Tecnologias Web

Aula 09 – Banco de Dados e ORM

Yuri Dirickson

yuri.dirickson@faculdadeimpacta.com.br

http://github.com/ImpactaTecWeb

Sumário

- Como conectar o Django em SGBD.
- Ver quais bancos são possíveis e recomendados serem utilizados com o Django.
- Entender a utilidade de uma ferramenta ORM.
- Criar tabelas a partir de modelos e criar modelos a partir de tabelas.

Django - Banco de Dados

- Todo *framework* possui suas maneiras de conectar a alguns bancos de dados.

- Conectamos ao BD - Banco de Dados - para salvar (ou persistir) os nossos dados das aplicações.

- Para conectar aos BD's, as linguagens de programação utilizam *drivers* de conexão.

- Ao conectar no BD, temos API's genéricas de acesso, leitura e escrita no BD.

Django - Banco de Dados

- No arquivo **settings.py** alteramos as configurações de conexão em banco de dados. Procuramos o trecho com a seguinte lista:

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

- O Django vem por padrão com o SQLite (v3). É um micro BD com persistência e consistência de dados, perfeito para testes e aplicações aninhadas (*embedded*).

Django - Banco de Dados

- O Django oferece suporte a praticamente todos os BD's relacionais comuns:
  - PostgreSQL
  - SQLite v3
  - MySQL
  - Oracle
  - SQLServer
- Para praticamente todos, deve ser instalado um driver adicional via pip (e estar presente no **requirements.txt**)
- Seguem exemplos de como conectar nos bancos mais comuns:

Django - Banco de Dados

- Conectando no PostgreSQL: **(pip install psycopg2)**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mydb',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost,
        'PORT': ''
    }
}
```

- Conectando no MySQL: **(pip install mysqlclient)**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'mydb',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': ''
    }
}
```

Django - Banco de Dados

- Conectando no SQLServer (2005, 2008/2008R2, 2012, 2014, 2016 and Azure SQL Database): **(pip install** pyodbc-azure**)**

```
DATABASES = {
    'default': {
        'ENGINE': 'sql_server.pyodbc',
        'NAME': 'mydb',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': 'servidor'
    }
}
```

Django - Banco de Dados

- Vamos continuar usando o SQLite nos exemplos.
- Para explorar o conteúdo das tabelas do SQLite, podemos usar o SQLite Studio, uma ferramenta open source e completamente portável (não precisa instalar nada).
- As modificações feitas via Django são independentes do SGBD que utilizarmos.
- Vamos começar a mexer no BD com o Django.

Django - ORM

- No mundo da orientação a objetos, é comum fazermos uma ponte, ou tradução, do modelo relacional para o modelo de objetos.

- Para auxiliar nessa transição, os *frameworks* costumam dispor de ferramentas auxiliares. Essas ferramentas são chamadas de ORM - Object Relational Mapping, ou Mapeamento Objeto-Relacional.

- Essas ferramentas utilizam as melhores práticas tanto de modelagem relacional quanto orientação a objetos para criar um mapa entre os dois modelos.

Django - ORM

- Em geral temos dois mapeamentos possíveis:
  - **Mapeamento Tabelas - Classes:**
    - A ferramenta escaneia todas as tabelas criadas e cria classes para todas as tabelas primárias (não relacionamento), os relacionamentos em chave estrangeira são representados por composição de objetos.
  - **Mapeamento Classes - Tabelas:**
    - Mapeia as suas classes criadas e constrói tabelas para cada uma. Composição de objetos são tratadas como relacionamentos em chave estrangeira.

Django - ORM

- Características de uma ferramenta ORM:
  - Mapeamento de duas mãos (classe - tabela).
  - Gerenciamento automático de conexões com o BD.
  - Identificação em classes que serão gerenciadas pela ORM (annotations ou metadados).
  - Geração automática de SQL.
  - Utilização de chave artificial numérica é preferível (em algumas é obrigatória).

