Sistema de Biblioteca com Django Models - Explicações Detalhadas

1. Introdução aos Models no Contexto de Biblioteca

Os models são a espinha dorsal de qualquer aplicação Django, funcionando como uma camada de abstração entre o código Python e o banco de dados relacional. Em um sistema de biblioteca, precisamos representar entidades do mundo real como livros, autores, usuários e empréstimos, além dos relacionamentos complexos entre elas.

O Django ORM (Object-Relational Mapping) traduz automaticamente operações Python em queries SQL, permitindo que você trabalhe com dados usando objetos Python familiares em vez de escrever SQL manualmente. Isso traz benefícios significativos:

- Produtividade: Desenvolve mais rápido com código Python
- Segurança: Previne vulnerabilidades como SQL injection
- Portabilidade: Funciona com diferentes bancos de dados (PostgreSQL, MySQL, SQLite)
- Manutenibilidade: Código mais limpo e organizado

No contexto da biblioteca, vamos modelar três entidades principais:

- Autor: Representa os escritores dos livros
- Livro: Representa os livros disponíveis no acervo
- Emprestimo: Controla os empréstimos dos livros aos usuários

---

2. Configuração dos Models da Biblioteca - Explicação Detalhada

Model Autor - A Base do Sistema

O model Autor armazena informações sobre os escritores e serve como referência para os livros. Vamos analisar cada parte:

```python
from django.db import models
from django.contrib.auth.models import User
from django.core.validators import MinLengthValidator

class Autor(models.Model):
    # Choices para gênero - fornece opções pré-definidas
    GENERO_CHOICES = [
        ('M', 'Masculino'),
        ('F', 'Feminino'),
        ('O', 'Outro'),
    ]
```

Explicação dos campos principais:

```python
nome = models.CharField(
    max_length=100,                    # Tamanho máximo do campo
    verbose_name="Nome completo",      # Nome amigável para exibição
    help_text="Digite o nome completo do autor",  # Texto de ajuda
    validators=[MinLengthValidator(3)] # Validação mínima de 3 caracteres
)
```

Campos de data - diferenças importantes:

```python
data_nascimento = models.DateField(
    verbose_name="Data de nascimento",
    null=True,        # Permite NULL no banco de dados
    blank=True        # Permite campo vazio em formulários
)

data_falecimento = models.DateField(
    null=True,
    blank=True,
    help_text="Deixe em branco se o autor estiver vivo"  # Guia para o usuário
)
```

A classe Meta configura comportamentos do model:

```python
class Meta:
    verbose_name = "Autor"                    # Nome singular
    verbose_name_plural = "Autores"           # Nome plural
    ordering = ['nome']                       # Ordenação padrão
    indexes = [                               # Índices para performance
        models.Index(fields=['nome']),
        models.Index(fields=['nacionalidade']),
    ]
```

Métodos personalizados adicionam lógica de negócio:

```python
def livros_publicados(self):
    """Retorna a quantidade de livros do autor"""
    return self.livros.count()  # 'livros' vem do related_name no model Livro
```

Model Livro - O Coração do Sistema

O model Livro representa os livros do acervo e demonstra relacionamentos importantes:

```python
class Livro(models.Model):
    # Relacionamento ForeignKey - MUITOS livros para UM autor
    autor = models.ForeignKey(
        Autor,                          # Model relacionado
        on_delete=models.CASCADE,       # O que acontece se autor for deletado
        related_name='livros',          # Nome para a relação reversa
        verbose_name="Autor principal"
    )
```

Explicação detalhada do `related_name`:

- `related_name='livros'` cria uma relação reversa
- De um autor, podemos acessar seus livros: `autor.livros.all()`
- Sem related_name: usaríamos `autor.livro_set.all()`
- É obrigatório quando há múltiplas relações com o mesmo model

Relacionamento ManyToMany - MUITOS livros para MUITOS autores:

```
python
coautores = models.ManyToManyField(
    Autor,                              # Mesmo model Autor
    blank=True,                         # Pode ficar vazio
    related_name='livros_coautor',      # Nome diferente para evitar conflito
    verbose_name="Coautores"
)
```

Campos com validações:

```
python
paginas = models.IntegerField(
    verbose_name="Número de páginas",
    validators=[models.validators.MinValueValidator(1)]  # Mínimo 1 página
)

quantidade_total = models.IntegerField(
    default=1,
    verbose_name="Quantidade total de exemplares",
    validators=[models.validators.MinValueValidator(0)]  # Não pode ser
negativo
)
```

Métodos de negócio para controle de estoque:

```
python
def emprestar(self):
    """Método para emprestar um exemplar"""
    if self.disponivel_para_emprestimo():
        self.quantidade_disponivel -= 1
        self.save()
        return True
```

```
    return False
```

