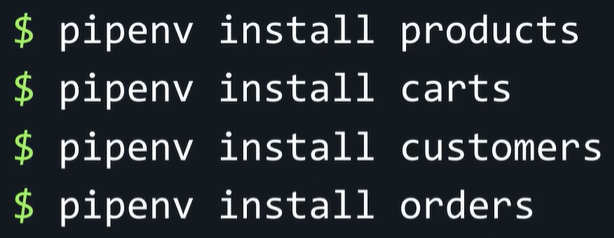


Now look at the coupling/ dependency between these apps.



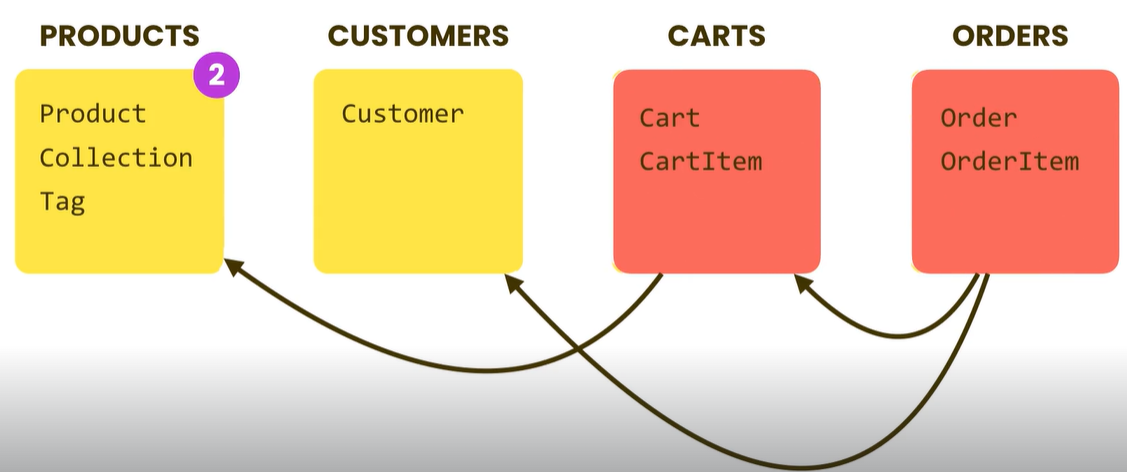
*Orders* app is dependent on *Cart* app which is further dependent on *Products* app.

So next time we are working on an ecommerce project we need to install all these apps one by one



Ideally each app should be self-contained, so we can easily drop it into a new project.

One problem is if we publish a new version of the Products app that might cause a breaking change in the Cart and Orders app. So all related apps might need to be upgraded.



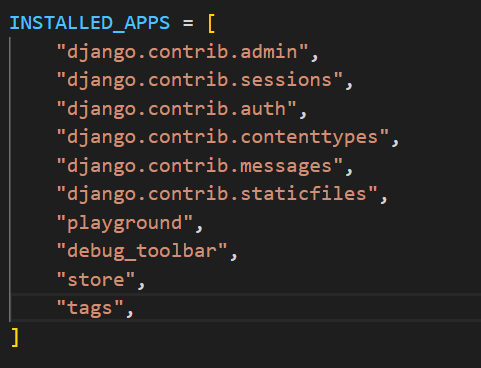
Since Related functionalities always go together, it does not make sense to create them separately (why do we need cart, if we cannot place order or why use product without the cart). So these features are related and should be bundled together

Best approach is *Minimal coupling and High cohesion(Focus). So each app is focused on a specific piece of functionality and equipped with everything needed to fulfill that piece of functionality*.

Now with this approach in mind we create two new apps

python manage.py startapp store

python manage.py startapp tags



Good practice: Every time we create a new app, add it the list of INSTALLED\_APPS *manually*.

**Creating Models**:

Go to the store app and find the models.py module. On the top we have an import statement

from django.db import models

Note: From django.db which is a package, we are importing the models module. In this module we have a bunch of useful classes.

1. Create a new model class called *Product* and have it inherit the Model class in Django.
2. Inside this, we need to define *fields* of this class e.g. title so we set it to an instance of models.Charfield()

Note: Search for *Django field types / Model field* reference and we can see BooleanField, CharField, Datefield etc storing specific type of values as their name suggest.

1. We set CharField’s *field option* max\_length to 255 characters.

Note: All these field types have common *options*, so search for *Field options* e.g. using null (if a field is empty we store it as null in case it is True), db\_index(using it we can create a database index on this field), default(to set the default value) and so on.

class Product(models.Model):

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

    description = models.TextField()

    price = models.DecimalField(max\_digits=6, decimal\_places=2)

    inventory = models.IntegerField()

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

🡪For *price* we are using DecimalField type for storing monetary values(because FloatField has rounding issues). max\_digits and decimal\_places are set as our maximum price will be $9999.99.

