Programação de Sistemas para Internet

Prof. Diego Cirilo

Aula 08: Models

Dados do sistema

- Como armazenar os dados do sistema?
- Conteúdo, dados de usuário, etc.;
- SGBD - Sistema de Gerenciamento de Banco de Dados;
- SQL - *Structured Query Language*.

ORM

- Gerenciar comandos SQL pode ser complicado;
- ORM - *Object Relational Mapping*;
- Podemos tratar os dados do BD como objetos.

Models Django

- No Django o ORM é feito nos *Models* (`models.py`);
- Nele definimos os modelos de dados, seus campos e comportamento;
- Cada modelo é uma classe Python que herda de `django.db.models.Model`;
- Cada atributo de um model representa um campo no BD;
- O Django se responsabiliza por gerenciar as tabelas, *queries*, etc.;
- O Django também cria campos automaticamente, como ID.

Exemplo

```python
from django.db import models

class Pessoa(models.Model):
    nome = models.CharField(max_length=30)
    sobrenome = models.CharField(max_length=30)
```

```sql
CREATE TABLE myapp_pessoa (
    "id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    "nome" varchar(30) NOT NULL,
    "sobrenome" varchar(30) NOT NULL
);
```

Alguns tipos de dados

- CharField: Textos curtos (nomes, títulos);
- TextField: Textos longos (descrições, artigos);
- IntegerField: Números inteiros (idades, quantidades);
- FloatField: Números decimais (preços, médias);
- BooleanField: Valores booleanos (Verdadeiro/Falso);
- DateField: Datas (aniversários, datas de criação);
- DateTimeField: Datas e horas (eventos, logs);
- Referência.

Alguns tipos de dados

- EmailField: Endereços de e-mail (validação automática);
- FileField: Arquivos;
- ImageField: Imagens;
- ForeignKey: Relacionamentos um-para-muitos;
- ManyToManyField: Relacionamentos muitos-para-muitos;
- OneToOneField: Relacionamentos um-para-um;
- Referência.

Alguns atributos dos campos

- `max_length` - tamanho máximo para texto;
- `null` - se vazio usa o `NULL` do SGBD;
- `blank` - o campo pode ser vazio se `True`, por padrão é `False`;
- `default` - valor padrão;
- `unique` - se `True` o valor deve ser único na tabela;
- `choices` - lista de valores possíveis.

FileField e *ImageField*

- Sobem arquivos para uma pasta no servidor;
- Segurança!
- No BD fica salvo o endereço o arquivo;
- Para *ImageField* é necessário instalar o pacote `pillow`;
- Os arquivos são salvos na pasta configurada em `MEDIA_ROOT`;
- O link para acesso aos arquivos fica configurado em `MEDIA_URL`;

FileField e *ImageField*

- Esses diretórios não são configurados por padrão;
- Devemos adicionar no `settings.py`:

```
MEDIA_URL = "media/"
MEDIA_ROOT = BASE_DIR / "media"
```

FileField e *ImageField*

- No `urls.py` (apenas durante o desenvolvimento!):

```
from django.conf.urls.static import static

urlpatterns = [
    ..., # importante acabar com vírgula!
]
# concatena a lista acima com o resultado de static()
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

FileField e *ImageField*

- Atributo `upload_to`:
  - diretório dentro de `MEDIA_ROOT`;
  - permite organizar melhor os arquivos;

```
class Perfil(models.Model):
    documento = FileField(upload_to="documentos/")
    avatar = ImageField(upload_to="avatars/")
```

Relacionamentos

- Relacionam modelos;
- Recebem como argumentos o nome da classe relacionada;
- ForeignKey: chave estrangeira (um-para-vários);
- ManyToManyField: muitos-para-muitos;
- OneToOneField: um-para-um.

Relacionamentos

- `on_delete` - define o que ocorre quando o objeto é removido;
- `on_delete=models.CASCADE` - deleta os objetos relacionados junto;
- `on_delete=models.SET_NULL` - escreve `NULL`;
- `on_delete=models.PROTECT` - impede a remoção enquanto houver dados relacionados;
- Referência.

Exemplos

- Um pra um:

```
class Pessoa(models.Model):
    nome = models.CharField(max_length=100)
    cpf = models.CharField(max_length=11, unique=True)

class DadosPessoais(models.Model):
    pessoa = models.OneToOneField(Pessoa, on_delete=models.CASCADE)
    data_nascimento = models.DateField()
```

Exemplos

- Um pra vários:

```
class Pessoa(models.Model):
    nome = models.CharField(max_length=100)
    cpf = models.CharField(max_length=11, unique=True)

class Veiculo(models.Model):
    pessoa = models.ForeignKey(Pessoa, on_delete=models.PROTECT)
    placa = models.CharField(max_length=7)
    modelo = models.CharField(max_length=50)
```

Exemplos

- Vários pra vários:

```
class Pessoa(models.Model):
    nome = models.CharField(max_length=100)
    cpf = models.CharField(max_length=11, unique=True)

class Empresa(models.Model):
    socios = models.ManyToManyField(Pessoa)
    nome = models.CharField(max_length=50)
    cnpj = models.CharField(max_length=14, unique=True)
```

Exemplos

- Vários pra vários (com tabela intermediária):

```
class Pessoa(models.Model):
    nome = models.CharField(max_length=100)
    cpf = models.CharField(max_length=11, unique=True)

class Empresa(models.Model):
    socios = models.ManyToManyField(Pessoa, through="Socio")
    nome = models.CharField(max_length=50)
    cnpj = models.CharField(max_length=14, unique=True)

class Socio(models.Model):
    pessoa = ForeignKey(Pessoa, on_delete=models.CASCADE)
    empresa = ForeignKey(Pessoa, on_delete=models.CASCADE)
    data_entrada = DateField() # campos extras
    data_saida = DateField() # campos extras
```

Choices

- Lista de valores que um campo pode assumir;
- Validação automática e consistência;

```
class Usuario(models.Model):
    ALUNO = "AL"
    PROFESSOR = "PR"
    MONITOR = "MO"
    TIPOS = {
        ALUNO: "Aluno",
        PROFESSOR: "Professor",
        MONITOR: "Monitor",
    }

    nome = models.CharField(max_length=100)
    tipo = models.CharField(max_length=2, choices=TIPOS, default=ALUNO)
```

Choices

- É possível acessar com `Usuario.ALUNO`, etc;
- Ex.

```
from .models import Usuario
...
if usuario.tipo == Usuario.ALUNO:
    print("O usuário é aluno.")
```

Classe Meta

- Subclasse que permite algumas informações extras;
- Ex.
  - `verbose_name`
  - `verbose_name_plural`
  - `ordering`

  -Referência

Métodos

- Como qualquer classe, é possível adicionar métodos aos models;
- Esses métodos podem tratar dados, retornar informações processadas, etc;
- Ex.: um método que retorna o `nome_completo` a partir dos campos `nome` e `sobrenome`;
- Existem alguns métodos padrão que também é possível sobrescrever;
- Ex. `__str__` retorna a string que será impressa quando fazemos um `print` em um objeto dessa classe;
- Para métodos que funcionem como atributos (sem precisar chamar com `()`) usamos o *decorator* `@property`.

Métodos

- Ex.:

```
class Pessoa(models.Model):
    nome = models.CharField(max_length=100)
    sobrenome = models.CharField(max_length=100)

    class Meta:
        ordering = ["sobrenome"] # retorna os resultados ordenados pelo sobrenome

