No modelo relacional, existem três tipos de associação entre entidades (tabelas) distintas. Estas relações são utilizadas para mapear como as tabelas interagem entre si e podem ser classificadas de três formas: 1-1 (um para um), 1-N (um para muitos) e N-N (muitos para muitos).

Sendo assim, veremos neste artigo como funciona e como implementar os três tipos de relacionamento no Django.

Curso**Django - Fundamentos**

[**Conhecer o curso**](https://www.treinaweb.com.br/curso/django-fundamentos)

**Relacionamento 1-1**

Assim como o próprio nome supõe, o relacionamento 1-1 define que um item de uma entidade só poderá se relacionar com um item de outra entidade. Por exemplo, supondo que temos as entidades Cliente e Endereço que se relacionam de forma 1-1, um cliente só poderá possuir um endereço, que também só pode estar relacionado a um cliente.

Para criar este tipo de relacionamento no Django, no arquivo models.py, precisamos criar um campo que armazene o ID do endereço na entidade cliente do tipo OneToOneField:

Copiar

class Endereco(models.Model):

# Definimos seus atributos

class Cliente(models.Model):

# Definimos seus atributos

endereco = models.OneToOneField(Endereco, on\_delete=models.SET\_NULL, null=True)

Desta forma, a classe Cliente irá armazenar o ID do Endereço que cada cliente possuirá. O código completo das classes Endereço e Cliente podem ser vistas abaixo:

Copiar

class Endereco(models.Model):

rua = models.CharField(max\_length=200, null=False, blank=False)

numero = models.IntegerField(null=False, blank=False)

complemento = models.CharField(max\_length=200, null=False, blank=False)

bairro = models.CharField(max\_length=50, null=False, blank=False)

cidade = models.CharField(max\_length=100, null=False, blank=False)

pais = models.CharField(max\_length=50, null=False, blank=False)

def \_\_str\_\_(self):

return self.rua

class Cliente(models.Model):

SEXO\_CHOICES = (

("F", "Feminino"),

("M", "Masculino"),

("N", "Nenhuma das opções")

)

nome = models.CharField(max\_length=100, null=False, blank=False)

data\_nascimento = models.DateField(null=False, blank=False)

email = models.EmailField(null=False, blank=False)

profissao = models.CharField(max\_length=50, null=False, blank=False)

sexo = models.CharField(max\_length=1, choices=SEXO\_CHOICES, blank=False, null=False)

endereco = models.OneToOneField(Endereco, on\_delete=models.SET\_NULL, null=True)

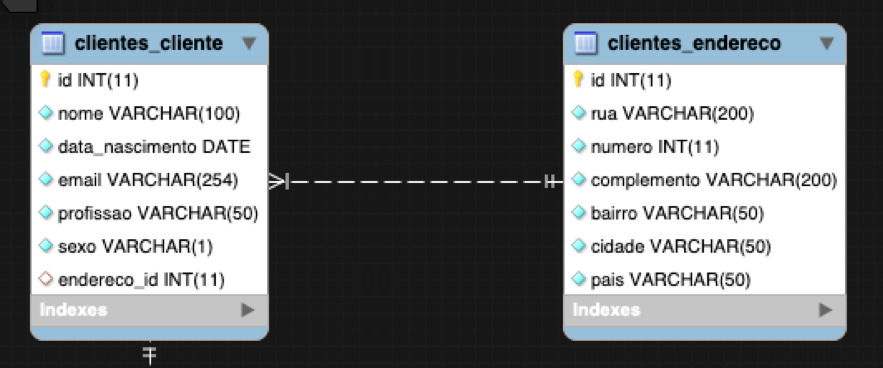
Após definir as classes acima, precisamos criar a migração para efetivar as alterações no banco de dados e, assim, criar as tabelas no banco de dados. Para isso, utilizamos os comandos abaixo:

Copiar

python manage.py makemigrations

python manage.py migrate

Isso fará com que as tabelas sejam criadas no banco de dados com a seguinte estrutura:



**Relacionamento 1N**

O relacionamento 1N determina que um item de uma tabela pode se relacionar com vários itens de uma outra tabela. Continuando o exemplo acima, podemos determinar que um cliente pode realizar diversos pedidos em um restaurante, porém cada pedido só pode possuir um cliente atrelado a ele.

