

Django

02 Modelos

Professor: Ritomar Torquato

02 Modelos

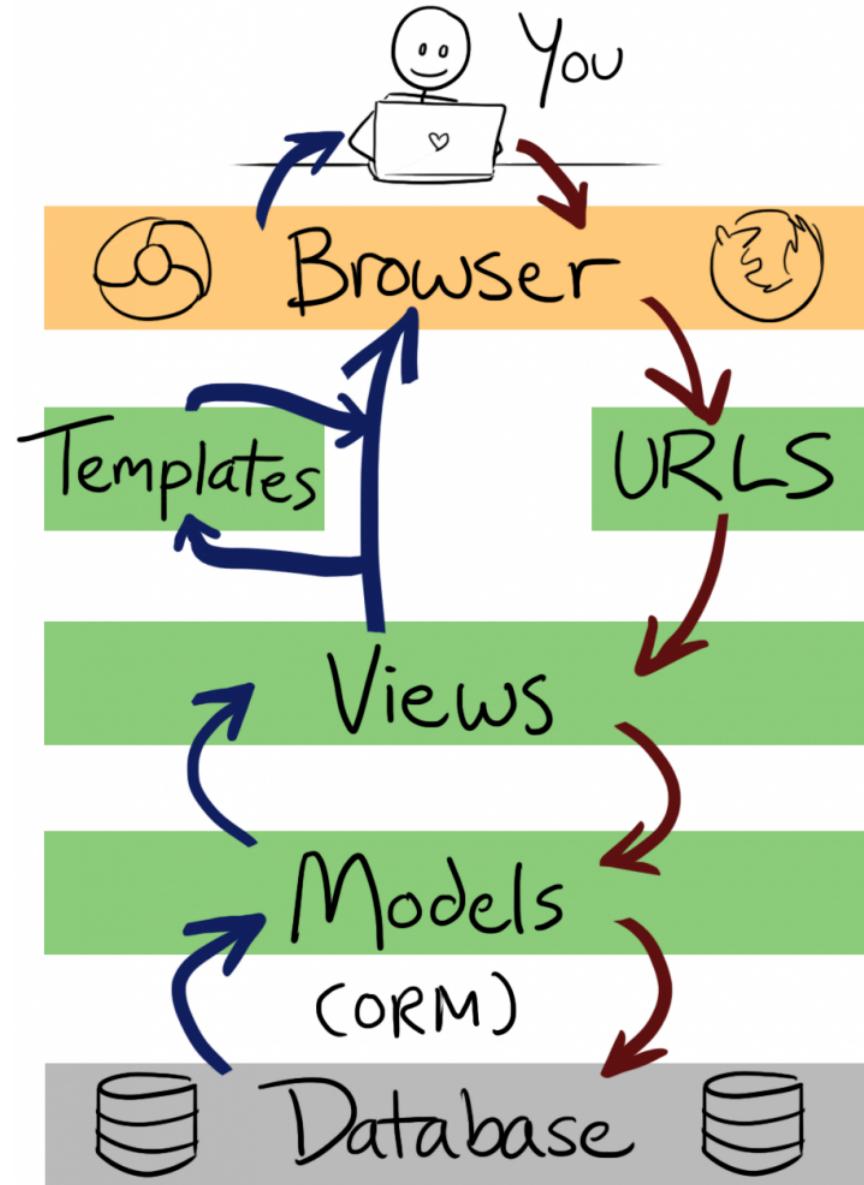
Objetivos: mostrar a camada de modelos do django para criação de modelos de alto nível.

O que são Modelos?

Django funciona no padrão M T V



Django funciona na
arquitetura M T V



O que são Modelos?



ItemAgenda

data: Date

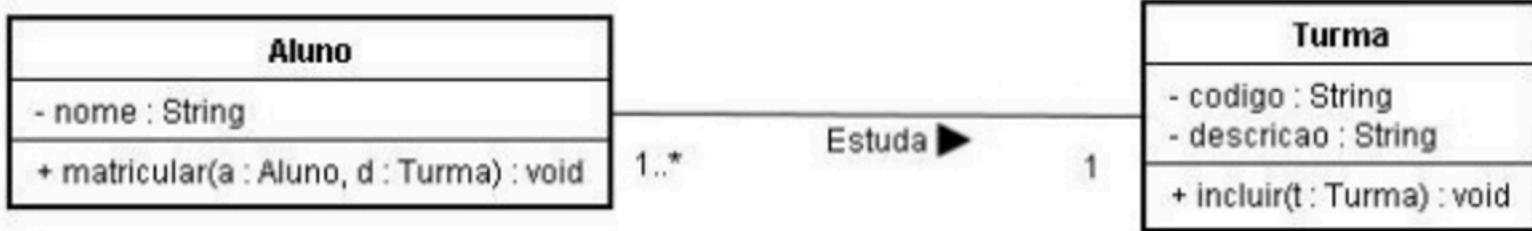
hora: Time

titulo: Varchar(100)