    def __str__(self):
        return self.nome

    @property
    def nome_completo(self):
        return f"{self.nome} {self.sobrenome}"
```

Acessando dados dos Models

- A *view* é responsável por acessar os dados;
- É possível fazer *queries* através do objeto do Model:
  - `Model.objects.all()` - retorna *tudo*;
  - `Model.objects.filter()` - permite *filtrar* os dados;
  - `Model.objects.get(pk=4)` - seleciona um objeto específico;
- Retornam `QuerySets`;
- Os resultados podem ser enviados para o template no `context`.

Filter

- Parecido com `WHERE` do SQL;
- Padrão:
  - `campo__condicao=valor`
- Ex. alunos com menos de 18 anos:
  - `Alunos.objects.filter(idade__lte=18)`

Filter Lookups

- `exact` - valor tem que ser exato;
- `iexact` - exato mas *case-insensitive*;
- `contains` - contém o valor;
- `startswith` - começa com;
- `endswith` - termina com;
- Referência.

Shortcuts

- O Django disponibiliza alguns atalhos;
- Já usamos o `render` e o `redirect`
- Para acesso aos models existem os:
  - `get_object_or_404()`
  - `get_list_or_404()`
- Podemos passar um Model e uma query, se não houver resposta, o sistema redireciona para a página de erro 404;
- Referência.

Exemplos

- views.py:

```
from django.shortcuts import render, get_object_or_404, get_list_or_404
from .models import Livro

def detalhar_livro(request, id_do_livro):
    livro = get_object_or_404(Livro, id=id_do_livro)
    context = {
        'livro': livro,
    }
    return render(request, "detalhar_livro.html", context)

def listar_livros(request):
    livros = get_list_or_404(Livro)
    context = {
        'livros': livros,
    }
    return render(request, "listar_livros.html", context)
```

Exemplos

- Também é possível filtrar diretamente ou usar QuerySets:

```
...
livros_com_M = get_list_or_404(Livro, titulo__startswith="M")

livros_com_N = Livros.objects.filter(titulo__startswith="N") #queryset
livros_com_N_do_autor1 = get_object_or_404(livros_com_N, autor=1)
...
```

Configurações do BD

- No arquivo `settings.py` há a seção `DATABASES`;
- Podemos configurar diversos SGBDs;
- Ex. MySQL, PostGres, SQLite, etc.;
- Para desenvolvimento o SQLite é simples e exige menos configuração;
- No sistema final devemos usar um SGBD mais completo (veremos em ICS).

Configurações de BD

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Migrations

- Arquivo com comandos para o SGBD;
- Permite recriar a estrutura (*schema*) do BD em qualquer computador;
- Novas migrações devem ser criadas sempre que alteramos os dados em Models:
  - `python manage.py makemigrations`
- Para **aplicar** as *migrations*, ou seja, criar/alterar as tabelas no SGBD:
  - `python manage.py migrate`
- É importante sempre lembrar de criar/aplicar as migrations.

Django Admin

- O Django foi pensado para facilitar o processo de desenvolvimento;
- Um projeto Django já possui uma interface de administração pronta;
- Para ativar, temos que:
  - Adicionar os models ao arquivo `admin.py`;
  - Criar um *superuser* do sistema;
  - Executar as *migrations*.

Django Admin

```
from django.contrib import admin

from .models import Tarefa

admin.site.register(Tarefa)
```

Django Admin

- Para criar o *superuser*:
  - `python manage.py createsuperuser`
- Para criar as *migrations*:
  - `python manage.py makemigrations`
- Para executar as *migrations*:
  - `python manage.py migrate`

Django Admin

- Para configurar a língua do sistema (incluindo o *admin*):
  - Altere `LANGUAGE_CODE` no `settings.py` para `pt-br`
- Na listagem de tarefas aparece `Tarefas object(1)`, esse é o resultado do `print` em um objeto da classe `Tarefas`
- Para imprimir algo mais interessante, escrevemos o método `__str__` para a classe `Tarefas`

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

Tarefa 05

- Crie um site simples de lista de tarefas;
- Cada tarefa deve ter: nome, status e prazo;
- O cadastro das tarefas deve ser feito pelo Django Admin;
- Diferencie as tarefas que estão atrasadas;
- Dica: use a biblioteca `datetime` do Python na view para passar a data atual no `context`:

```
from datetime import date

...
context['hoje'] = date.today()
```

Tarefa 06

- Crie um blog simples;
- O blog deve ter um *header* com o título e um *footer* com informações do desenvolvedor;
- O conteúdo de cada postagem deve ser apenas uma imagem, um título, o texto e a data de publicação;
- Todas essas informações devem existir no BD;
- Crie um *superuser* e cadastre as postagens pela página de *admin*.

Dúvidas?