Para criar este relacionamento no Django, utilizamos um campo do tipo ForeignKey, como podemos ver abaixo:

Copiar

class Pedido(models.Model):

cliente = models.ForeignKey("Cliente", on\_delete=models.CASCADE, related\_name='pedidos')

Isso fará com que a tabela Pedidoarmazene o ID do cliente de cada relação. O código completo das classes Pedido e Cliente pode ser visto abaixo:

Copiar

class Pedido(models.Model):

STATUS\_CHOICES = (

("P", "Pedido realizado"),

("F", "Fazendo"),

("E", "Saiu para entrega"),

)

cliente = models.ForeignKey("Cliente", on\_delete=models.CASCADE, related\_name='pedidos')

observacoes = models.CharField(max\_length=300, null=False, blank=False)

data\_pedido = models.DateTimeField(default=timezone.now)

valor = models.FloatField(blank=False, null=False)

status = models.CharField(max\_length=1, choices=STATUS\_CHOICES, blank=False, null=False)

def \_\_str\_\_(self):

return self.cliente.nome

class Cliente(models.Model):

SEXO\_CHOICES = (

("F", "Feminino"),

("M", "Masculino"),

("N", "Nenhuma das opções")

)

nome = models.CharField(max\_length=100, null=False, blank=False)

data\_nascimento = models.DateField(null=False, blank=False)

email = models.EmailField(null=False, blank=False)

profissao = models.CharField(max\_length=50, null=False, blank=False)

sexo = models.CharField(max\_length=1, choices=SEXO\_CHOICES, blank=False, null=False)

endereco = models.OneToOneField(Endereco, on\_delete=models.SET\_NULL, null=True)

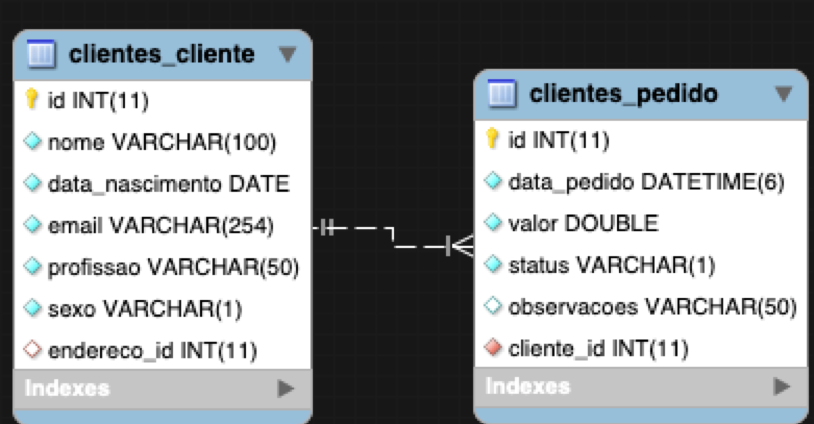
Após definir as classes acima, precisamos criar a migração para efetivar as alterações no banco de dados e, assim, criar as tabelas no banco de dados. Para isso, utilizamos os comandos abaixo:

Copiar

python manage.py makemigrations

python manage.py migrate

Isso fará com que as tabelas sejam criadas no banco de dados com a seguinte estrutura:



Agora, a tabela Pedido armazenará, além de todos os seus atributos, o ID do cliente. Assim, um cliente poderá estar relacionado a vários pedidos, porém um pedido só pode se relacionar com um único cliente.

Curso**Django - Fundamentos**

[**Conhecer o curso**](https://www.treinaweb.com.br/curso/django-fundamentos)

**Relacionamento NN**

O relacionamento NN define que um item de uma tabela pode se relacionar com vários itens de uma outra tabela e vice-versa. Por exemplo, podemos determinar que um pedido pode possuir diversos produtos relacionados a ele, assim como um mesmo produto pode estar relacionado a diversos pedidos diferentes.

