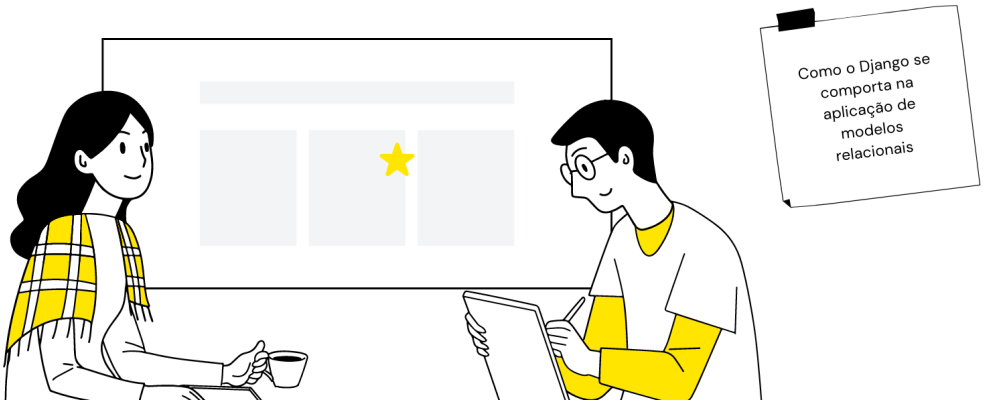
RAISSA AZEVEDO

Django e modelos relacionais



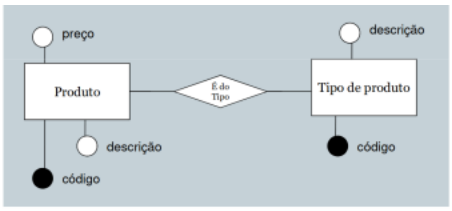
Raíssa Azevedo

Modelos Relacionais no Django Framework

Modelagem conceitual, lógica e Física:

Geralmente as entidades nunca estão sozinhas, é comum que estejam associadas umas com as outras, fornecendo uma descrição mais rica dos elementos do modelo. Um relacionamento pode acontecer entre uma, duas ou várias entidades.

Ex: Relacionamento entre entidade Produto e Tipo Produto:

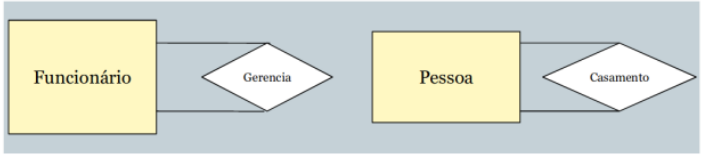


Outros conceitos:

Chave estrangeira – Também conhecido como foreign key ou fk. É um atributo presente em uma entidade que indica um relacionamento e representa a chave primária de uma entidade

Grau de relacionamento – Indica a quantidade de entidades ligadas a um relacionamento. Pode ser: Unário, binário ou Ternário.

Unário – Grau 1, é quando uma entidade se relaciona com ela mesmo.

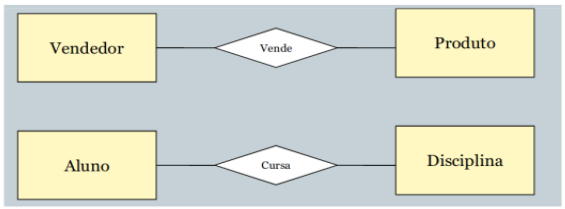


Exemplo onde um funcionário é gerenciado por outro funcionário.

Exemplo onde uma pessoa casa com outra pessoa.

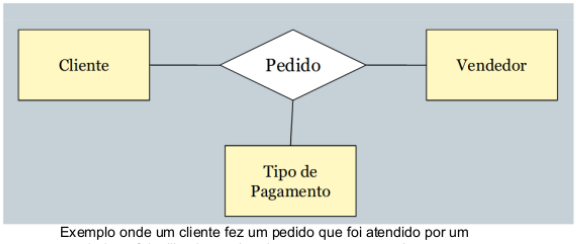
Binário – Grau 2, onde uma entidade se relaciona com outra entidade, é o tipo mais comum de relacionamento.

