Yedinci Bölüm

CRUD İşlemlerinin Uygulanması

Django'da CRUD'a Giriş

CRUD (Oluştur, Oku, Güncelle, Sil) işlemleri web uygulamalarının ayrılmaz bir parçasıdır ve veritabanlarıyla dinamik etkileşimler sağlar. Kapsamlı bir Python çatısı olan Django, ORM (Object-Relational Mapping) katmanı aracılığıyla bu işlemleri etkin bir şekilde destekler. Bu ORM, veritabanı sorgularını Python koduna dönüştürmeyi basitleştirerek veritabanı işlemlerini daha sezgisel hale getirir. Aşağıda, Django'nun CRUD işlevlerini nasıl kolaylaştırdığını inceleyeceğiz.

Django'nun ORM'si: Veritabanı Etkileşimlerini Kolaylaştırma

Django'nun ORM'si, veritabanı ile Python kodu arasında bir aracı görevi görerek geliştiricilerin doğrudan SQL yazmadan veritabanı eylemleri gerçekleştirmesine olanak tanır. Bu kurulum CRUD işlemlerini önemli ölçüde basitleştirir. İşte bu işlemlerin nasıl yönetildiğine dair ayrıntılar:

1. Oluştur

Django'da bir veritabanına yeni girişler eklemek için, bir veritabanı tablosunu temsil eden bir model tanımlayarak başlarsınız:

```python
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    published_date = models.DateTimeField(auto_now_add=True)

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

Yeni bir veritabanı kaydı oluşturmak basittir:

```python
from blog.models import Post

new_post = Post(title='Introduction to Django CRUD', content='Learn about CRUD
    operations in Django.')
new_post.save()  # Saves the new entry to the database
```

2. Okuyun

Django, verileri verimli bir şekilde almak için çeşitli yollar sunar:

```python
# Get a single post by its ID
post = Post.objects.get(pk=1)

# Get all posts
all_posts = Post.objects.all()

# Get posts from a certain year
recent_posts = Post.objects.filter(published_date__year=2022)
```

Bu örnekler, Django'nun karmaşık sorguları nasıl basit ve etkili bir şekilde kolaylaştırdığını vurgulamaktadır.

3. Güncelleme

Django ile bir veritabanı girdisini değiştirmek, nesneyi almayı, özniteliklerini değiştirmeyi ve kaydetmeyi içerir:

```
post = Post.objects.get(pk=1)
post.title = 'Updated Title'
post.save()  # Commits the changes to the database
```

Bu yöntem bir gönderinin başlığını günceller ve yeni başlığı veritabanına kaydeder.

4. Silme

Bir kaydı kaldırmak, nesneyi getirmek ve **.delete() işlevini** çağırmak kadar basittir:

```
post = Post.objects.get(pk=1)
post.delete()  # Removes the post from the database
```

Bu komut, gönderiyi veritabanından etkin bir şekilde siler.

CRUD İşlemlerinde Django'nun ORM'sinin Faydaları

Kullanılması Django'nun ORM için CRUD operasyonlar teklifler önemli avantajlar sağlamaktadır:

- **Soyutlama**: Dönüşümler karmaşık SQL sorgular basit Python koduna dönüştürür.
- **Güvenlik**: SQL ifadelerini sanitize ederek SQL enjeksiyonunu otomatik olarak önler.
- **Sürdürülebilirlik**: Destekler ve KURU Prensip, yapmak uygulamanın bakımı ve güncellenmesi daha kolaydır.
- **Veritabanı Esneklik**: Sorunsuz bir şekilde işler ile Farklı veritabanları, kolay geçiş ve ölçeklenebilirliği kolaylaştırır.

Sonuç

CRUD işlemleri dinamik web uygulamalarının temelini oluşturur ve Django'nun ORM'si bu işlemleri zahmetsizce yönetmek için güçlü ve güvenli bir çerçeve sağlar. Django'yu kullanarak, geliştiriciler doğrudan veritabanı manipülasyonuyla uğraşmak yerine uygulama mantığı geliştirmeye konsantre olabilirler. Django sadece CRUD işlemlerini basitleştirmekle kalmaz, aynı zamanda bunların güvenli ve verimli bir şekilde gerçekleştirilmesini sağlayarak sofistike, veri merkezli uygulamalar geliştiren geliştiriciler için paha biçilmez bir araç haline gelir.

Veri Oluşturma ve Okuma

Veri oluşturma ve okuma, web uygulamalarında temel yeteneklerdir ve kullanıcıların bir veritabanından bilgi girmesini ve almasını sağlayarak kullanıcı etkileşimini ve özelleştirmeyi geliştirir. Python tabanlı bir web çerçevesi olan Django, Nesne İlişkisel Eşleme (ORM) sistemi ile bu alanlarda öne çıkmaktadır. Bu sistem, tipik olarak veritabanı işlemleriyle ilişkili karmaşıklığı basitleştirir. Aşağıda, Django'nun ORM'sini kullanarak verilerin oluşturulmasını ve alınmasını nasıl kolaylaştırdığını inceleyeceğiz.

Django ile Veri Oluşturma

Django'da veri oluşturma, kullanıcı girdilerini toplamayı ve bu bilgileri bir veritabanına kaydetmeyi içerir. Python sınıflarında veritabanı tablolarının şemasını tanımlayan Django modelleri bu süreci kolaylaştırır.

Django Modeli Tanımlama

Bir Django modeli bir veritabanı tablosunu temsil eder ve **django.db.models.Model**'den türeyen bir sınıf olarak tanımlanır. Her model niteliği bir veritabanı alanına karşılık gelir. Örnek olarak bir **Kitap** modelini ele alalım:

```python
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    isbn_number = models.CharField(max_length=20)
    summary = models.TextField()

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

Bu model, başlık, yazar, yayın tarihi, ISBN numarası ve özet gibi alanlara sahip kitaplar için bir veritabanı tablosunun ana hatlarını çizer.

Kayıt Ekleme

Veritabanına bir kayıt eklemek için modelin bir örneğini oluşturur ve **.save()** yöntemini çağırırsınız. İşte yeni bir kitabın nasıl ekleneceği:

```
from myapp.models import Book
from datetime import date

new_book = Book(
    title='Sample Book Title',
    author='Author Name',
    published_date=date.today(),
    isbn_number='1234567890123',
    summary='A brief summary of the book.'
)
new_book.save()  # Saves the new book to the database
```

Bu kod parçacığı, yeni bir **Book** örneği oluşturmayı ve bunu veritabanında gösterir.

Django ile Veri Okuma

Django'da veri okumak veya veritabanını sorgulamak, temiz ve okunabilir bir sözdizimi kullanarak hem basit hem de gelişmiş sorguları destekleyen ORM aracılığıyla gerçekleştirilir.

Basit Alımlar

Bir modelin tüm örneklerini, tek bir örneği veya belirli ölçütlerle eşleşen örnekleri alan basit komutları kullanarak verileri getirebilirsiniz:

```
# Fetch all books
all_books = Book.objects.all()

# Fetch a single book by ID
book = Book.objects.get(id=1)

# Fetch books by a specific author
books_by_author = Book.objects.filter(author='Author Name')
```

Bu örnekler tüm kayıt kümelerine, ID'ye göre tek tek kayıtlara ve belirli koşullara göre filtrelenmiş kayıtlara nasıl erişileceğini göstermektedir.

Gelişmiş Sorgular

Django ayrıca sonuçları sıralama ve birden fazla koşulu zincirleme gibi karmaşık sorgulama özelliklerini de destekler. Örneğin:

```
# Fetch books published after a certain date and sort by title
recent_books = Book.objects.filter(published_date__gt=date(2020, 1, 1)).order_by('title')
```

Bu sorgu, 1 Ocak 2020'den sonra yayınlanan kitapları filtreler ve başlığa göre alfabetik olarak düzenler.

Sonuç

Django'nun ORM'si, özellikle veri oluşturma ve okuma gibi CRUD işlemlerini gerçekleştirmek için güçlü ve kullanıcı dostu bir yöntem sağlar. Django, geliştiricilerin modelleri tanımlamasına ve sezgisel sorgulama yöntemlerini kullanmasına izin vererek, veritabanı etkileşimlerini basit ve verimli hale getirir. Bu, geliştiricilerin veritabanı yönetimi yerine uygulama mantığına daha fazla odaklanmasını sağlayarak Django'yu dinamik, veri odaklı web uygulamaları geliştirmek için tercih edilen bir çerçeve haline getirir.