🡪last\_update is the current Date time value which is updated every time when we update our *Product* table (auto\_now=True).

Similarly, we can create a model class for our customer as well,

class Customer(models.Model):

    first\_name=models.CharField(max\_length=255)

    last\_name=models.CharField(max\_length=255)

    email=models.EmailField(unique=True)

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

    birth\_date=models.DateField(null=True)

Note: If we do not want Django to create an ID field automatically in our table, we can define one ourselves using primary\_key option and set it to True.

sku=models.CharField(max\_length=10,primary\_key=True)

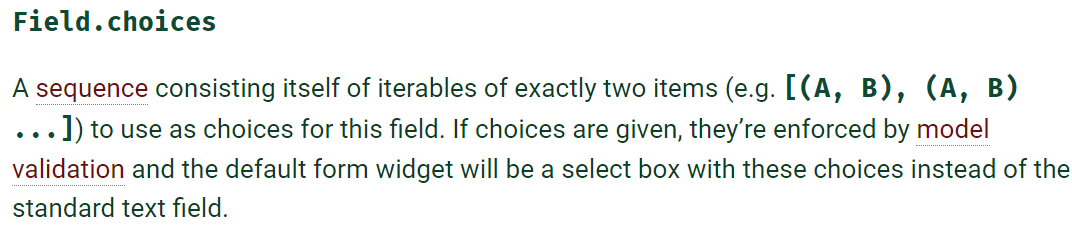
**Choice Fields**:

Sometimes we need to limit the list of values that can be stored in a field.

membership=models.CharField(max\_length=1)

This membership can be either B(bronze), S(silver) or G(gold)

There is one field option which we can use to implement this, choices. With its help we can specify the possible values of a field.



*The first element in each tuple is the actual value to be set in model and second element is the human readable name*.

MEMBERSHIP\_CHOICES = [("B", "Bronze"), ("S", "Silver"), ("G", "Gold")]

🡪Afterwards we can set this list to choices. We can also set a default value of choices.

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    first\_name = models.CharField(max\_length=255)

    last\_name = models.CharField(max\_length=255)

    email = models.EmailField(unique=True)

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

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

And last one is Order class

class Order(models.Model):

    PAYMENT\_STATUS\_PENDING = "P"

    PAYMENT\_STATUS\_COMPLETE = "C"

    PAYMENT\_STATUS\_FAILED = "F"

    PAYMENT\_STATUS\_CHOICES = [

        (PAYMENT\_STATUS\_PENDING, "Pending"),

        (PAYMENT\_STATUS\_COMPLETE, "Complete"),

        (PAYMENT\_STATUS\_FAILED, "Failed"),

    ]

    placed\_at = models.DateTimeField(auto\_now\_add=True)

    payment\_status = models.CharField(

        max\_length=1, choices=PAYMENT\_STATUS\_CHOICES, default=PAYMENT\_STATUS\_PENDING

    )

**Defining one to one relationships**:

In this section, we will see how can we create one to one relationship between two models.

Suppose we create an *Address* class and assume that every customer will have only one address and each address shall belong to only one customer.

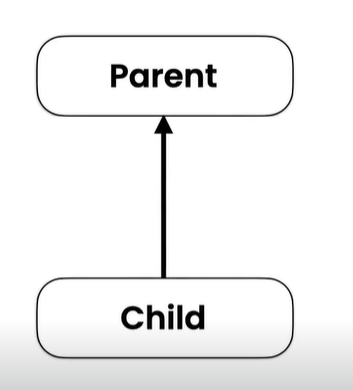
So this is a one to one relationship between *Address* and *Customer*.

class Address(models.Model):