Django - ORM

- Ferramentas ORM conhecidas:
  - **Hibernate (JAVA - JPA)**
  - **EclipseLink (JAVA - JPA)**
  - **Django ORM (Python - Django)**
  - **CodeIgniter (PHP)**
  - **SQLAlchemy (Python)**
  - **ActiveRecord (Ruby)**
  - **NHibernate (.NET)**
  - **Muitos outros: https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software**

Django - ORM

- Como funciona no Django?

- A ORM do Django já vem instalada nativamente. Utiliza o conceito de ***migrations*** (migrações).

- Migrações são arquivos de diferenças entre dois estados do seu modelo (tabelas ou classes), ou seja, sempre que você alterar as suas classes é possível gerar um arquivo de diferenças no SQL (e vice-versa).

- Como conectar no BD para testar?

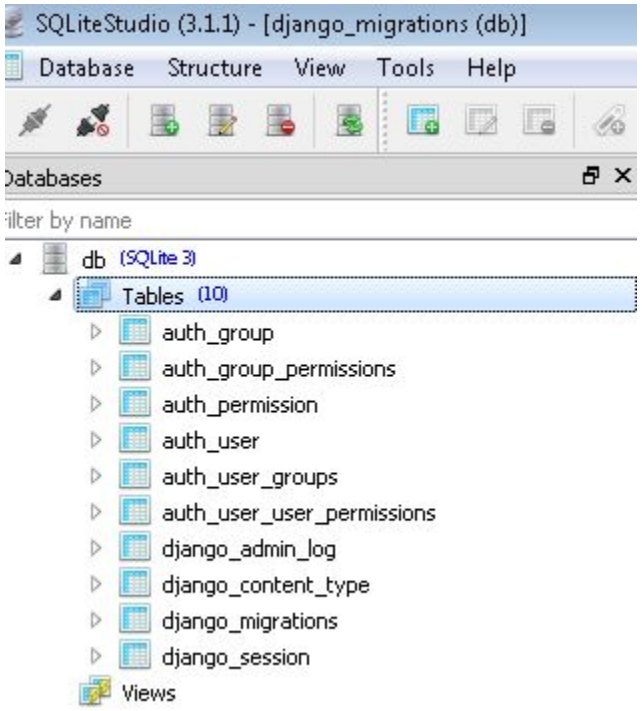
Django - ORM - Migrações

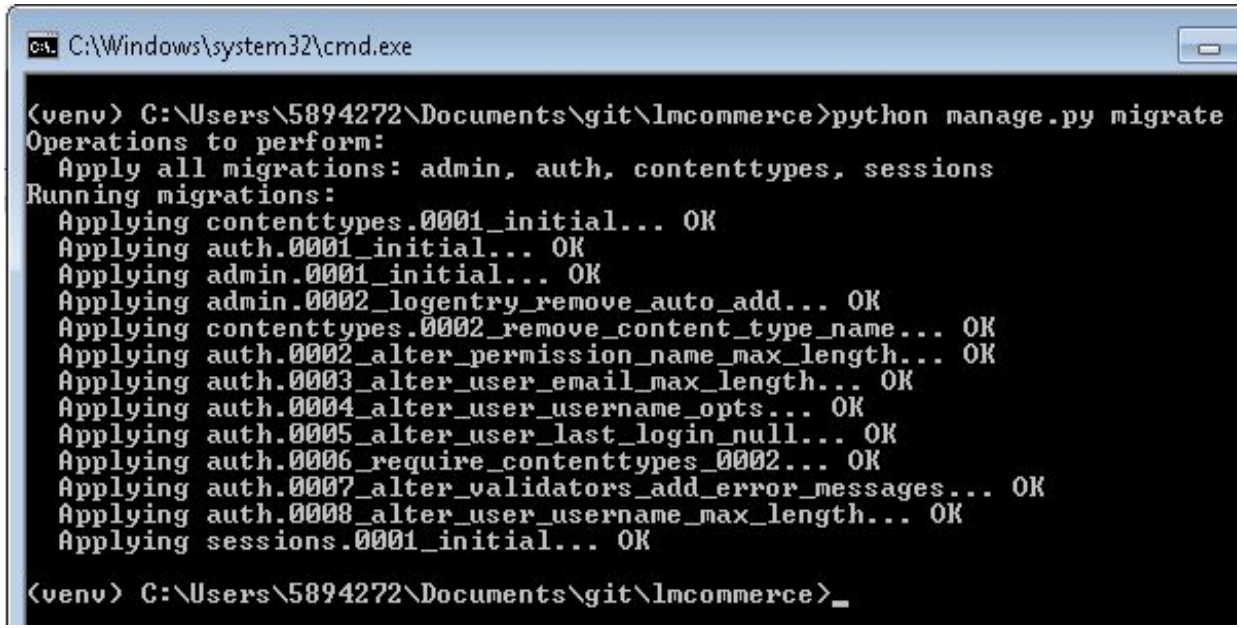
- Por enquanto não temos nenhuma classe nossa para migrar para o Banco de Dados. Mas e todas aquelas aplicações já instaladas no Django?

