Django

Persistência – Parte 1

Ely – elydasilvamiranda [at] gmail.com

Fontes

- Esses slides são baseados na documentação oficial do Django em sua versão 1.11:

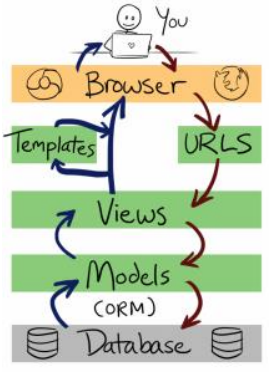
  https://docs.djangoproject.com/en/1.11/

- Para auxiliar o acompanhamento:
  - Crie um projeto chamado myproject;
  - Uma app chamada myapp;
  - Crie um banco também.

Padrão MTV

- Modelo:
  - Responsável pela persistência e recuperação de dados;
  - Possui uma API de mapeamento objeto-relacional (ORM);
- Template:
  - tem o objetivo de exibir dados em um formato padrão, tipicamente HTML;
- Visão:
  - trata o esquema de requisições e respostas efetuando processamentos necessários;
  - Faz a integração entre modelo e template

Padrão MTV



http://blog.easylearning.guru/implementing-mtv-model-in-python-django/#prettyPhoto

Models (relembrando)

- Modelo em Django é uma classe básica, uma entidade do sistema;
- Tipicamente são classes que possuem atributos a serem persistidos;
- Fazem parte do modelo de objetos do sistema.
- São classes que ficam armazenadas em meios de persistência como banco de dados;
- Exemplos: Conta, Pessoa, Perfil, Livro, Processo, Produto;
- No Django, colocamos essas classes no arquivo **models.py**.

5

Models

- Cada model é uma classe em Python que herda de django.db.models.Model;
- Cada atributo representa um campo em uma tabela;
- O Django fornece uma API madura para acesso e manipulação de dados;
- Caso essa API não seja suficiente, podem ser elaboradas queries para buscas avançadas:
  - https://docs.djangoproject.com/en/1.11/topics/db/queries/

6

Classe/tabela

- Cada classe é mapeada para uma tabela em banco após executadas as migrações;
- A classe pessoa abaixo:

```
from django.db import models

class Person(models.Model):

    first_name = models.CharField(max_length=30)

    last_name = models.CharField(max_length=30)
```

- Equivale à seguinte entidade de banco (PostgreSQL):

```
CREATE TABLE myapp_person (
 "id" serial NOT NULL PRIMARY KEY,
 "first_name" varchar(30) NOT NULL,
 "last_name" varchar(30) NOT NULL
);
```

7

Classe/tabela

- A tabela gerada possui nome mysite_person:
  - A princípio, o nome da tabela possui o nome da aplicação + o nome da entidade;
  - Pode-se alterar esse padrão:
    - https://docs.djangoproject.com/en/1.11/ref/models/options/#table-names
- É adicionado um id como chave primária e auto-incremento;
  - Esse comportamento também pode ser alterado:
    - https://docs.djangoproject.com/en/1.11/topics/db/models/#automatic-primary-key-fields

8

Campos

- Os atributos são mapeados em campos de tabela;
- É necessário fornecer algumas informações sobre os campos:
  - Tipo: tipicamente são classes que herdam da classe django.db.Field. Ex: CharField, DateField;
  - Opções de validação:
    - Se o campo aceita valores nulos, tamanho máximo;
    - As opções são passadas como parâmetros no construtor do campo;

Opções

- null:
  - Se True, o Django aceita valores nulos;
  - O padrão é False;
- choices:
  - Um conjunto de tuplas com valores pré-definidos;
  - Ideal para campos select em formulários HTML;

```
class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)
```


Nota: rode os comandos makemigrations e migrate

Opções

- choices (continuação):
  - Em um campo que possui uma opção choices, o primeiro valor da tupla é salvo em banco;
  - É possível obter o segundo valor para exibição usando o método get_xxx_display()
  - Ex:

```
>>> from myapp.models import Person
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```


Opções

- primary_key:
  - Se True, define um campo como chave primária;
  - Esse campo é somente leitura, caso ele seja alterado, um novo objeto é criado;

```
class Fruit(models.Model):
    name = models.CharField(max_length=100, primary_key=True)
```