Ex:



Exemplo onde um vendedor vende produto e um aluno cursa disciplina.

Ternário – Grau 3, onde três entidades estão relacionadas por um mesmo relacionamento.



Exemplo onde um cliente fez um pedido que foi atendido por um vendedor e foi utilizado um tipo de pagamento para tal.

Obs: Dependendo da complexidade por haver mais graus.

PROJETO – DJANGO ORM

O Relacionamento um para um:

No relacionamento one to one (One2One). É um relacionamento pouco comum.

Após fazer as aplicações básicas no settings do projeto do django. Seguir para o models presente no app core.

```
from django.db import models


class Chassi(models.Model):
    numero = models.CharField('Chassi', max_length=16, help_text='Máximo:16 caracteres')

    class Meta:
        verbose_name = "Chassi"
        verbose_name_plural = 'Chassis'

    def __str__(self):
        return self.numero
```

O modelo Chassi implica em cadastrar o Chassi de veículos, que é a estrutura de identificação do veiculo. O Chassi é único.

Em seguida, é necessário criar outra classe, onde ficará o relacionamento de fato.

```
class Carro(models.Model):
    chassi = models.OneToOneField(Chassi, on_delete=models.CASCADE)
    modelo = models.CharField('Modelo', max_length=30, help_text='Máximo:30 caracteres')
    preco = models.DecimalField('Preço', max_digits=8, decimal_places=2)
    descricao = models.TextField('Descrição', max_length=500, help_text='Máximo:500 caracteres')

    class Meta:
        verbose_name = 'Carro'
        verbose_name_plural = 'Carros'

    def __str__(self):
        return self.modelo
```

O relacionamento One to One significa que: Cada carro só pode relacionamento com 1 Chassi. Ou seja, um mesmo chassi também não pode estar relacionado a dois carros. Ou seja, um carro, um chassi (One2One).

No admin.py:

```
from django.contrib import admin

from .models import Chassi, Carro


@admin.register(Chassi)
class ChassiAdmin(admin.ModelAdmin):
    list_display = ['numero']


@admin.register(Carro)
class CarroAdmin(admin.ModelAdmin):
    list_display = ['’modelo’, ‘chassi’, ‘descricao’, ‘preco’]
```

Obs: Toda vez que criar um modelo executar as migrations.

```
python3 manage.py makemigrations
python3 manage.py migrate
```

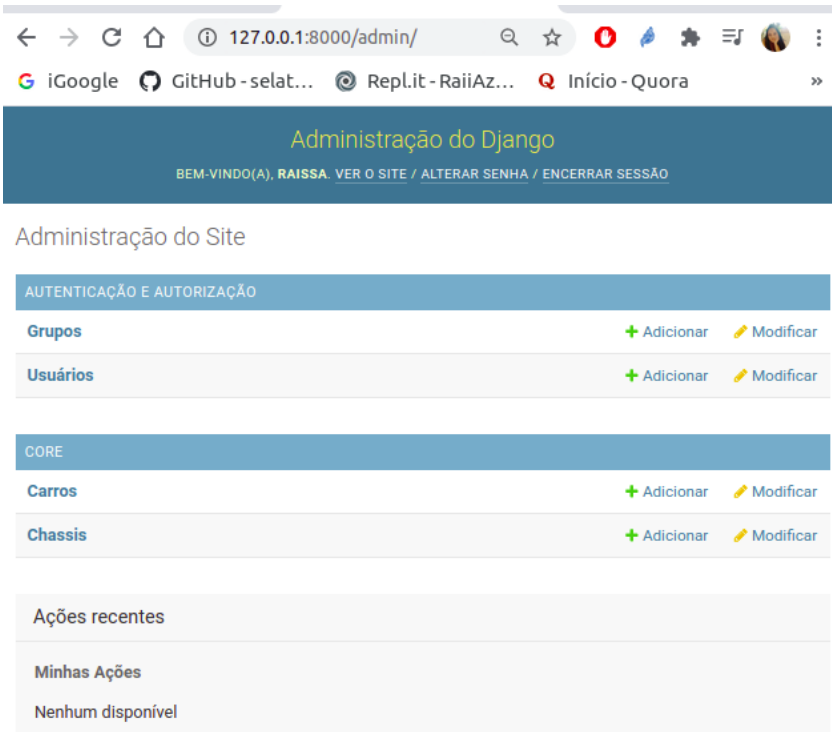
Em seguida executar os comandos para a criação de super usuário para administração do Banco de dados relacional:

```
python3 manage.py createsuperuser
Usuário (leave blank to use 'raissa'):
Endereço de email: rhaii.azevedo@gmail.com
Password:
Password (again):
Superuser created successfully.
```