- Elas possuem classes e modelos próprios, portanto podemos migrá-las para o BD.

- Execute o seguinte comando **python manage.py migrate** e veja o resultado

Django - ORM - Migrações





Vemos que o Django já aplicou uma série de migrações vindas de várias aplicações padrões (auth, admin, sessions).

Django - ORM - Migrações

- E nossas próprias classes, como funciona?
- Para estudar melhor a modelagem de objetos, vamos começar uma nova aplicação no nosso projeto: **o currículo da faculdade**.
- Para isso vamos fazer o comando **python manage.py startapp curriculo** (sem acentos mesmo).
- Não esqueça de registrar a aplicação no **settings.py**
- Quando a aplicação estiver criada, vamos entrar no arquivo de modelos da aplicação.

Django - ORM - Modelos

- O arquivo por enquanto está com apenas um **import**

```
from django.db import models

# Create your models here.
```

- Esse **import** é importante pois todo modelo que é gerenciado pelo Django deve herdar da classe ***models.Model***

- Dentro desse arquivo vamos definir os nosso modelos do Django.

- Como estamos fazendo o nosso currículo, vamos criar dois modelos: Curso e Disciplina.

Django - ORM - FieldTypes

- Como o Python é dinamicamente tipado, não teríamos como definir os tipos dos atributos para as tabelas.

- Para isso, o Django possui diversos *Model Field Types* (Tipos de campos de modelo).

- Esses *Field Types* fazem com que o modelo consiga traduzir um atributo para uma coluna do SQL.

- Existem quase 30 tipos definidos.

Django - ORM - FieldTypes

- Alguns dos tipos comuns de SQL:
  - **CharField:** Referente ao VARCHAR do SQL
  - **DateField e DateTimeField:** Referente ao DATE e DATETIME do SQL
  - **DecimalField:** Referente ao DECIMAL do SQL
  - **IntegerField e BigIntegerField:** Referente ao INT e BIGINT do SQL
  - **TextField:** Referente ao TEXT do SQL
  - Todos os campos podem ser vistos em: https://docs.djangoproject.com/en/2.0/ref/models/fields/

Django - ORM - FieldTypes

- Alguns tipos úteis
  - **SlugField:** Usa o VARCHAR do SQL. Usado para identificadores de texto (não usa caracteres não ASCII - acentos).
  - **AutoField e BigAutoField:** Usa os campos INT e BIGINT do SQL. São campos de auto incremento, utilizados para identificação de elementos (PK).
  - **EmailField:** Usa o VARCHAR do SQL. Valida e-mails.
  - **URLField:** Usa o VARCHAR do SQL. Valida URL's.
  - **ForeignKey:** Faz referência a outro modelo (vamos ver adiante).