```
>>> fruit = Fruit.objects.create(name='Apple')
>>> fruit.name = 'Pear'
>>> fruit.save()
>>> Fruit.objects.values_list('name', flat=True)
['Apple', 'Pear']
```

Nota: rode os comandos makemigrations e migrate

Opções

- default:
  - O valor padrão caso o campo não seja preenchido;
- help_text:
  - O texto descritivo sobre o campo a ser exibido em algum componente visual;
- unique:
  - Define que o campo possui valor único em toda a tabela;
- db_column:
  - Representa o nome da coluna no banco de dados. Caso não seja fornecido, será o nome do atributo;

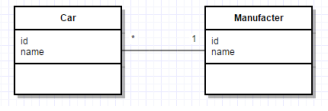
Para mais definições de opções:
https://docs.djangoproject.com/en/1.11/ref/models/fields/#common-model-field-options

Relacionamentos

- O poder dos bancos relacionais reside nos relacionamentos de tabelas as outras;
- Django fornece meios de se definir os 3 tipos de relacionamentos:
  - Um para muitos ou muitos para um:
    - Definido através de um campo do tipo ForeignKey;
  - Um para um:
    - Definido com um campo OneToOneField;
  - Muitos para muitos:
    - Definido por um campo ManyToManyField.

Muitos para um

- Para este relacionamento, cria-se um atributo de classe do tipo django.db.models.ForeignKey;
- Feito do lado do relacionamento "muitos";
- Requer um argumento representando a qual classe se relaciona;
- Pode-se especificar o comportamento em relação às exclusões (on_delete): cascata, setar nulo...
- Também se pode especificar o nome de um atributo do lado "um" do relacionamento.



https://docs.djangoproject.com/en/1.11/ref/models/fields/#ref-foreignkey

Muitos para um

- Exemplo:

```
class Manufacter(models.Model):
    name = models.CharField(max_length=50)

class Car(models.Model):
    name = models.CharField(max_length=50)
    manufacter = models.ForeignKey(Manufacter,
                                   on_delete = models.CASCADE,
                                   related_name = "cars")
```

- Execute as migrações...

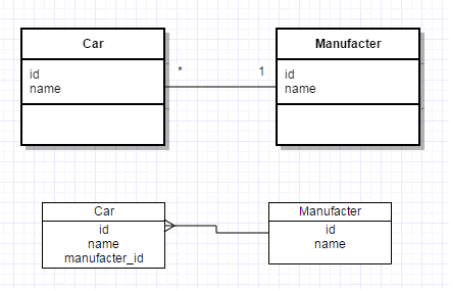
Muitos para um

- Exemplo:
  - No prompt:

```
>>> from myapp.models import Car, Manufacter
>>> car = Car(name='celta')
>>> manufacter = Manufacter(name = 'GM')
>>> manufacter.save()
>>> car.manufacter = manufacter
>>> car.save()
>>> m = Manufacter.objects.get(id=1)
>>> m.cars.all()
<QuerySet [<Car: Car object>]>
>>> carros = m.cars.all()
>>> carros[0].name
u'celta'
```

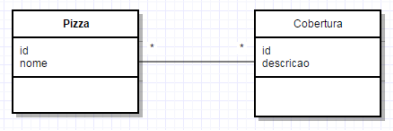

Muitos para um

- Equivalente em MER



Muitos para muitos

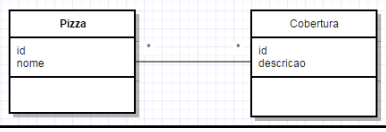
- Para este relacionamento, usa-se um atributo do tipo django.db.models.ManyToManyField;
- Esse atributo, obviamente, é uma lista;
- Requer também um argumento representando a qual classe se relaciona;
- Assim como no relacionamento muitos para um, pode-se definir a bidirecionalidade;



https://docs.djangoproject.com/en/1.11/ref/models/fields/#django.db.models.ManyToManyField

Muitos para muitos

- Deve-se escolher um dos lados do relacionamento para colocar o campo, nunca em ambos;
- Em qual dos lados colocá-lo?
  - Deve-se escolher o elemento que seria "criado" em uma tela envolvendo as duas entidades;
  - No exemplo abaixo:
    - Ao criarmos uma nova pizza, adicionaríamos/excluiríamos as coberutras;
    - Nunca o contrário, pois pizza é o lado "mais forte".

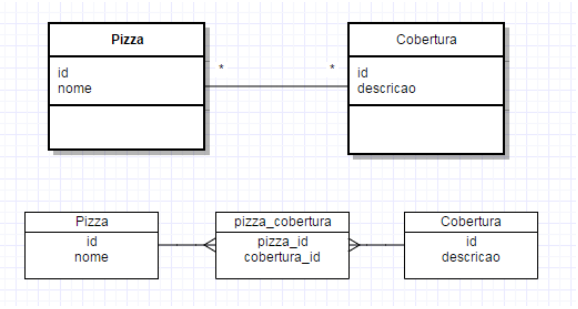


Muitos para muitos