Para criar este tipo de relacionamento, utilizamos um campo do tipo ManyToManyField, como podemos ver abaixo:

Copiar

class Produto(models.Model):

nome = models.CharField(max\_length=50, null=False, blank=False)

descricao = models.CharField(max\_length=200, null=False, blank=False)

valor = models.FloatField(null=False, blank=False)

def \_\_str\_\_(self):

return self.nome

class Pedido(models.Model):

STATUS\_CHOICES = (

("P", "Pedido realizado"),

("F", "Fazendo"),

("E", "Saiu para entrega"),

)

cliente = models.ForeignKey("Cliente", on\_delete=models.CASCADE, related\_name='pedidos')

observacoes = models.CharField(max\_length=300, null=False, blank=False)

data\_pedido = models.DateTimeField(default=timezone.now)

valor = models.FloatField(blank=False, null=False)

status = models.CharField(max\_length=1, choices=STATUS\_CHOICES, blank=False, null=False)

produtos = models.ManyToManyField(Produto)

def \_\_str\_\_(self):

return self.cliente.nome

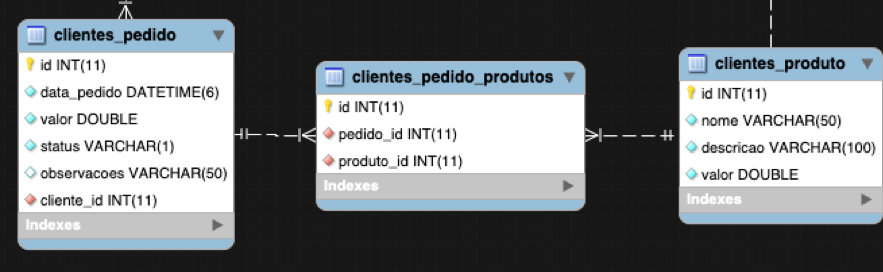
Após definir as classes acima, precisamos criar a migração para efetivar as alterações no banco de dados e, assim, criar as tabelas no banco de dados. Para isso, utilizamos os comandos abaixo:

Copiar

python manage.py makemigrations

python manage.py migrate

Isso fará com que as tabelas sejam criadas no banco de dados com a seguinte estrutura:



Para este tipo de relacionamento, uma tabela “auxiliar” será criada para armazenar os IDs do pedido e do produto que se relaciona.

A tabela clientes\_pedido\_produtosirá armazenar o id do pedido e o id do produto que fazem relação. Desta forma, um pedido pode se relacionar com diferentes produtos e um produto pode estar relacionado a diferentes pedidos.

**Conclusão**

Vimos neste artigo o quão simples é criar os três tipos de relacionamento em bancos de dados relacionais utilizando o ORM do Django. Com isso, agora podemos criar nossas aplicações e estruturar nosso banco de dados da melhor forma possível.

## Django: QuerySets e ORM

O mapeador de objeto-relacional [**ORM**](https://pt.wikipedia.org/wiki/Mapeamento_objeto-relacional) do **Django** é compatível com MySQL, PostgreSQL, SQLite e Oracle.

## O que o ORM do Django faz?

O ORM do Django é fornecido com construções especiais de abstração que podem ser usadas para criar consultas complexas ao banco de dados. Em outras palavras, no lugar de realizar uma ação direta no banco de dados com código SQL por exemplo, utilizamos o ORM como ponte de comunicação entre o banco e a aplicação.

## O que é um QuerySet?

**QuerySet** é um conjunto de ações que serão realizadas no banco de dados, ou seja, podemos criar, buscar, atualizar ou deletar os dados sem escrever a query SQL será executada no banco.

Abaixo, segue uma lista com as principais ações de um QuerySet:

## Ações de um QuerySet

Usaremos como base o seguinte modelo, usado no curso [**Fundamentos de Django da Alura**](https://cursos.alura.com.br/course/fundamentos-django-2):

from django.db import models

class Pessoa(models.Model):

nome = models.CharField(max\_length=200)

email = models.CharField(max\_length=200)

def \_\_str\_\_(self):

return self.nome

Podemos também criar uma instância de um objeto na memória e posteriormente salvar no banco de dados usando o método save:

pessoa = Pessoa(nome='Adriélly', email='adrielly@alura.com')

pessoa.save()

A ação anterior, executa o **insert no banco de dados.** Vimos como criar um objeto na memória primeiro e depois persistir no banco de dados, mas também podemos criar o objeto e persistir no banco de dados em uma única operação usando o método create:

Pessoa.objects.create(nome='Guilherme', email='gui@alura.com')

Já para buscar todas as pessoas cadastradas no banco, usamos o método all:

Pessoa.objects.all()

Caso queira buscar uma instância do banco de dados que contenha o nome Guilherme por exemplo, posso utilizar o método get:

pessoa = Pessoa.objects.get(nome='Guilherme')

O método get permite recuperar um **único objeto do banco de dados** (por exemplo, uma primary key). Se nenhum resultado for retornado pelo banco de dados, esse método gerará uma exceção de **DoesNotExist** indicando que o objeto não existe e, se o banco de dados retornar mais de um resultado, gerará uma exceção **MultipleObjectsReturned**, indicando que existe múltiplos objetos retornados.

Para atualizar o email do Guilherme por exemplo, utilizamos também o método save, porém, dessa vez, **o método executa uma instrução SQL UPDATE**, já que temos uma instância.

pessoa.email = 'guilherme@alura.com'

pessoa.save()

As alterações feitas no objeto não são persistidas no banco de dados até que o método save seja executado.

Você pode excluir determinados resultados do seu QuerySet usando o método delete:

pessoa = Pessoa.objects.get(id=1)

pessoa.delete()

## Filtros

Para os próximos exemplos, considere o seguinte modelo:

from django.db import models

from datetime import datetime

class Receita(models.Model):

nome\_receita = models.CharField(max\_length=200)

modo\_preparo = models.TextField()

date\_receita = models.DateTimeField(default=datetime.now, blank=True)

publicada = models.BooleanField(default=False)

def \_\_str\_\_(self):

return self.nome\_receita

Para filtrar um QuerySet, você pode usar o método filter. Por exemplo, podemos recuperar todas as receitas publicadas, como mostra o código abaixo:

receitas\_publicadas = Receita.objects.filter(publicada=True)

É possível também filtrar por vários campos:

receita\_publicada\_chocolate = Receita.objects.filter(publicada=True, nome\_receita='Bolo de chocolate')

## Ordenação

Também é possível recuperar todos os objetos ordenados com o método order\_by, por exemplo, todas as receitas ordenadas pelo nome:

receitas\_ordenadas = Receita.objects.order\_by('nome\_receita')

A ordem crescente está **implícita**. Você pode indicar a ordem decrescente com um prefixo de sinal negativo, assim:

receitas\_ordenadas = Receita.objects.order\_by('-nome\_receita')

códigos –

**dumpdata**

python manage.py dumpdata –ident 2 > dados. json

organizar os dados dentro de uma pasta.

**Loaddata** – para importar os dados para o nosso banco

# **[Utilizando arquivos Dotenv](https://www.treinaweb.com.br/painel/cursos/django-utilizando-arquivos-dotenv)**

Precisa baixar um pacote. Django-environ

Para isolar as configurações de segurança do ambiente de desenvolvimento.

Criar um arquivo **.env** – mesmo nível do settings.py , e lá iremos por oque não será visível no desenvolvimento.

Configurações – DATABASE\_URL = mysql://toot:admin@localhost:3306/Django\_arquivos\_dotenv

INSTANCIAS – env = environ.Env() e environ.Env.read\_env() – para ler no arquivo settings.py

No databases do settings por : DATABASES ={

‘default’: env.db(‘DATABASE\_URL’)

}

Debug: env.bool(‘DEBUG’) **dentro do env =** DEBUG=True

Secret\_key = env.str(‘SECRET\_KEY’) **dentro do env =** SECRET\_KEY= cod\_chave

# **[Mapeando de banco de dados legado](https://www.treinaweb.com.br/painel/cursos/django-mapeando-de-banco-de-dados-legado)**

**Dev.mysql.com/doc/**  -> documentação com alguns tipos de bancos de dados

Faz mapeamento automático no banco de dados.

Exemplo, pegar banco de dados em outra linguagem.

**Python manage.py inspectdb > app/models.py**

Antes de fazer, tem que configurar o settings.py e o manage.py precisa estar no mesmo nível do mapeamento legado.

A casos que o inspectdb, não aceita alguns dados, e não transforma alguns dados, precisa verificar.

Caso queria algumas tabelas

**Python manage.py inspectdb tablela2(exemplo)** **> app/models.py**