Django - ORM - Modelos

- Vamos construir nosso primeiro modelo, Curso. Ele deve ter os atributos nome, sigla e etiqueta (um identificador especial). Fica assim:

```
from django.db import models

class Curso(models.Model):

    nome = models.CharField("Nome", max_length=50)
    sigla = models.CharField("Sigla", max_length=5)
    etiqueta = models.SlugField("Etiqueta", max_length=50)

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

- No nosso primeiro modelo criamos uma classe com nome e etiqueta e a função de representação (**__str__**).
- Todo *FieldType* possui como primeiro parâmetro o ***verbose_name***, que seria um texto descritivo curto sobre o atributo.
- Depois vem uma sequência de parâmetros nomeados para configurar o campo (ex: **max_length**)

Django - ORM - Modelos

- Ao criar um primeiro modelo, vamos criar a sua migração para o BD.
- Rode o comando **python manage.py makemigrations**
- Dentro das pastas **migrations** das aplicações que contiverem modelos novos ou alterados, vão aparecer alguns arquivos gerados.

Django - ORM - Modelos

- Esses arquivos vão conter as diferenças encontradas nos modelos e no BD, toda vez que alteramos o modelo de alguma maneira e quisermos replicar isso no BD, devemos rodar o comando **python manage.py makemigrations** para ver se tudo está certo.

- Caso a classe contenha algum erro de interpretação, durante o comando vamos saber quais erros existem.

- Se tudo estiver certo, podemos rodar o **python manage.py migrate** e olhas novamente o SQLite Studio.

Django - ORM - Modelos

- Acontece a migração agora da aplicação **curriculo**.
- É criada a tabela **curriculo_curso**, com os campos especificados no modelo.
- Automaticamente é criado um ID numérico auto incrementado (bônus do Django).

Django - ORM - Modelos

- Com a categoria criada, vamos criar o modelo Produto no catálogo. Esse modelo deve ter os atributos: nome, identificador, preço, descrição e categoria. Temos então o segundo modelo:

```
from django.db import models

class Curso(models.Model):
    ...

class Disciplina(models.Model):

    nome = models.CharField("Nome", max_length=50)
    etiqueta = models.SlugField("Etiqueta", max_length=50)
    carga_horaria = models.IntegerField("Carga Horaria")

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

Django - ORM - Shell

- E como podemos manipular esses objetos no BD?
- Para testar se o acesso está correto e eficiente, temos um comando presente: **python manage.py shell**.
- Isso abre um *shell* do Python com as informações e configurações do Django já carregadas.
- A primeira coisa a fazer é importar os nossos modelos:

```
>>> from curriculo.models import Curso, Disciplina
```

Django - ORM - Shell

- Agora temos acesso aos nossos modelos. Podemos digitar o nome deles que o Python já reconhece os seus tipos:

```
>>> Curso
<class 'curriculo.models.Curso'>
>>> Disciplina
<class 'curriculo.models.Disciplina'>
```

- Para cada modelo que estende o objeto *models.Model,* o Python adiciona uma série de ferramentas embaixo do objeto ***objects***. Ao mandar imprimir o ***objects*** do modelo, temos a resposta:

```
>>> Curso.objects
<django.db.models.manager.Manager  object at 0x00000000043834A8>
```

- Esse objeto **Manager** vai ser nossa interface com o BD.

Django - ORM - Shell

- Vamos analisar os métodos mais gerais do **Manager**:
  - all(): Lista todas as entradas desse tipo.
  - create(...): Cria uma nova entrada desse tipo.
  - get(...): Obtém uma única entrada baseada em um critério (deve voltar um).
  - filter(...): Lista todas as entrada que obedecem algum critério

Django - ORM - Shell

- Criando um novo elemento:

```
>>> Curso.objects.create(nome="Sistema da da
Informação",etiqueta="sistemas-da-informacao")
```

- Podemos listar as entradas agora:

```
>>> cursos = Curso.objects.all()
>>> cursos
<QuerySet [<Curso: Sistemas da Informação>]>
>>> cursos[0]
<Curso: Sistemas da Informação>
```

- No caso do **all()**, é retornado um objeto do tipo *QuerySet*, que contém uma lista dos objetos retornados do BD.
- Para usar o get, podemos usar:

```
>>> Curso.objects.get(nome="Sistemas da Informação")
<Curso: Sistemas da Informação>
```

Django - ORM - Shell

- Vamos adicionar mais uma categoria e filtrar elas:

```
>>> Curso.objects.create(nome="Banco  de Dados",etiqueta="banco-de-dados")
<Curso:Banco de Dados>
>>> Curso.objects.filter(nome="Banco  de Dados")
<QuerySet [<Curso: Banco de Dados>]>
>>> Curso.objects.filter(nome__icontains="dad")
<QuerySet [<Curso: Banco de Dados>]>
```