Model Emprestimo - Controle de Transações

O model Emprestimo registra as transações de empréstimo e demonstra relacionamentos complexos:

```
python
```

```
class Emprestimo(models.Model):
    # Relacionamento com Livro
    livro = models.ForeignKey(
        Livro,
        on_delete=models.CASCADE,        # Se livro for deletado, empréstimos também
        related_name='emprestimos',      # De um livro: livro.emprestimos.all()
        verbose_name="Livro emprestado"
    )

    # Relacionamento com User (model built-in do Django)
    usuario = models.ForeignKey(
        User,
        on_delete=models.CASCADE,        # Se usuário for deletado, empréstimos também
        related_name='emprestimos',      # De um usuário: user.emprestimos.all()
        verbose_name="Usuário"

    )
```

Campos de data com comportamentos diferentes:

```
python
```

```
data_emprestimo = models.DateTimeField(
    auto_now_add=True,        # Define automaticamente na criação
    verbose_name="Data do empréstimo"
)

data_devolucao_prevista = models.DateField(
    verbose_name="Data prevista para devolução"
```

```
    # Definida manualmente ou calculada
)

data_devolucao_real = models.DateField(
    null=True,                # Pode ser NULL no banco
    blank=True,               # Pode ficar vazio em formulários
    verbose_name="Data real de devolução"
)
```

Método save() personalizado para lógica automática:

```
python
def save(self, *args, **kwargs):
    """Atualiza status automaticamente baseado nas datas"""
    from datetime import date

    if self.data_devolucao_real:
        self.status = 'devolvido'
    elif self.data_devolucao_prevista and date.today() >
self.data_devolucao_prevista:
        self.status = 'atrasado'
    else:
        self.status = 'ativo'

    super().save(*args, **kwargs)  # Chama o save original
```

---

3. Migrações - Controle de Versão do Banco de Dados

As migrações são o sistema de versionamento do Django para o esquema do banco de dados. Elas permitem evoluir a estrutura do banco ao longo do tempo sem perder dados.

Comandos Básicos de Migração:

```
bash
```

```
# Cria arquivos de migração baseados nas mudanças dos models
python manage.py makemigrations biblioteca

# Aplica as migrações ao banco de dados
python manage.py migrate biblioteca

# Mostra o estado das migrações
python manage.py showmigrations biblioteca
```

Fluxo típico de desenvolvimento:

1. Modificar os models em `models.py`
2. Criar migração: `makemigrations`
3. Aplicar migração: `migrate`
4. Testar as mudanças

Exemplo Prático de Evolução do Model:

Suponha que queremos adicionar um campo `edicao` ao Livro:

```
python
# No model Livro, adicione:
edicao = models.PositiveIntegerField(
    default=1,
    verbose_name="Edição",
    help_text="Número da edição do livro"
)
```

Comandos para aplicar a mudança:

```
bash
# Criar migração
python manage.py makemigrations biblioteca
# Saída: Migrations for 'biblioteca':
#   biblioteca/migrations/0002_livro_edicao.py
#     - Add field edicao to livro

# Aplicar migração
```

```
python manage.py migrate biblioteca
# Saída: Operations to perform:
#   Apply all migrations: biblioteca
# Running migrations:
#   Applying biblioteca.0002_livro_edicao... OK
```

Comandos Avançados de Migração:

```
bash
# Ver o SQL que será executado
python manage.py sqlmigrate biblioteca 0002

# Criar migração vazia para modificações manuais
python manage.py makemigrations --empty biblioteca

# Migrar para uma migração específica
python manage.py migrate biblioteca 0001

# Desfazer todas as migrações de um app
python manage.py migrate biblioteca zero
```