```
class Cobertura(models.Model):
    descricao = models.CharField(max_length=50)

    def __str__(self):
      return self.descricao


class Pizza(models.Model):
    nome = models.CharField(max_length=50)
    coberturas = models.ManyToManyField(Cobertura)
```


Muitos para muitos

- Equivalente em um MER:

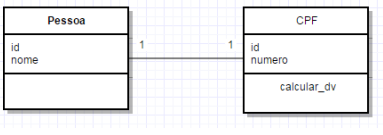


Muitos para muitos

```
>>> from myapp.models import *
>>> p = Pizza(nome = 'Calabresa')
>>> c1 = Cobertura(descricao = 'Molho de tomate')
>>> c2 = Cobertura(descricao = 'Mozarela')
>>> c3 = Cobertura(descricao = 'Linguica calabresa')
>>> coberturas = [c1,c2,c3]
>>> p.save()
>>> c1.save()
>>> c2.save()
>>> c3.save()
>>> coberturas = [c1,c2,c3]
>>> p.coberturas = coberturas
>>> p.save()
>>> p1 = Pizza.objects.get(id = 1)
>>> p1.coberturas.all()
<QuerySet [<Cobertura: Molho de tomate>, <Cobertura: Mozarela>,
<Cobertura: Linguica calabresa>]>
```


Um para um

- Para este relacionamento, usa-se um atributo do tipo django.db.models.OneToOneField;
- Requer também um argumento representando a qual classe se relaciona;
- Assim como no relacionamento muitos para um, pode-se definir a bidirecionalidade;
- No banco, a chave estrangeira ficará na entidade que tiver o atributo.



https://docs.djangoproject.com/en/1.11/ref/models/fields/#django.db.models.OneToOneField

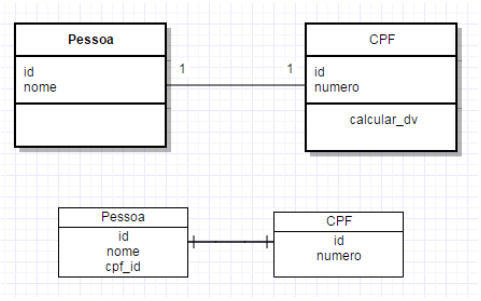
Um para um

- Exemplo:

```
class CPF(models.Model):
    numero = models.CharField(max_length = 9)

    def calcular_dv(self):
        return '00'
      def __str__(self):
        return self.numero + '-' + self.calcular_dv()

class PessoaFisica(models.Model):
    nome = models.CharField(max_length=100)
    cpf  = models.OneToOneField(CPF,
                                related_name = 'pessoa_fisica')

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


Um para um

- Equivalente em um MER:



Um para um

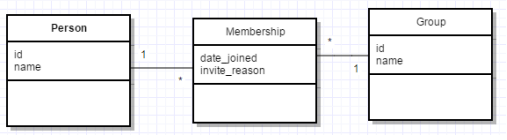
```
>>> from myapp.models import *
>>> cpf_ = CPF(numero = '888')
>>> cpf_.save()
>>> p = PessoaFisica(nome = 'ely', cpf = cpf_)
>>> p.save()
>>> p1 = PessoaFisica.objects.get(id = 1)
>>> p1.nome
u'ely'
>>> p1.cpf
<CPF: 888-00>
>>> p1.cpf.pessoa_fisica
<PessoaFisica: PessoaFisica object>
>>> p1.cpf.pessoa_fisica.nome
u'ely'
```


Muitos para muitos avançado

- Muitos para muitos simples:
  - Pizzas x coberturas;
  - Personagem x filmes;
  - Empregado x habilidades;
- Dependendo da abstração:
  - Não há "campos extras" na entidade associativa/modelo intermediário ;
  - ManyToManyField padrão resolve;

Muitos para muitos avançado

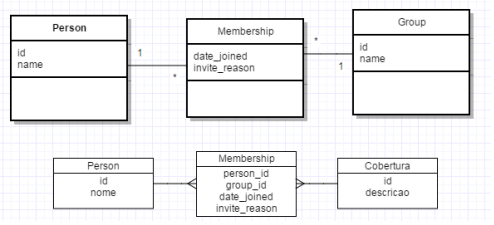
- Muitos para muitos com campos a mais:
  - Há campos pertencentes à entidade associativa;
  - Ex: Pessoas x Grupos, onde a associação de uma pessoa a um grupo pode ter:
    - Data que a pessoa passou a fazer parte do grupo;
    - Motivo do convite.