- Fizemos duas filtragens:
  - Por nome exato, usando **name=”valor”**. Ele atua como se fosse uma cláusula WHERE name=”valor” no SQL.
  - Por parte de nome. Toda vez que em filtragens usarmos o marcador __ (dois sublinhados) após o nome de uma propriedade ele usa o ***lookup***, que é uma forma de buscar por parte de nomes, nesse caso todo nome que contenha a String “th”. Isso fica similar ao operador **nome LIKE “%th%”.**

Django - ORM - Relações

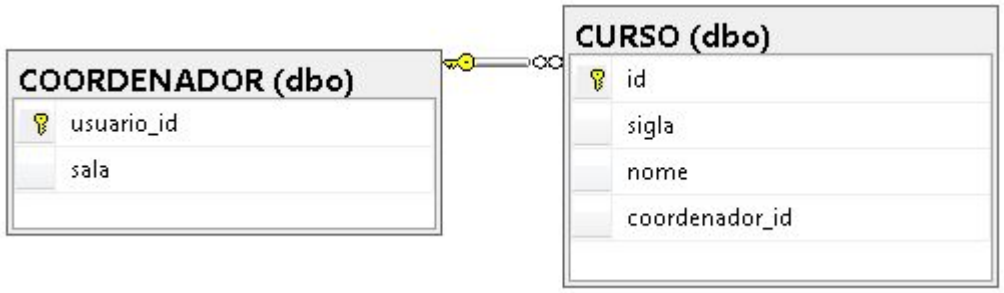
- Mas e como mapeamos relações entre objetos?
- Quando modelamos bancos de dados, fazemos todo tipo de migração de chaves estrangeiras para mapear os relacionamentos das nossas entidades. Como fazer isso no mundo objeto?
- Objetos se relacionam por composição. Por exemplo, um curso tem várias disciplinas, então teria uma propriedade que seria uma lista de disciplinas.
- Vamos mostrar os tipos de relacionamentos no ORM.
- Em via de regra temos apenas três tipos: **muitos para um**, **muitos para muitos** e **um para um**.

Django - ORM - Relações

- **Um para Um:**

Esse tipo de relação conecta dois objetos de maneira que o objeto pai tenha apenas um do objeto filho.

Exemplo: Vamos imaginar que no nosso sistema, o **Coordenador** deve estar vinculado a um único **Curso**. Para isso, como **Curso** só irá existir se estiver vinculado a um **Coordenador**, teremos uma chave estrangeira na sua tabela:



Django - ORM - Relações

- **Um para Um:**

Para construir essa relação, vamos alterar os seguintes modelos:

```
...
class Curso(models.Model):
   sigla = models.CharField(unique=True, max_length=5)
   nome = models.CharField(unique=True, max_length=50)
   coordenador = models.OneToOneField(
      "contas.Coordenador",
      null=True # Se o modelo já existia antes
    )

class Coordenador(Usuario):
    sala = models.CharField("Sala", max_length=3)

...
```

Para garantir o tipo **Um para Um**, usamos o OneToOneField, passando o nome do modelo a ser ligado

Django - ORM - Relações

- **Um para Um:**

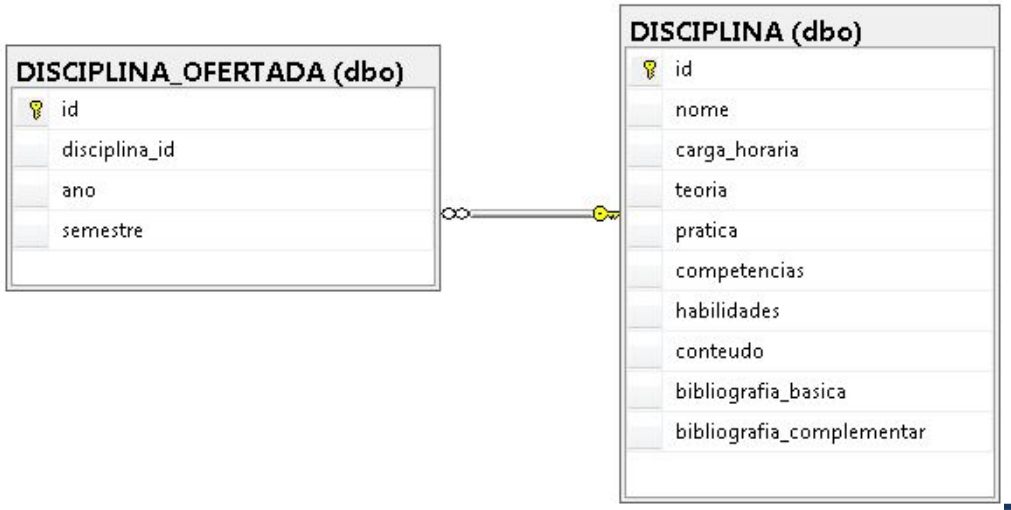
O uso do relacionamento **Um para Um** garante que você não possa fazer com que o coordenador tenha mais do que um curso.