---

4. Operações CRUD - Explicações Contextualizadas

CREATE - Criando Registros no Sistema

Cenário: Cadastrando um novo autor e seus livros

```
python
from biblioteca.models import Autor, Livro
from django.contrib.auth.models import User
from datetime import date

# CREATE Autor - Método 1: create() (mais direto)
autor1 = Autor.objects.create(
```

```
    nome="Machado de Assis",
    data_nascimento=date(1839, 6, 21),
    data_falecimento=date(1908, 9, 29),
    nacionalidade="Brasileira",
    genero='M',
    biografia="Fundador da Academia Brasileira de Letras..."
)
print(f"Autor criado: {autor1.nome}")

# CREATE Autor - Método 2: save() (mais controle)
autor2 = Autor(
    nome="Clarice Lispector",
    data_nascimento=date(1920, 12, 10),
    nacionalidade="Brasileira",
    genero='F'
)
autor2.save()  # Só persiste no banco após save()
print(f"Autor criado: {autor2.nome}")

# CREATE Livro com ForeignKey
dom_casmurro = Livro.objects.create(
    titulo="Dom Casmurro",
    isbn="9788535909023",
    autor=autor1,  # Relacionamento ForeignKey
    categoria="romance",
    editora="Companhia das Letras",
    ano_publicacao=1899,
    paginas=256,
    quantidade_total=5,
    quantidade_disponivel=5
)

# CREATE relacionamento ManyToMany
livro_coautor = Livro.objects.create(
    titulo="Antologia Poética",
    isbn="9788535912344",
    autor=autor1,
    editora="Nova Fronteira",
    ano_publicacao=2001,
    paginas=300,
    quantidade_total=2
)
```

```
livro_coautor.coautores.add(autor2)  # Adiciona coautor

print(f"Coautores: {[coautor.nome for coautor in
livro_coautor.coautores.all()]}")
```

READ - Consultando Dados do Sistema

Cenário: Buscando informações para relatórios e funcionalidades

```
python
# READ - Buscar todos os livros
todos_livros = Livro.objects.all()
print(f"Total de livros no acervo: {todos_livros.count()}")

# READ - Buscar com filtro simples
livros_romance = Livro.objects.filter(categoria='romance')
print(f"Livros de romance: {livros_romance.count()}")

# READ - Buscar com filtro em relacionamento (autor)
livros_machado = Livro.objects.filter(autor__nome="Machado de Assis")
print(f"Livros do Machado: {[livro.titulo for livro in livros_machado]}")

# READ - Buscar com filtros avançados (lookups)
from datetime import date
livros_recentes = Livro.objects.filter(ano_publicacao__gte=2000)
print(f"Livros do século 21: {livros_recentes.count()}")

# READ - Buscar livros disponíveis
livros_disponiveis = Livro.objects.filter(quantidade_disponivel__gt=0)
print(f"Livros disponíveis: {livros_disponiveis.count()}")

# READ - Buscar usando related_name (do autor para seus livros)
autor = Autor.objects.get(nome="Machado de Assis")
livros_do_autor = autor.livros.all()  # 'livros' é o related_name
print(f"Machado escreveu {livros_do_autor.count()} livros")

# READ - Consultas complexas com Q objects
from django.db.models import Q
livros_brasileiros_romance = Livro.objects.filter(
    Q(autor__nacionalidade="Brasileira") &
```

```
    Q(categoria="romance")
)
```

UPDATE - Atualizando Registros

Cenário: Atualizando informações e processando devoluções

```
python
# UPDATE - Atualizar um único objeto
livro = Livro.objects.get(titulo="Dom Casmurro")
livro.quantidade_total = 8  # Adquirimos mais exemplares
livro.quantidade_disponivel = 8
livro.save()  # IMPORTANTE: chamar save() para persistir
print(f"Estoque atualizado:
{livro.quantidade_disponivel}/{livro.quantidade_total}")

# UPDATE - Atualizar múltiplos objetos
# Aumentar estoque de todos os livros de uma categoria
Livro.objects.filter(categoria="romance").update(quantidade_disponivel=5)
print("Estoque de romances atualizado")

# UPDATE - Usando métodos personalizados
livro.emprestar()  # Usa nosso método personalizado
print(f"Após empréstimo: {livro.quantidade_disponivel} disponíveis")

# UPDATE - Em lote para performance
from datetime import date, timedelta
# Renovar todos os empréstimos que vencem hoje
emprestimos = Emprestimo.objects.filter(
    data_devolucao_prevista=date.today(),
    status='ativo'
)
for emprestimo in emprestimos:
    emprestimo.renovar()
```

DELETE - Removendo Registros

Cenário: Limpeza de dados e manutenção do sistema

```python
# DELETE - Remover um objeto específico
try:
    livro_obsoleto = Livro.objects.get(titulo="Livro Antigo")
    livro_obsoleto.delete()
    print("Livro removido do acervo")
except Livro.DoesNotExist:
    print("Livro não encontrado")

# DELETE - Remover com verificação de dependências
# Não permitir remover autor se tiver livros
autor = Autor.objects.get(nome="Autor Teste")
if autor.livros.count() == 0:  # Verifica via related_name
    autor.delete()
    print("Autor removido")
else:
    print("Não é possível remover - autor tem livros no acervo")

# DELETE - Remover múltiplos objetos
# Limpar empréstimos muito antigos
from datetime import datetime, timedelta
data_limite = datetime.now() - timedelta(days=365*2)  # 2 anos
emprestimos_antigos = Emprestimo.objects.filter(
    data_emprestimo__lt=data_limite,
    status='devolvido'
)
quantidade_removida = emprestimos_antigos.count()
emprestimos_antigos.delete()

print(f"Removidos {quantidade_removida} empréstimos antigos")
```