descricao: Text

Modelos são essencialmente o layout do banco de dados, com metadados adicionais.

O que são Modelos?



São usados para representar objetos do mundo real
e seus relacionamentos.

Modelos viram classes



```
from django.db import models

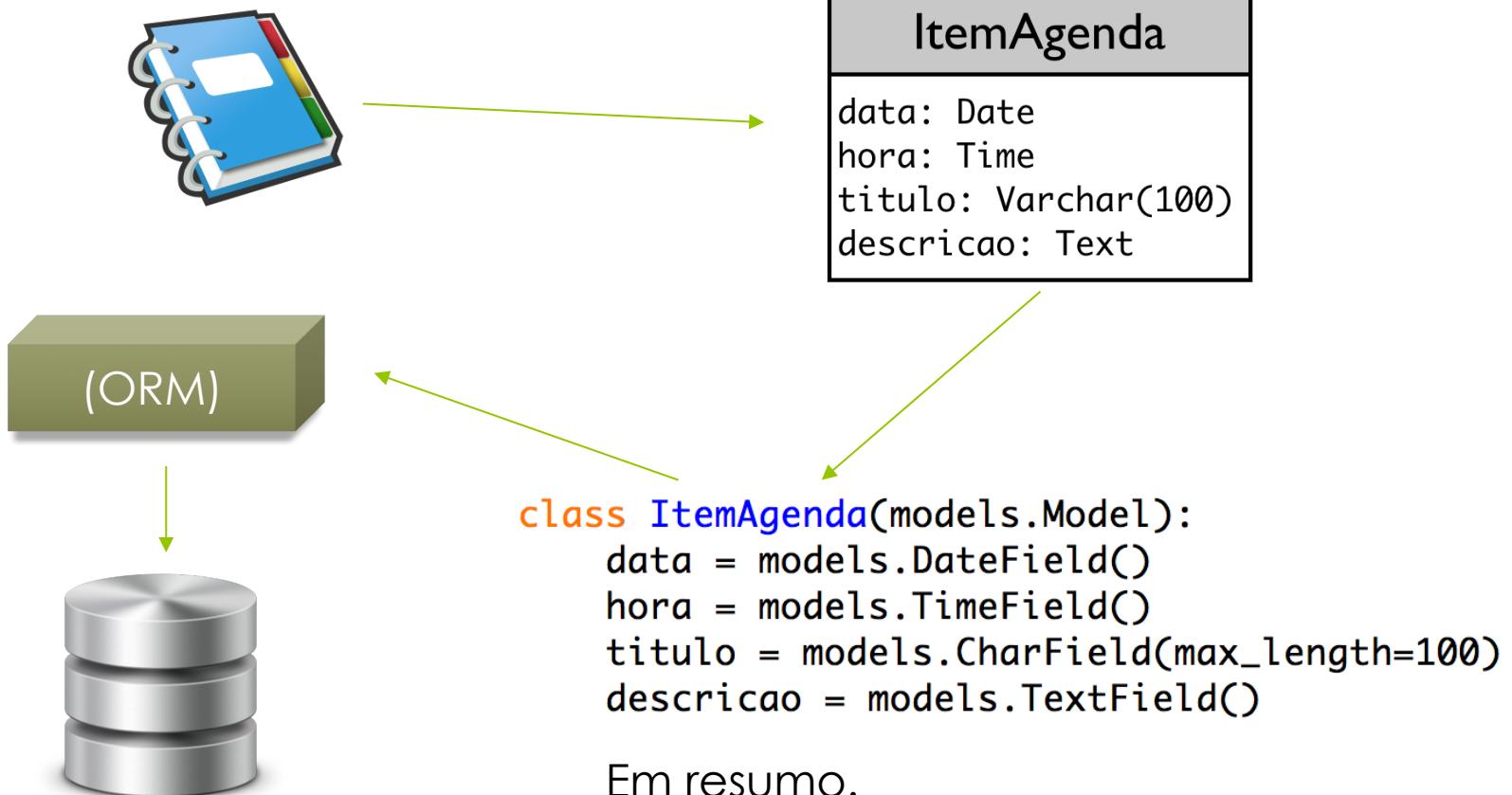
class ItemAgenda(models.Model):
    data = models.DateField()
    hora = models.TimeField()
    titulo = models.CharField(max_length=100)
    descricao = models.TextField()
```