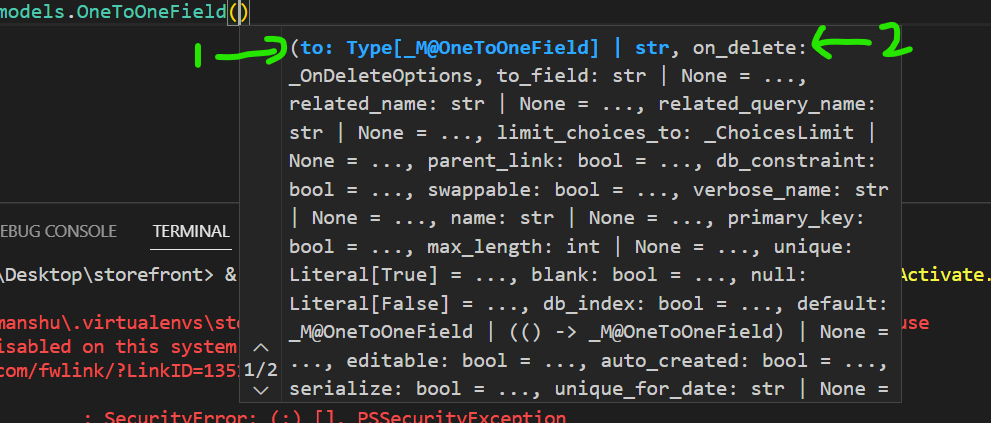
    street = models.CharField(max\_length=255)

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

A database relationship has two ends Parent and Child

A parent should exist before we can store the child, Here *Customer* is the parent and *Address* is the child model class.

So, in the child class we need to specify the parent, so add a new field called customer and set it to model.OneToOneField



This field has a couple of required arguments

🡪First one is type of the parent model, so we specify name of our parent class here.

🡪Second argument is on\_delete by which we specify the delete behavior(what should happen if we delete its parent or Customer here…)

Note: on\_delete has a couple of options

models.Cascade(delete the Customer(parent) and associated address will also be deleted)

models.SET\_NULL(delete the Customer(parent) and associated address will not be deleted)

models.SET\_DEFAULT=(delete the Customer(parent) and associated address will be set to default value)

model.PROTECT=We can prevent deletion, If there is a child associated with this parent, we cannot delete that parent directly (first need to delete the child).

🡪Third argument is primary\_key = True , if we do not set it, Django will create another field called id and will end up with one to many relationship between customer and addresses (Many addresses with same customer).

customer=models.OneToOneField(Customer,on\_delete=models.CASCADE, primary\_key=True)

Since customer is the primary\_key, it will be always unique.

We do not need to define same child in parent class(reverse relationship) because Django automatically creates it for us.

**Defining a One to Many relationship**:

We can change one to one relationship between customer and address in last example into one to many (i.e. One customer can have multiple addresses.)

Change to models.ForeignKey and remove primary\_key attribute to allow duplicate values in the column.

class Address(models.Model):

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

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

    customer = models.ForeignKey(Customer, on\_delete=models.CASCADE)

Note: Always define parent class inside child class not the other way around while setting up relationships.

**Exercise**:

Create this one to many relationships between Parent and child classes

Collection 🡪Product (*one* collection having *many* products)

Customer 🡪Order

Order 🡪 Item

Cart 🡪Item

class Collection(models.Model):

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

class Product(models.Model):

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

    description = models.TextField()

    price = models.DecimalField(max\_digits=6, decimal\_places=2)

    inventory = models.IntegerField()

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

    collection = models.ForeignKey(Collection, on\_delete=models.PROTECT)

We used models.PROTECT here so that if we accidently delete a collection, it will not remove all the products associated with it.

class Cart(models.Model):

    created\_at = models.DateTimeField(auto\_now\_add=True)

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

class CartItem(models.Model):

    cart = models.ForeignKey(Cart, on\_delete=models.CASCADE)

    product = models.ForeignKey(Product, on\_delete=models.CASCADE)

    quantity = models.PositiveSmallIntegerField()

Note: with PositiveSmallIntegerField we prevent negative values being stored in the field.

**Defining Many to Many relationships**:

We are going to introduce a new class called *Promotion* and define many to many relation between promotions and products.

*Products might have many promotions and a promotion can be applied on many products*.

We create a Promotion class,

class Promotion(models.Model):

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

    discount=models.FloatField()We can

can also have start\_date , end\_date and so on.

Note: We can define many to many relationship in either of the classes and Django will automatically create the reverse relationship.

Here we choose to define Promotion class inside Product class because we want to show Products to the customer and associated Promotions with the selected product.

class Product(models.Model):

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

    description = models.TextField()

    price = models.DecimalField(max\_digits=6, decimal\_places=2)

    inventory = models.IntegerField()

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

    collection = models.ForeignKey(Collection, on\_delete=models.PROTECT)

    promotions = models.ManyToManyField(Promotion)

Note: Here we use the plural word promotions because a product will have multiple promotions.

We use ManyToManyField with class name as its first parameter. With this Django will create a reverse relation in the Promotion class as a field name called *product\_set* that returns all the products that a particular promotion is applied to.

