Django

Prof. José Matheus

Modelagem avançada com Django

Modelagem avançada com Django

A modelagem avançada no Django envolve a criação de modelos mais complexos, adicionando funcionalidades como timestamps automáticos, soft-delete (deleção lógica) e gerenciadores personalizados. Esses elementos proporcionam uma estrutura sólida para o desenvolvimento de aplicações robustas.

Modelagem avançada com Django

Vantagens:

1. **Auditoria de Dados:** Os campos created_at e updated_at fornecem informações cruciais sobre quando um registro foi criado ou modificado, facilitando auditorias e análises temporais;
2. **Deleção Lógica:** O uso de deleted_at e is_active permite a implementação de deleção lógica, preservando os dados historicamente e oferecendo a opção de recuperar registros excluídos;
3. **Reusabilidade de Código:** O modelo base pode ser estendido por outros modelos, promovendo a reutilização de código e a consistência nas funcionalidades temporais.

Modelagem avançada com Django

Desvantagens:

1. **Complexidade Adicional:** A introdução de funcionalidades avançadas pode aumentar a complexidade do modelo. Deve-se ponderar se essa complexidade é justificada para os requisitos do projeto;
2. **Aprendizado Inicial:** Para desenvolvedores menos experientes, entender e implementar esses conceitos pode exigir um tempo adicional de aprendizado.

Deleção lógica

Deleção lógica

A **deleção lógica**, também conhecida como "soft-delete", é uma abordagem na qual os registros em um banco de dados não são removidos fisicamente quando são excluídos, mas sim marcados como inativos ou excluídos lógicamente. Em vez de apagar permanentemente os dados, um indicador, geralmente um campo booleano ou uma coluna de data/hora, é utilizado para marcar o registro como "excluído".

Deleção lógica

Principais características da deleção lógica:

1. **Preservação de Dados:** Em vez de remover permanentemente os registros, a deleção lógica preserva os dados históricos, permitindo a recuperação de registros excluídos, se necessário;
2. **Integridade Referencial:** A deleção lógica facilita a manutenção da integridade referencial, já que os registros excluídos não geram problemas em relacionamentos com outros registros;
3. **Auditoria e Rastreabilidade:** Ao manter registros excluídos marcados como inativos, é possível rastrear quando um registro foi excluído e por quem, o que pode ser útil para fins de auditoria;
4. **Recuperação de Dados:** Em muitos casos, é possível reverter ou recuperar a deleção lógica, restaurando o status do registro para ativo.

Implementando a Deleção lógica

O **BaseManager** personalizado é essencial para filtrar automaticamente registros excluídos, garantindo que queries padrão excluam apenas registros ativos. O **BaseModelQuerySet** é uma extensão do QuerySet padrão, customizado para fornecer métodos específicos do modelo base.

O **BaseModelQuerySet** sobrescreve o método delete() para realizar a deleção lógica, marcando o registro como inativo.

PYTHON

```python
from django.db import models

from django.utils import timezone


class BaseModelQuerySet(models.QuerySet):

    def delete(self):

        self.update(deleted_at=timezone.now(), is_active=False)


class BaseManager(models.Manager):

    def get_queryset(self):

        return   BaseModelQuerySet( self.model, using=self._db).filter(deleted_at__isnull =True, is_active=True)
```

Implementando a Deleção lógica

created_at: Armazena a data e hora de criação do registro;

updated_at: Atualizado automaticamente sempre que o registro é modificado;

deleted_at: Utilizado para soft-delete, marcando a data e hora de exclusão;

is_active: Indica se o registro está ativo ou foi excluído.

Note que se trata de uma classe abstrata, que não pode ser instanciada, apenas herdada.

PYTHON

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

    class Meta:

        abstract = True

                            created_at          =
models.DateTimeField( auto_now_add=True)

                            updated_at          =
models.DateTimeField( auto_now=True)

                            deleted_at          =
models.DateTimeField( editable=False,
blank=True, null=True)

                             is_active          =
models.BooleanField( editable=False,
default=True)



    objects = BaseManager()
```

Implementando a Deleção lógica

PYTHON

```python
def delete(self, **kwargs):
    self.is_active = False
    self.deleted_at = timezone.now()
    self.save()


def hard_delete(self, **kwargs):
    super(BaseModel, self).delete(**kwargs)
```

Autenticação e Autorização

Autenticação e Autorização

Qual é a diferença?

Autenticação: Processo de verificar a identidade do usuário, geralmente utilizando credenciais como nome de usuário e senha.

Autorização: Processo de conceder permissões a usuários autenticados para acessar recursos específicos ou realizar determinadas ações.

Django Rest Framework: Autenticação

Ao utilizar um **ModelViewSet** do DRF, a autorização é simplificada através da especificação de permissões diretamente no ViewSet.

PYTHON

```python
from rest_framework import viewsets, permissions

from minha_app import models

from minha_app.api import serializers


class MeuModeloViewSet(viewsets.ModelViewSet):

    queryset = models.MeuModelo.objects.all()

    serializer_class = serializers.MeuModeloSerializer

    permission_classes = [permissions.IsAuthenticated]
```

Django Rest Framework: Autorização

Para implementar uma ViewSet no DRF com uma rota autenticada apenas para um grupo específico de usuários, você pode seguir os passos ao lado.

IsInSpecificGroup é uma permissão personalizada que herda de BasePermission do DRF. Ela verifica se o usuário pertence ao grupo especificado.

IsInSpecificGroup é usado como parte da lista permission_classes em MeuModeloViewSet. Isso garante que a permissão seja verificada antes de permitir o acesso à view.

PYTHON

```python
from rest_framework import permissions


class IsInSpecificGroup(permissions.BasePermission):

    nome_do_grupo = 'nome_do_seu_grupo'


    def has_permission(self, request, view):

                                            return request.user.groups.filter( name=self.nome_do_grupo).exists()
```

Django Rest Framework: Autorização

1. Criar um Grupo no Django: Antes de testar, certifique-se de ter criado um grupo no Django para o qual deseja restringir o acesso. Isso pode ser feito no Django Admin ou por meio de scripts de migraçãoç

2. Configurar Permissão Personalizada: Crie uma permissão personalizada que verifica se o usuário pertence ao grupo desejado.

PYTHON

```python
from rest_framework import viewsets, permissions

from minha_app import models

from minha_app.api import serializers

from .permissions import IsInSpecificGroup


class MeuModeloViewSet(viewsets.ModelViewSet):

   queryset = models.MeuModelo.objects.all()

   serializer_class = serializers.MeuModeloSerializer

    permission_classes = [permissions.IsAuthenticated,
IsInSpecificGroup]
```

Dúvidas?

Referências

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