Django

Prof. José Matheus

Introdução

O Django foi lançado em 2005 por Adrian Holovaty e Simon Willison.

Inicialmente, era uma ferramenta interna usada pela equipe de desenvolvimento do jornal online Lawrence Journal-World, onde eles trabalhavam.

Em julho de 2005, eles decidiram lançar o Django como um projeto de código aberto.

O nome "Django" é uma homenagem ao guitarrista de jazz Django Reinhardt.

Sua abordagem pragmática, foco na eficiência e no princípio "Don't Repeat Yourself" (DRY) atraíram muitos desenvolvedores.

Atualmente, o Django é mantido pela Django Software Foundation (DSF), uma organização sem fins lucrativos.

Sua versatilidade e recursos integrados, como o ORM e o sistema de administração automático, tornam o desenvolvimento web mais eficiente.

Adrian Holovaty [esq] Simon Willison [dir]

Modelos

Modelos

O que é um modelo?

Os modelos no Django são classes Python que representam a estrutura e o comportamento dos dados em um banco de dados relacional. Eles desempenham um papel fundamental na construção de aplicativos Django, pois definem como os dados são armazenados e manipulados. Os modelos fornecem uma camada de abstração entre o código Python e o banco de dados subjacente, permitindo que os desenvolvedores interajam com os dados usando objetos familiares.

Definindo modelos utilizando classes Python

No Django, os modelos são definidos como subclasses da classe django.db.models.Model. Cada atributo na classe representa um campo no banco de dados. Esses campos podem ter vários tipos, como caracteres, números, datas, relacionamentos e muito mais. Além disso, os modelos podem incluir métodos personalizados para realizar operações específicas nos dados.

Definições

No modelo ao lado, temos as seguintes definições:

Nesse exemplo, temos o modelo Post, que possui os campos title, content e pub_date. O campo title é um CharField, o campo content é um TextField e o campo pub_date é um DateTimeField. O método __str__ é usado para retornar uma representação legível do objeto quando convertido em uma string.

PYTHON

```python
from django.db import models


class Post(models.Model):

   title = models.CharField(max_length=200)

    content = models.TextField()

    pub_date = models.DateTimeField(auto_now_add=True)

def __str__(self):

       return self.nome
```

Modelos

Uma das vantagens do Django é a capacidade de mapear relacionamentos entre modelos de forma fácil e eficiente. Isso permite representar a estrutura e as associações dos dados no banco de dados.

Existem vários tipos de relacionamentos disponíveis no Django, como relacionamentos de um para um (OneToOneField), relacionamentos de um para muitos (ForeignKey) e relacionamentos de muitos para muitos (ManyToManyField). Esses relacionamentos são definidos como campos nos modelos, indicando a associação entre os objetos.

A seguir está um exemplo de mapeamento de um relacionamento de um para muitos usando ForeignKey:

Exemplo de mapeamento de um relacionamento de um para muitos usando ForeignKey

PYTHON

```python
from django.db import models

class Categoria(models.Model):
    nome = models.CharField(max_length=100)

def __str__(self):
    return self.nome

class Produto(models.Model):
    nome = models.CharField(max_length=100)
    categoria = models.ForeignKey(Categoria, on_delete=models.CASCADE)
    preco = models.DecimalField(max_digits=5, decimal_places=2)

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

ORM

O que é ORM?

ORM (Object-Relational Mapping) é uma técnica que permite mapear objetos em uma aplicação para tabelas em um banco de dados relacional, facilitando o acesso e a manipulação dos dados.

Exemplo

Suponha que estamos desenvolvendo um aplicativo de blog usando o Django. Temos uma classe Python chamada "**Post**" que representa um post do blog e queremos armazenar as informações desse post em um banco de dados relacional.

Declaração de um modelo em Django

Usando o ORM do Django, podemos definir o modelo "Post" da seguinte forma:

PYTHON

```python
from django.db import models


class Post(models.Model):
    title = models.CharField( max_length=200)
    content = models.TextField()
    pub_date = models.DateTimeField( auto_now_add=True)
```

Utilizações do ORM

Ao criar uma instância do modelo "Post" e salvá-la no banco de dados, o ORM do Django cuida do processo de mapeamento dos atributos do objeto para as colunas da tabela do banco de dados correspondente. Podemos fazer isso da seguinte maneira:

```
post = Post(title="Título do post", content="Conteúdo do post")

post.save()
```

Além disso, o ORM do Django fornece métodos úteis para consultar e manipular os dados. Por exemplo, podemos recuperar todos os posts do banco de dados da seguinte maneira:

```
posts = Post.objects.all()
```

O ORM traduz essa consulta em uma consulta SQL apropriada para selecionar todos os registros da tabela "Post" e retorna os objetos "Post" correspondentes.

Pré-requisitos de instalação

Python: O Django é um framework desenvolvido em Python, portanto, você precisará ter o Python instalado em seu sistema. Recomenda-se usar a versão 3.x do Python.

Etapas básicas para instalar o Django no sistema:

1. Verifique se o Python está instalado: `python --version`
2. (Opcional) Crie um ambiente virtual: `python -m venv venv` (Isso criará um ambiente virtual chamado "venv")
3. Ative o ambiente virtual: No Windows: `\venv\Scripts\activate` No Mac/Linux: `source -m venv/bin/activate`
4. Instale o Django (Com o ambiente virtual ativado): `pip install django`
5. Verifique a instalação: `django-admin --version`

Iniciando um projeto Django

Para criar seu projeto Django, é bastante simples, basta rodar o seguinte comando: `django-admin startproject meu_projeto .`

(Observação: O ponto no final do comando indica que o projeto deve ser criado no diretório atual.)

Para rodar o servidor de aplicação, basta rodar o seguinte comando: `python manage.py runserver`

Iniciando um projeto Django

Arquivos gerados

manage.py: É um utilitário de linha de comando que facilita a execução de comandos do Django para gerenciar o projeto.

meu_projeto/: É o diretório raiz do projeto, que contém as configurações e os arquivos principais.

meu_projeto/settings.py: É o arquivo de configuração principal do projeto, onde são definidas as configurações globais, como o banco de dados, middleware, aplicativos instalados, etc.

meu_projeto/urls.py: É o arquivo de configuração das URLs do projeto, onde você mapeia as URLs para as views correspondentes.

meu_projeto/asgi.py e meu_projeto/wsgi.py: São arquivos de configuração do servidor para as interfaces ASGI (Asyncronous Server Gateway Interface) e WSGI (Web Server Gateway Interface), respectivamente.

Criando apps Django

Em Django, uma "app" (ou "aplicação") refere-se a um componente modular e reutilizável de um projeto Django. Uma app é um pacote de código que realiza uma funcionalidade específica dentro do projeto.

Para criar um novo componente desse, basta utilizar o comando `python manage.py startapp minha_app` e então um novo app será criado.

(Observação: Certifique-se de estar no diretório raiz do projeto, ou seja, onde se encontra o arquivo **manage.py**)

Criando apps Django

Arquivos gerados

minha_app/: É o diretório da aplicação criada, contendo os arquivos relacionados a essa aplicação.

minha_app/models.py: É onde você define os modelos de dados da aplicação usando a ORM do Django.

minha_app/views.py: É onde você define as views, que são responsáveis por processar as requisições e retornar as respostas.

minha_app/tests: É onde se define testes para os modelos criados

minha_app/migrations: É um diretório onde ficam armazanadas as migrações, o histórico das mudanças do banco de dados

minha_app/admin.py: É onde você pode registrar os modelos para que sejam gerenciados pelo sistema de administração automático do Django.

Criando apps Django

Adicione a aplicação ao projeto:

Abra o arquivo meu_projeto/settings.py.

Na lista INSTALLED_APPS, adicione a string 'minha_app' em uma nova linha.

Migração de dados

Migração de dados

Após mapear nossos modelos, devemos então por o Django ORM para trabalhar utilizando os comandos:

```
python manage.py makemigrations
```

Quando você executa o comando makemigrations, o Django examina as alterações feitas nos modelos e gera arquivos de migração que descrevem essas alterações. Esses arquivos são criados no diretório migrations dentro de cada aplicação do projeto.