Ao obter o curso, o objeto resultante no atributo **coordenador** será do tipo coordenador mesmo (e não uma lista).

Django - ORM - Relações

- **Muitos para Um:**

Esse tipo de relação conecta dois objetos de maneira que o objeto pai tenha vários objetos filhos. Do ponto de vista do banco de dados, a modelagem é similar (chave estrangeira na entidade fraca).
Exemplo: Uma **Disciplina** pode ser oferecida em vários anos e semestre **(DisciplinaOfertada**), mas apenas uma vez por semestre e ano.



Django - ORM - Relações

- **Muitos para Um:**

Para construir essa relação, vamos alterar os seguintes modelos:

```
...
class DisciplinaOfertada(models.Model):
   ano = models.SmallIntegerField("Ano")
   semestre = models.CharField("Semestre", max_length=1)
   disciplina = models.ForeignKey(
      Disciplina,
      null=True # Se o modelo já existia antes
    )

class Disciplina(Usuario):
    nome = models.CharField(“Nome")

...
```

Para garantir o tipo **Muitos para Um**, usamos o campo ForeignKey, passando o nome do modelo a ser ligado.

Django - ORM - Relações

- **Muitos para Um:**

O uso do relacionamento **Muitos para Um** garante que uma disciplina possa ter várias ofertas

Ao obter a oferta da disciplina, o objeto resultante no atributo **disciplina** será do tipo coordenador mesmo (e não uma lista).

Mesmo não sendo recomendado, podemos usar o bidirecional em disciplina com a seguinte propriedade:

```
 oferta = models.ForeignKey(to=DisciplinaOfertada,
related_name="disciplina", null=True, blank=True)
```

Django - ORM - Relações

- **Muitos para Muitos:**

Esse tipo de relação conecta dois objetos de maneira que o objeto pai tenha vários objetos filhos e os filhos tenham vários pais.

Em geral é representado por uma tabela de associação.

Exemplo: Um aluno pode se matricular em várias **Turmas**, assim como uma **Turma** possui vários alunos. Para isso se usa uma tabela associativa (**Matrícula**)

Django - ORM - Relações

- **Muitos para Muitos:**

Para construir essa relação, vamos alterar os seguintes modelos:

```
...
class Turma(models.Model):
    disciplina_ofertada = models.ForeignKey(DisciplinaOfertada)
    identificador = models.CharField(max_length=1)
    turno = models.CharField(max_length=15, blank=True, null=True)
    professor = models.ForeignKey("contas.Professor", blank=True, null=True)
    alunos = models.ManyToManyField(
        "contas.Aluno",
        db_table="MATRICULA"
    )

class Aluno(Usuario):
...
```

Para garantir o tipo **Muitos para Muitos**, usamos o ManyToMany, passando o nome do modelo a ser ligado e a tabela que cuida dessa ligação

Django - ORM - Relações

- **Muitos para Muitos:**

O uso do relacionamento **Muitos para Muitos** garante que um aluno possa ter várias turmas e uma turma tenha vários alunos.

Ao obter a turma, o objeto resultante no atributo **alunos** será uma lista de objetos do tipo **aluno**.

Lembrando que primeiro ambas as entidades devem existir (inseridas no banco) antes de serem associadas.