Configuração do Superuser

Para acessar o admin, precisamos de um superuser:

```bash
python manage.py createsuperuser
```

```
# O comando solicitará:
# Username: admin
# Email address: admin@biblioteca.com
# Password: ********
# Password (again): ********

# Após criar, execute o servidor:
python manage.py runserver
```

```
# Acesse: http://127.0.0.1:8000/admin
```

Views para Funcionalidades do Sistema

As views conectam os models com a interface do usuário:

python

```python
from django.shortcuts import render, get_object_or_404
from django.db.models import Q
from .models import Livro, Emprestimo
from django.contrib.auth.decorators import login_required

def catalogo_livros(request):
    """
    View para exibir catálogo com filtros e busca
    """
    # Parâmetros da URL
    query = request.GET.get('q', '')  # Termo de busca
    categoria = request.GET.get('categoria', '')  # Filtro de categoria

    # Query inicial - todos os livros
    livros = Livro.objects.all()

    # Aplicar filtros se fornecidos
    if query:
        livros = livros.filter(
            Q(titulo__icontains=query) |
            Q(autor__nome__icontains=query) |  # Busca no relacionamento
            Q(isbn__icontains=query)
```

```
        )

    if categoria:
        livros = livros.filter(categoria=categoria)

    # Estatísticas para o template
    total_livros = livros.count()
    livros_disponiveis = livros.filter(quantidade_disponivel__gt=0).count()

    context = {
        'livros': livros,
        'query': query,
        'categoria_selecionada': categoria,
        'total_livros': total_livros,
        'livros_disponiveis': livros_disponiveis,
    }

    return render(request, 'biblioteca/catalogo.html', context)
```

View para Empréstimos do Usuário

python

```
@login_required
def meus_emprestimos(request):
    """
    View para usuário autenticado ver seus empréstimos
    """
    # Filtra empréstimos do usuário logado
    emprestimos = Emprestimo.objects.filter(
        usuario=request.user
    ).select_related('livro').order_by('-data_emprestimo')

    # Otimização: select_related() evita queries adicionais para livro

    # Calcula multas para empréstimos atrasados
    emprestimos_com_multa = []
    for emp in emprestimos:
        if emp.status == 'atrasado':
            multa = emp.calcular_multa()
            emprestimos_com_multa.append({
```

```
                'emprestimo': emp,
                'multa': multa
            })

    context = {
        'emprestimos': emprestimos,
        'emprestimos_com_multa': emprestimos_com_multa,
    }

    return render(request, 'biblioteca/meus_emprestimos.html', context)
```

Template com Dados dos Models

O template exibe os dados dos models de forma amigável:

```
html
<!-- biblioteca/templates/biblioteca/catalogo.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Catálogo - Biblioteca</title>
</head>
<body>
    <h1>Catálogo de Livros</h1>

    <!-- Filtros e Busca -->
    <form method="get">
        <input type="text" name="q" value="{{ query }}"
               placeholder="Buscar por título, autor ou ISBN...">
        <select name="categoria">
            <option value="">Todas as categorias</option>
            <option value="ficcao"
                    {% if categoria_selecionada == 'ficcao' %}selected{% endif
%}>
                Ficção
            </option>
            <option value="romance"
                    {% if categoria_selecionada == 'romance' %}selected{%
endif %}>
                Romance
```

```
            </option>
        </select>
        <button type="submit">Buscar</button>
    </form>

    <!-- Estatísticas -->
    <div class="estatisticas">
        <p>Total de livros: {{ total_livros }}</p>
        <p>Disponíveis para empréstimo: {{ livros_disponiveis }}</p>
    </div>

    <!-- Lista de Livros -->
    <div class="livros-lista">
        {% for livro in livros %}
        <div class="livro-item">
            <h3>
                <a href="{% url 'detalhes_livro' livro.id %}">
                    {{ livro.titulo }}
                </a>
            </h3>
            <p><strong>Autor:</strong> {{ livro.autor.nome }}</p>
            <p><strong>ISBN:</strong> {{ livro.isbn }}</p>
            <p><strong>Disponibilidade:</strong>
                {% if livro.disponivel_para_emprestimo %}
                    <span class="disponivel">
                        ✅ Disponível ({{ livro.quantidade_disponivel }}
exemplares)
                    </span>
                {% else %}
                    <span class="indisponivel">❌ Indisponível</span>
                {% endif %}
            </p>
        </div>
        {% empty %}
        <p>Nenhum livro encontrado com os filtros aplicados.</p>
        {% endfor %}
    </div>
</body>

</html>
```