Em seguida, acessar a área administrativa através do servidor local do Django:

```
python3 manage.py runserver
```

Como não há um template criado ainda. Ficará somente a base administrativa padrão do Django:



Agora todo o modelo está disponível para aplicação.

Ao cadastrar o Chassi, pode-se cadastrar o carro. Entretanto ao tentar cadastrar um novo carro com o mesmo Chassi o Django irá acusar que esse chassi já está cadastrado. Isso significa relacionamento One to One.

No django Shell:

```
 python3 manage.py shell
Python 3.8.5 (default, Jan 27 2021, 15:41:15)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from core.models import Carro
>>> carros = Carro.objects.all()
>>> carros
<QuerySet [<Carro: Corolla Cross XR 2.0>]>
>>> type(Carro)
<class 'django.db.models.base.ModelBase'>
>>>
```

É importante pedir o dir(Carros) para observar todos os atributos que podem ser utilizados e relacionados.

É possível fazer o o query:

```
>>> print(carros.query)
SELECT "core_carro"."id", "core_carro"."chassi_id", "core_carro"."modelo", "core_carro"."preco",
"core_carro"."descricao" FROM "core_carro"
>>>
```

Ao pesquisar o objeto carro:

```
>>> carro = Carro.objects.get(pk=1)
>>> carro
<Carro: Corolla Cross XR 2.0>
>>> type(carro)
<class 'core.models.Carro'>
```

É possível recuperar toda e qualquer informação fornecida através da primary key (pk) do carro através do Sheel.

```
>>> carro
<Carro: Corolla Cross XR 2.0>
>>> carro.modelo
'Corolla Cross XR 2.0'
>>> carro.preco
Decimal('139990.00')
>>> carro.descricao
'Sete airbags, câmera de ré com projeção na central multimídia (só as versões XRE, XRV e XRX têm guias
dinâmicas), controle de estabilidade, controle de tração, assistente de partida em rampa, sensor de estacionamento
traseiro, faróis com acendimento automático e ajuste de altura elétrico, faróis de neblina dianteiros em LED,.'
>>> carro.chassi
<Chassi: 111sjdnsadn22535>
```

Mesmo a classe Chassi sendo construída separada da classe carro nos Models… Ela pôde ser acessada sem fazer o import no Shell devido ao relacionamento 1 para 1.

Chegando ao carro no Shell através do Chassi:

```
>>> from core.models import Chassi
>>> chassi = Chassi.objects.get(id=3)
>>> chassi
<Chassi: 111sjdnsadn22535>
>>> chassi.carro
<Carro: Corolla Cross XR 2.0>
>>> chassi.carro.modelo
'Corolla Cross XR 2.0'
```

Devido ao relacionamento One 2 One é possível acessar os dados tendo o número do Chassi fornecido na classe Chassi. Ou tendo a chave primária do carro fornecido em outra classe nos models.

O Relacionamento One to Many:

Imagine que todo carro possui uma montadora. Ou seja, existe a fabricante.

No models:

```
class Montadora(models.Model):
    nome = models.CharField('Nome', max_length=50)

    class Meta:
        verbose_name = "Nome"
        verbose_name_plural = 'Nomes'

    def __str__(self):
        return self.nome
```

Na class carro. Adicionar a montadora como uma chave estrangeira

```
class Carro(models.Model):
    chassi = models.OneToOneField(Chassi, on_delete=models.CASCADE)
    modelo = models.CharField('Modelo', max_length=30, help_text='Máximo:30 caracteres')
    preco = models.DecimalField('Preço', max_digits=8, decimal_places=2)
    descricao = models.TextField('Descrição', max_length=500, help_text='Máximo:500 caracteres')
    montadora = models.ForeignKey(Montadora, on_delete=models.CASCADE)

class Meta:
    verbose_name = 'Carro'
    verbose_name_plural = 'Carros'

def __str__(self):
    return f'{self.montadora} {self.modelo}'
```

Isso é feito dessa foram (**Foreign Key**). Porque cada carro possui apenas uma montadora. Mas uma montadora pode produzir diversos carros. Por isso ela deve ser apenas uma chave estrangeira dentro da classe de veículos.