```
python manage.py migrate
```

Ao executar o comando migrate, o Django verifica os arquivos de migração existentes e aplica as alterações necessárias ao banco de dados. Isso pode incluir a criação de tabelas, a adição ou remoção de colunas, a criação de índices e outras modificações relacionadas à estrutura do banco de dados.

Migração de dados

E para onde os dados vão?

- Por padrão, o Django configura o banco de dados como SQLite3 quando você cria um novo projeto.

SQLite3: O que é?

- SQLite3 é um sistema de gerenciamento de banco de dados relacional que não requer um servidor separado para funcionar;
- Ele é embutido diretamente nas aplicações, o que torna fácil de usar e é uma escolha comum para desenvolvimento e projetos menores.

Migração de dados

Principais Pontos sobre o SQLite3:

- **Zero Configuração:** Como SQLite3 é um banco de dados embutido, você não precisa configurar um servidor de banco de dados separado. O arquivo db.sqlite3 é criado automaticamente.
- **Simplicidade:** É fácil de configurar e usar, sendo uma escolha excelente para projetos menores e aplicações de desenvolvimento rápido.
- **Transações ACID:** SQLite3 oferece transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade), garantindo integridade e confiabilidade nos dados.
- **Suporte SQL Completo:** Suporta a maioria das instruções SQL padrão, tornando-o poderoso o suficiente para muitas aplicações.
- **Desempenho Adequado:** Para muitos casos de uso, especialmente em projetos menores, o SQLite3 fornece um desempenho adequado.

Migração de dados

Curiosidade: Transações ACID são um conjunto de propriedades cruciais que garantem a consistência e confiabilidade das transações em sistemas de banco de dados. ACID é um acrônimo que representa:

- **Atomicidade (Atomicity):** Uma transação é considerada atômica se todas as suas operações são executadas como uma única unidade indivisível. Isso significa que todas as operações da transação são concluídas com sucesso ou, caso ocorra algum problema, são revertidas para o estado anterior à execução da transação.
- **Consistência (Consistency):** A consistência garante que uma transação leve o banco de dados de um estado consistente para outro consistente. Em outras palavras, a execução de uma transação não viola as regras de integridade do banco de dados.
- **Isolamento (Isolation):** A propriedade de isolamento assegura que os efeitos de uma transação em execução não sejam visíveis para outras transações até que ela seja concluída. Isso evita interferências entre transações concorrentes, mantendo a integridade dos dados.
- **Durabilidade (Durability):** A durabilidade garante que uma vez que uma transação é concluída com sucesso, seus efeitos são permanentes e persistem, mesmo em caso de falha do sistema. Os dados alterados pela transação são armazenados de forma duradoura e não serão perdidos, mesmo após um reinício do sistema.

Django Admin

Django Admin

O Django Admin é uma interface administrativa gerada automaticamente pelo Django para facilitar a administração e manipulação dos dados do aplicativo. Ele oferece uma interface amigável para criar, ler, atualizar e excluir (CRUD) registros de banco de dados sem a necessidade de escrever código personalizado. O Django Admin é uma ferramenta poderosa e flexível, que pode ser personalizada e estendida para atender às necessidades específicas de um aplicativo.

Ao utilizar o Django Admin, os desenvolvedores podem economizar tempo e esforço, pois não precisam criar interfaces de administração personalizadas do zero. Ele fornece uma interface intuitiva baseada em modelos que reflete a estrutura dos modelos de dados definidos na aplicação.

Django Admin

Antes de acessar o Django Admin, devemos criar um usuário, para tanto, digitamos o comando `python manage.py createsuperuser` no terminal e preenchemos as informações necessárias.

Para acessar o Django Admin, por padrão, basta entrar no endereço `http://localhost:8000/admin/` e então preencher os campos correspondentes com as credenciais criadas anteriormente.

Registrando novos modelos no Django Admin

Para tal ação, basta adicionar o seguinte código ao arquivo minha_app/admin.py

PYTHON

```python
from django.contrib import admin
from posts import models


admin.site.register(models.Post)
```

Registrando novos modelos no Django Admin

Há ainda uma segunda maneira de chegar a esse resultado:

PYTHON

```python
@admin.register(Modelo)
class ModeloAdmin(admin.ModelAdmin):
    list_display = ("campo",)
    filter_horizontal = ("campo_m2m",)
    search_fields = ("campo",)
    list_filter = ("campo",)
    autocomplete_fields = ("campo_fk",)
```

Arquivos de mídia

Arquivos de mídia

Passo 1: Configuração

No arquivo **settings.py** do seu projeto Django, você precisa configurar as definições relacionadas aos arquivos de mídia. Defina as configurações **MEDIA_ROOT** e **MEDIA_URL**.

MEDIA_ROOT: Especifica o caminho absoluto para o diretório onde os arquivos de mídia serão armazenados [lembre-se de criar uma pasta na raiz do projeto com o mesmo nome]. Por exemplo:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

MEDIA_URL: Define o URL base para os arquivos de mídia. Por exemplo:

```
MEDIA_URL = '/media/'
```

Arquivos de mídia

Passo 1: Configuração

Também será necessário instalar a seguinte dependência:

```
pip install Pillow
```

A Python Imaging Library (PIL) adiciona recursos de processamento de imagem ao seu interpretador Python. Esta biblioteca oferece amplo suporte a formatos de arquivo, uma representação interna eficiente e recursos de processamento de imagem bastante poderosos.

Arquivos de mídia

Passo 2: Configuração de URL

Adicionar a seguinte configuração ao arquivo meu_projeto/urls.py

PYTHON

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


urlpatterns = [
    # ... suas outras URLs ...
] + static(settings.MEDIA_URL , document_root=settings.MEDIA_ROOT)
```

Arquivos de mídia

Passo 3: Upload de arquivos de mídia

Em seu modelo Django, você precisa definir um campo de arquivo para lidar com o upload de arquivos de mídia. Use a classe FileField ou ImageField do módulo django.db.models.fields

PYTHON

```python
class Produto(models.Model):

    # ... seus outros campos ...

    imagem = models.ImageField( upload_to='produtos/', null=True, blank=True)
```

Arquivos de mídia

Passo 4: Manipulação de arquivos de mídia

Ao receber um arquivo de mídia no seu aplicativo, o Django cuidará automaticamente do processo de armazenamento. O arquivo será salvo no diretório definido em **MEDIA_ROOT** com um nome único.

Dúvidas?

Referências

- **Documentação do Django:** https://docs.djangoproject.com/
- **Documentação do Django Rest Framework:** https://www.django-rest-framework.org/
- **Pillow:** https://pypi.org/project/Pillow/