https://docs.djangoproject.com/en/1.11/ref/models/fields/#manytomany-arguments
https://docs.djangoproject.com/en/1.11/topics/db/models/#intermediary-manytomany

Muitos para muitos avançado

- Equivalente em MER:



Muitos para muitos avançado

- Solução com Django:
  - Haverá 3 classes: os "lados muitos" e um modelo intermediário;
  - Uma delas terá o relacionamento muitos para muitos, e um novo parâmetro:
    - through: aponta para o modelo que será o modelo intermediário;
  - A entidade associativa terá deverá ter relacionamentos ForeignKey

Exemplo

```
class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __str__(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

Exemplo

```
>>> from myapp.models import *
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")

>>> from datetime import date
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()

>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>]>
>>> ringo.group_set.all()
<QuerySet [<Group: The Beatles>]>

>>> m2 = Membership.objects.create(person=paul, group=beatles
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.")
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>, <Person: Paul McCartney>]>
```


Muitos para muitos avançado

- Um modelo intermediário a princípio deve ter apenas um atributo de cada lado:
  - E se Membership tivesse o usuário que fez o convite?
  - Como o Django descobre se person ou inviter é um dos lados "muito" da relação?

```
class Membership(models.Model):
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    inviter = models.ForeignKey(
        Person,
        on_delete=models.CASCADE,
        related_name="membership_invites",
    )
    invite_reason = models.CharField(max_length=64)
```


Muitos para muitos

- A solução é especificar no ManyToManyField que campos definem o relacionamento:

```
class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(
        Person,
        through='Membership',
        through_fields=('group', 'person'),
    )
```


Auto relacionamento

- Um caso especial de muitos para muitos;
- Usa-se a palavra self para especificar que se trata de relacionamento entre mesma entidades;
- Por padrão, auto relacionamentos são simétricos:
  - Se uma X é amigo e Y, Y é amigo de X;
  - Caso não haja simetria, deve-se especificar o parâmetro simetric para False.

```
class Perfil(models.Model):
    friends = models.ManyToManyField("self")
```

Métodos para relacionamentos

- As coleções de um relacionamentos, o lado "muitos" possuem métodos de manipulação:
  - create(), add(), remove(), clear() e set()

```
class Cobertura(models.Model):
    # ...
    pass
class Pizza(models.Model):
    coberturas = models.ManyToManyField(Cobertura)
```

- Os métodos citados estão disponíveis para: cobertura.pizza_set e em pizza.coberturas;
- O atributo pizza_set é disponibilizado caso não se use o parâmetro related_name.

https://docs.djangoproject.com/en/1.11/ref/models/relations/

create

- Cria um novo objeto, salva-o e relaciona-o com um objeto;
- Retorna o objeto recém-criado.

```
>>> from myapp.models import *
>>> p = Pizza(nome = 'mexicana')
>>> p.save()
>>> c = p.cobertura_set.create(
...     descricao='pimenta')
>>> p.cobertura_set.all()[0].descricao
pimenta
# Não é necessário chamar e.save()
```

Tente criar a mesma versão de código sem usar o método create

add/remove/clear/set

- add: adiciona um objeto a uma coleção:

```
>>> from myapp.models import *
>>> p = Pizza.objects.get(id = 1)
>>> c = Cobertura.objects.get(id = 3)
>>> p.cobertura_set.add(c)
```

- remove: retira um objeto de uma coleção

```
>>> p.cobertura_set.remove(c)
```

- clear: limpa todos os elementos de uma coleção

```
>>>  p.cobertura_set.clear()
```

- set: adiciona elementos a uma coleção:

```
>>> c2 = Cobertura.objects.get(id = 2)
>>> c3 = Cobertura.objects.get(id = 3)
>>> c4 = Cobertura.objects.get(id = 4)
>>> coberturas = [c2, c3, c4]
>>> p.cobertura_set.set(coberturas)
```


Atividade 1

- Considerando os modelos abaixo:

```
class Blog(models.Model):
    name = models.CharField(max_length = 50)

class Entry(models.Model):
    headline = models.CharField(max_length = 60)
    body_text = models.CharField(max_length = 255)
    pub_date = models.DateTimeField()
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
```

  - Gere os esquemas;
  - Escreva scripts que testem os métodos create(), remove(), add(), clear e set()
  - Algum dos métodos "não funcionou"? Justifique.

https://docs.djangoproject.com/en/1.11/ref/models/relations/

Django

Persistência – Parte 1

Ely – elydasilvamiranda [at] gmail.com