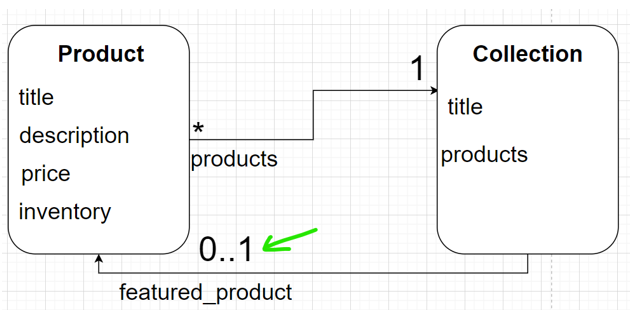
Note: If do not like the name of the field that Django automatically creates using reverse relationship, we can always change it. So the class where we defined ManyToMany field, add one more keyword argument called related\_name and set it to desired name.

promotions = models.ManyToManyField(Promotion,related\_name='products')

So a field with name ‘*products*’ will be created inside Promotions table instead of *product\_set*.

**Resolving circular relationships**:

As we know we can have multiple relations between two models.



e.g. here we have two relationships(1 to many and zero to one ) between two models.

We have implemented one to many relationship between these two models, but in order to implement other relationship, we are going to face a situation called *circular dependency*.

*It happens when two classes depend on each other at the same time*. Product class is going to be dependent on collection class and at the same time vice versa.

In the Product model we have

class Product(models.Model):

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

    description = models.TextField()

    price = models.DecimalField(max\_digits=6, decimal\_places=2)

    inventory = models.IntegerField()

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

    collection = models.ForeignKey(Collection, on\_delete=models.PROTECT)

    promotions = models.ManyToManyField(Promotion)

a collection field *which is a foreign key* to the Collection model.

So here we have a dependency from the product class towards the collection class.

Now to implement the other relationship, we have to go to the collection class and add a new field called *featured\_product* which is going to be a foreign key to the Product class.

Note: If we delete a Product and that product happens to be the featured\_product for a collection, we need to set that field to be null. So on\_delete = models.SET\_NULL.

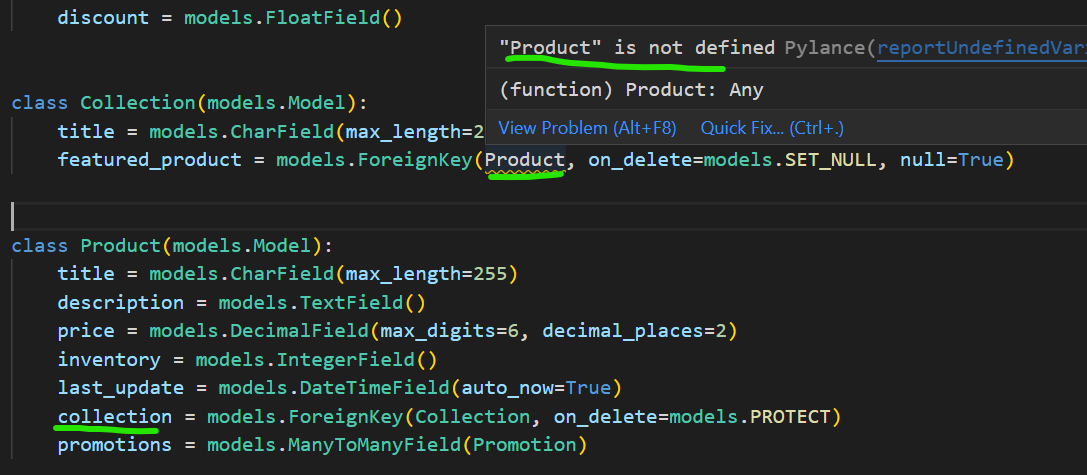
class Collection(models.Model):

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

    featured\_product= models.ForeignKey(Product, on\_delete=models.SET\_NULL, null=True)

Note: In case a field can be null, we also set the argument null=True.

Notice , we are getting this error



‘Product’ is not defined. It is happening because Product class is defined after Collection class. This is where we have circular dependency. *You can notice collection field already defined in the Product model as foreign key*.

One way to do this is wrap the Product class in quotes and pass it as a string in featured\_product field.

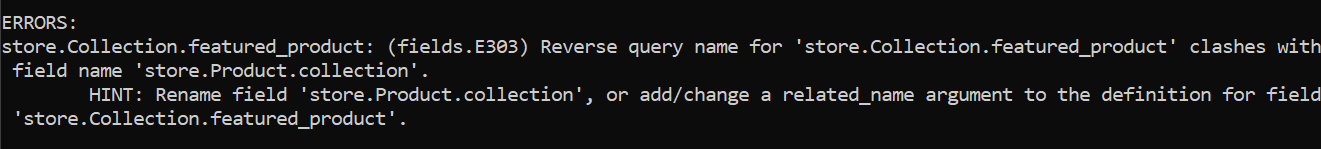
class Collection(models.Model):

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

    featured\_product = models.ForeignKey('Product', on\_delete=models.SET\_NULL, null=True)

But the problem is, if tomorrow we change the Product class name to Product\_XYZ, it will not reflect inside our ForeignKey field.