ItemAgenda

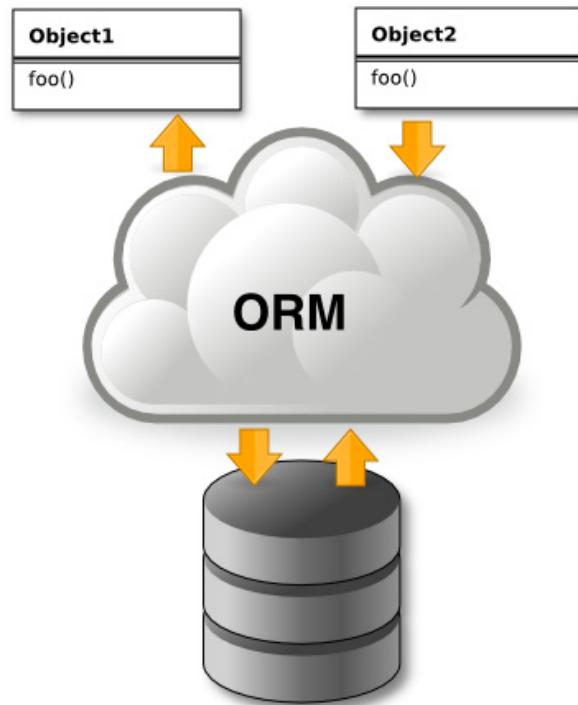
data: Date
hora: Time
titulo: Varchar(100)
descricao: Text

São transformados em código.

São persistidos



Object relational mapping (ORM)

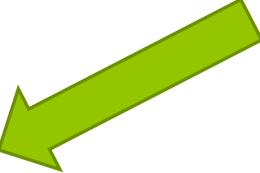


A camada de Mapeamento Objeto-Relacional (ORM) se encarrega de persistir o modelo em um banco de dados relacional.

Modelos

Modelos são representados por simples classes Python. O local padrão para criar modelos é no arquivo **core/models.py**

```
mysite/core
├── migrations/
├── static/
├── templates/
└── __init__.py
    └── models.py
    └── tests.py
    └── views.py
```



Modelos

Cada modelo é representado por uma classe derivada da classe `django.db.models.Model` e possui atributos, que representam campos do banco de dados relacional.

```
from django.db import models

class ItemAgenda(models.Model):
    data = models.DateField()
    hora = models.TimeField()
    titulo = models.CharField(max_length=100)
    descricao = models.TextField()
```

Django vem com os tipos apropriados para criação de campos.

Tipos de Campos

| | | | | |
|---------------|---------------|----------------|----------------------------|-----------|
| BooleanField | EmailField | IntegerField | SmallIntegerField | URLField |
| CharField | FileField | IPAddressField | PositiveSmallIntegerField | XMLField |
| DateField | FilePathField | TextField | CommaSeparatedIntegerField | TimeField |
| DateTimeField | FloatField | SlugField | NullBooleanField | UUIDField |
| AutoField | DecimalField | ImageField | PositiveIntegerField | ... |

Opções dos campos

| | | | |
|-----------|---------------|-----------------|------------------|
| null | db_index | help_text | unique_for_month |
| blank | db_tablespace | primary_key | unique_for_year |
| choices | default | unique | verbose_name |
| db_column | editable | unique_for_date | ... |

Relacionamentos

```
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Aqui, criamos um relacionamento usando [ForeignKey](#). O Django suporta todos os relacionamentos comuns:

- ForeignKey (muitos-para-um);
- ManyToManyField (muitos-para-muitos); e
- OneToOneField (um-para-um).

makemigrations e migrate

Migrations guardam as alterações feitas nos modelos. São arquivos Python editáveis para o caso de ajustes manuais. Ficam em:

```
mysite/core
└── migrations/
    └── 0001_initial.py
```

O comando **makemigrations** diz que você fez algumas mudanças em seus modelos e que você gostaria que essas alterações sejam colocadas para migração ao banco de dados.

O comando **migrate** efetiva as mudanças em seus modelos que estão colocadas para migração para o banco de dados.

Banco de Dados

A persistência dos dados é feita de acordo com o que é especificado na sessão DATABASES do arquivo de configuração settings.py

```
DATAASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': 'mydatabase',  
    },  
}
```

DATABASES é um dicionário de dicionários. No mínimo a configuração **default** deve ser especificada com o ENGINE e NAME

Banco de Dados

O padrão atual é o Django criar o projeto configurado para o banco de dados SQLITE3 e o arquivo ficar gravado na raiz do nosso projeto com o nome db.sqlite3

```
 DATABASES = {  
     'default': {  
         'ENGINE': 'django.db.backends.sqlite3',  
         'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
     }  
 }
```

Banco de Dados

Outras ENGINEs estão disponíveis:

- 'django.db.backends.postgresql'
- 'django.db.backends.mysql'
- 'django.db.backends.sqlite3'
- 'django.db.backends.oracle'

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'mydatabase',  
        'USER': 'mydatabaseuser',  
        'PASSWORD': 'mypassword',  
        'HOST': '127.0.0.1',  
        'PORT': '5432',  
    }  
}
```

Nomes significativos

Vejamos essa classe Contato

```
class Contato(models.Model):
    nome = models.CharField(max_length=200)
    telefone = models.CharField(max_length=15)
```

Ao recuperar um objeto do banco de dados você terá algo como <Contato: Contato object>, uma representação totalmente inútil desse. Corrigimos isso adicionando um método `__str__()` :

```
class Contato(models.Model):
    nome = models.CharField(max_length=200)
    telefone = models.CharField(max_length=15)

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

```
import datetime

from django.db import models
from django.utils import timezone

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def __str__(self):
        return self.question_text

    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)

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

Vamos considerar essas duas classes: Question e Choice.

```
>>> from core.models import Question, Choice
>>> Question.objects.all()
<QuerySet []>

# Criar uma nova Question salva
>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())
>>> q.save()

# Verifica o ID da Question.
>>> q.id
1

# Acessa o campo pelo atributo Python.
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Altera e salva o valor do atributo
>>> q.question_text = "What's up?"
>>> q.save()
```

```
# Obtém todos os objetos gravados
>>> Question.objects.all()
<QuerySet [<Question: Question object>] >      # Sem definir __str__()
<QuerySet [<Question: What's up?>] >          # Com __str__() definida

# Django fornece uma poderosa API de pesquisa baseada em argumentos.
>>> Question.objects.filter(id=1)
<QuerySet [<Question: What's up?>] >
>>> Question.objects.filter(question_text__startswith='What')
<QuerySet [<Question: What's up?>] >

# Obtem Questions que foram publicadas esse ano.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# Tentar recuperar um ID inexistente gera uma exceção
>>> Question.objects.get(id=2)
Traceback (most recent call last):
...
DoesNotExist: Question matching query does not exist.
```

```
# O comando seguinte é identico a Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Verifique se o método personalizado was_published_recently()
# está funcionando.
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
True

# Exibir as Choices da Question relacionada -- nada até agora.
>>> q.choice_set.all()
<QuerySet []>

# Cria três Choices para a Questions relacionada.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Objetos Choice possuem acesso via API aos seus objetos Question.
>>> c.question
<Question: What's up?>
```

```
# E vice versa: Objetos Question acessam Choice.  
>>> q.choice_set.all()  
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking  
again>]>  
>>> q.choice_set.count()  
3
```

```
# A API segue automaticamente relacionamentos sempre que for preciso.  
# Utilize sublinhados duplos para separar os relacionamentos.  
# Isto funciona em vários níveis de profundidade; não há limite.  
# Encontrar todas as Choice para qualquer Question cuja pub_date é neste  
# ano (Reutilizando a variável 'CURRENT_YEAR' que criamos acima).  
>>> Choice.objects.filter(question__pub_date__year=current_year)  
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking  
again>]>
```

```
# Vamos remover uma Choice. Use delete() para isso.  
>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')  
>>> c.delete()
```