Ao mexer nos models a migrations precisa ser criada novamente.

Se houver erros no models, e para corrigir precisa adicionar um defaul. E não quiser propor Blank ou default. O recomentado e apagar o banco de dados:

```
ls
core  db.sqlite3  djangoorm  manage.py  venv
rm db.sqlite3
```

Além de apagar o banco de dados, apagar os arquivos **rm 000*** dentro da pasta migrations também.

Agora, executar as migrations novamente.

Após cadastrar os veículos através do servidor local do Django ou diretamente pelo shell

No Shell:

```
>>> from core.models import Carro
>>> carros = Carro.objects.all()
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
```

Agora é possível pesquisar tudo relacionado ao carro.

```
>>> from core.models import Carro
>>> carros = Carro.objects.all()
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
>>> fit = Carro.objects.get(pk=2)
>>> fit
<Carro: Honda Fit>
>>> honda = fit.montadora
>>> honda
<Montadora: Honda>
```

Chegando no carro através da montadora:

```
>>> from core.models import Montadora
>>> toyota = Montadora.objects.get(pk=1)
>>> toyota
<Montadora: Toyota>
>>> carros = toyota.carro_set.all()
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>]>
```

Como um montadora pode ter vários carros. Nas suas configurações ela vai ter o **carro_set** que consiste em uma lista com todos seus carros.

Isso significa a relação One to many.

O relacionamento muitos para muitos:

Existe um model carro. E cada carro pode ter vários motoristas. Como uma espécie de aluguel de carros. Portanto irá ser feita a vinculação do model carros com um model User.

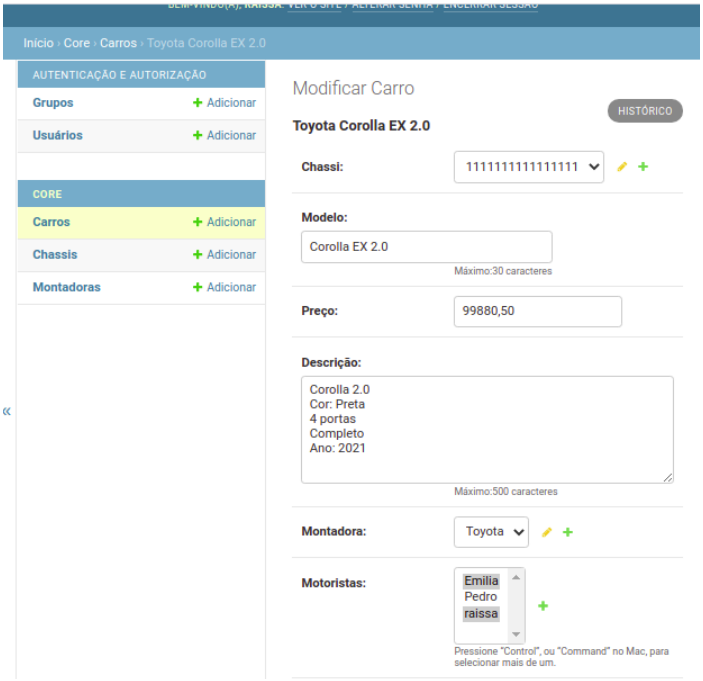
Porque um carro pode ser dirigido por vários motoristas, e um motorista pode dirigir vários carros.

```
from django.contrib.auth import get_user_model

class Carro(models.Model):
    chassi = models.OneToOneField(Chassi, on_delete=models.CASCADE)
    modelo = models.CharField('Modelo', max_length=30, help_text='Máximo:30 caracteres')
    preco = models.DecimalField('Preço', max_digits=8, decimal_places=2)
    descricao = models.TextField('Descrição', max_length=500, help_text='Máximo:500 caracteres')
    montadora = models.ForeignKey(Montadora, on_delete=models.CASCADE)
    motoristas = models.ManyToManyField(get_user_model())
```

Os usuários serão registrados na aba usuários da administração do Django:

E a seleção para motoristas de cada carro fica dentro das especificações de cada carro:



Assim é feito o controle de motoristas, os motoristas autorizado ficam com a coloração acinzentada no menu de descrição de cada veículo.