Next problem with circular dependency



store.Collection.featured\_product: (fields.E303) Reverse query name for 'store.Collection.featured\_product' clashes with field name 'store.Product.collection'.

As we know, Django automatically creates the reverse relationship and this is exactly where the problem happens. Because we have already created a field there for implementing our one to many relation. So Django cannot create reverse relation due to the name crash.

We need to tell Django not to create reverse relation by using related\_name attribute and setting it to “+” like this.

class Collection(models.Model):

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

    featured\_product = models.ForeignKey(

        "Product", on\_delete=models.SET\_NULL, null=True, related\_name="+"

    )

**Generic Relationships**:



We want to design the tags app such that we can reuse it in any project anywhere. *We want to have the ability to tag items like products, videos, articles etc*. That is why we have two models in this app,

*Tag*: Represents an actual tag

*TaggedItem*: Represents a tag applied to a particular item which can be a product, video anything.

Inside models.py of Tags app,

class Tag(models.Model):

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

class TaggedItem(models.Model):

    # What tag applied to what object

    tag = models.ForeignKey(Tag, on\_delete=models.CASCADE)

Nothing new so far in the code, The challenging part is identifying the object that this tag is applied to.

Poor way of implementing it…

We import our Product and hard set this product model inside TaggedItem like this,

from django.db import models

from store.models import Product

class Tag(models.Model):

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

class TaggedItem(models.Model):

    # What tag applied to what object

    tag = models.ForeignKey(Tag, on\_delete=models.CASCADE)

    product=models.ForeignKey(Product)

But hardwiring this app makes this app limits its reusability with different models.

Generic way of identifying any object…

We will create two fields or pieces of information that will help us identifying the object.

Type and ID.

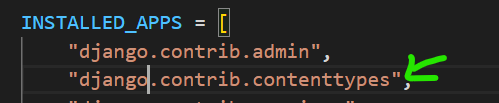
Here *Type*: represents product, video, article

*ID*: Id of that particular object product\_id, video\_id etc.

*Using the Type we can find the table and using ID we will find the record inside that table*. “*In Database term, using these attributes we can find any record in any table”*.

So instead of using concrete model like Product we should use *abstract model* called ContentType which comes with Django.

It comes from the list of installed apps.



Note:Using contenttypes we can create generic relationships within our models.So ContentType is a model that represents the type of object in our application. Import this from the top as from django.contrib.contenttypes.models import ContentType

🡪For ID we can use PositiveIntegerField because we are assuming each table is going to have a primary key and primary keys are generally positive integers.

🡪Lastly we want to see the actual object that we get from type and id, for this we use GenericForeignKey. With this we can read the actual object that tag is applied to.

from django.db import models

from django.contrib.contenttypes.models import ContentType

from django.contrib.contenttypes.fields import GenericForeignKey

# Create your models here.

class Tag(models.Model):

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

class TaggedItem(models.Model):

    # What tag applied to what object

    tag = models.ForeignKey(Tag, on\_delete=models.CASCADE)

    content\_type=models.ForeignKey(ContentType, on\_delete=models.CASCADE)

    object\_id=models.PositiveIntegerField()

    content\_object=GenericForeignKey()

**Exercise**:

Create an app called Likes and in this app we will have model called LikedItem and using an instance of this we can tell what user likes what object.

🡪what user likes what object

🡪user: ForeignKey to user (django.contrib.auth.models)

Solution:

We create a new app called likes and a new model called LikedItem.

Here we have an additional user field which is a foreign key to the User class from django.contrib.auth.models import User

We use models.CASCADE, because if a user is deleted, we need to delete all the items that user has liked to be deleted as well.

from django.contrib.auth.models import User

from django.contrib.contenttypes.models import ContentType

from django.contrib.contenttypes.fields import GenericForeignKey

class LikedItem(models.Model):

    user = models.ForeignKey(User, on\_delete=models.CASCADE)

    content\_type = models.ForeignKey(ContentType, on\_delete=models.CASCADE)

    object\_id = models.PositiveIntegerField()

    content\_object = GenericForeignKey()

rest of the three fields are exactly same like before

content\_type for identifying the type of object user likes.

object\_id for referencing that particular object.

content\_object to read the actual object itself.