No Shell:

```
>>> from core.models import Carro
>>> carros = Carro.objects.all()
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
>>> carro1 = carros.first()
>>> carro1
<Carro: Toyota Corolla EX 2.0>
>>> motors = carro1.motoristas.all()
>>> motors
<QuerySet [<User: raissa>, <User: Emilia>]>
```

Qual é, e quem são os motoristas do carro 2?

```
>>> carro2 = carros.last()
>>> carro2
<Carro: Honda Fit>
>>> motors2 = carro2.motoristas.all()
>>> motors2
<QuerySet [<User: Pedro>, <User: Emilia>]>
```

Isso é o relacionamento **many to many**, onde é possível ter vários carros com vários motoristas.

A medida que a complexidade de cada modelo relacional vai aumentando. Aumenta também a complexidade do query desse modelo.

```
>>> print(motors2.query)
SELECT "auth_user"."id", "auth_user"."password", "auth_user"."last_login", "auth_user"."is_superuser",
"auth_user"."username", "auth_user"."first_name", "auth_user"."last_name", "auth_user"."email",
"auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined" FROM "auth_user" INNER JOIN
"core_carro_motoristas" ON ("auth_user"."id" = "core_carro_motoristas"."user_id") WHERE
"core_carro_motoristas"."carro_id" = 2
```

E se quiser saber a lista de carros que uma determinada lista de usuários dirige?

```
>>> motors
<QuerySet [<User: raissa>, <User: Emilia>]>
>>> m1 = motors.first()
>>> m1
<User: raissa>
>>> carros = Carro.objects.filter(motoristas=m1)
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>]>
>>> print(carros.query)
SELECT "core_carro"."id", "core_carro"."chassi_id", "core_carro"."modelo", "core_carro"."preco",
"core_carro"."descricao", "core_carro"."montadora_id" FROM "core_carro" INNER JOIN "core_carro_motoristas"
ON ("core_carro"."id" = "core_carro_motoristas"."carro_id") WHERE "core_carro_motoristas"."user_id" = 1
```

E se quiser saber qual a lsta de carros dirigida pela lista inteira:

```
>>> motors
<QuerySet [<User: raissa>, <User: Emilia>]>
>>> carros = Carro.objects.filter(motoristas__in = motors)
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
```

Há uma repetição no nome dos carros, porque as vezes vários motoristas dirigem um mesmo carro.
Para filtrar:

```
>>> carros = Carro.objects.filter(motoristas__in = motors).distinct()
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
>>>
```

Pronto agora o Shell exibe a lista sem repetição por causa do comando **distinct()** no final do comando de exibição dos carros dirigidos pela lista 1 de motoristas.

Aproveitando os recurso do Django models:

O que acontece com o Banco de Dados se deletar uma montadora?

Ou seja, o Django pede pela confirmação de deleção, e indica o que será perdido ao deletar uma das montadoras salvas no sistema. Que consiste em: 1 montadora, 1 carros e 2 relacionamentos de usuários que estão autorizados a dirigir.
Ou seja há uma remoção em cascata.

Isso ocorre porque:

```
montadora = models.ForeignKey(Montadora, on_delete=models.CASCADE)
```

Ao indicar a montadora no models.py. Colocamos a função **on_delete =** models.CASCADE. Isso orienta o django a fazer a remoção em cascada em tudo que tiver associado a montadora.

O on_delete em CASCADE deve ser adicionado tanto em Foreign Key quanto em many to many porque, se algo está vinculado a um objeto e esse objeto(montadora) deixa de existir, todo o vinculo também precisa ser removido.
Porque o carro depende da foreign key e da montadora como seus vínculos de identificação.

Ou seja, se você deleta o Chassi, o carro também precisa de deixar de existir dentro do sistema.