Verilerin Güncellenmesi ve Silinmesi

Web uygulamalarında, veritabanının verilerin güncel ve doğru durumunu yansıtmasını sağlamak için depolanan bilgileri güncelleme ve silme yeteneği kritik öneme sahiptir. Python tabanlı çerçevesini kullanan Django, Nesne İlişkisel Eşleme (ORM) sistemi aracılığıyla bu yetenekleri geliştirerek geliştiricilerin veritabanı kayıtlarını verimli ve güvenli bir şekilde yönetmesini sağlar. Bu tartışmada Django'nun verilerin güncellenmesini ve silinmesini nasıl kolaylaştırdığı ayrıntılı örnekler ve en iyi uygulamalarla ele alınacaktır.

Django'da Veri Güncelleme

Django'nun ORM'si, mevcut veritabanı girdilerini değiştirme görevini basitleştirir. Bu işlem, veri girişinin getirilmesini, istenen alanların değiştirilmesini ve güncellenen bilgilerin veritabanına geri kaydedilmesini gerektirir.

Veri Güncelleme Örneği

Customer olarak tanımlanmış bir Django modeli düşünün:

```python
from django.db import models

class Customer(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    active = models.BooleanField(default=True)

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

Mevcut bir müşterinin e-postasını güncellemek için prosedür aşağıdaki gibidir:

```python
from myapp.models import Customer

# Locating the customer by their ID
customer = Customer.objects.get(id=1)
customer.email = 'updated.email@example.com'
customer.save()
```

Bu kod bir **Customer** nesnesini alır, **e-posta** özniteliğini günceller ve güncellenmiş nesneyi kaydederek değişiklikleri veritabanında kalıcı hale getirir.

Django'da Veri Silme

Değerli bilgilerin kazara kaybolmasını önlemek için verilerin silinmesine dikkatle yaklaşılmalıdır. Django, kayıtları tek tek veya toplu olarak silmek için etkili yollar sağlar.

Kayıt Silme Örneği

Aynı **Müşteri** modelini kullanarak:

```python
# Selecting the customer by their ID
customer = Customer.objects.get(id=1)
customer.delete()
```

Bu işlem belirtilen müşteriyi getirir ve ardından veritabanından siler. Django SQL silme komutlarını otomatik olarak işler.

Verilerin Güncellenmesi ve Silinmesi için En İyi Uygulamalar

1. Koşullu İşlemler: İş kurallarına veya veri bütünlüğü gereksinimlerine uygunluğu sağlamak için güncellemeleri veya silme işlemlerini gerçekleştirmeden önce kontroller yapmak çok önemlidir.

2. F İfadeleri Kullanarak Güncellemeler: Mevcut değerlerine göre alan güncellemeleri için Django'nun **F** ifadeleri, eş zamanlı durumlarda daha verimli ve güvenli olan doğrudan veritabanı sorgusunda güncellemelere izin verir:

```
from django.db.models import F
from myapp.models import Product

# Apply an increment to the price
Product.objects.filter(id=1).update(price=F('price') + 10)
```

Bu yaklaşım, veritabanındaki fiyat alanını doğrudan değiştirerek performansı artırır ve yarış koşulları olasılığını azaltır.

3. Yumuşak Silme İşlemleri: Kayıtları kalıcı olarak kaldırmak yerine, yumuşak silme işlemlerini uygulamayı düşünün. Bu yaklaşım, kayıtları silinmiş olarak işaretler ve tarihsel referans için veritabanında korur:

```
class Customer(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    active = models.BooleanField(default=True)
    deleted_at = models.DateTimeField(null=True, blank=True)

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

Burada, silme yöntemi kaydı tamamen kaldırmak yerine bir **deleted_at** alanını güncelleyecek şekilde değiştirilmiştir.

Sonuç

Verilerin güncellenmesini ve silinmesini verimli bir şekilde yönetmek, web uygulamalarının doğruluğunu ve bütünlüğünü korumak için esastır. Django'nun ORM'si, güvenlik ve verimliliğe vurgu yaparak bu işlemleri kolaylaştıran güçlü araçlar sağlar. İşlemlerden önce doğrulama gibi en iyi uygulamalara bağlı kalarak, verimli güncelleme mekanizmaları kullanarak ve

yumuşak silmeler sayesinde geliştiriciler uygulamalarının verileri sorumlu ve etkili bir şekilde işlemesini sağlayabilir.

Pratik Örnekler: Blog Oluşturma

Bir blog geliştirmek, web geliştirme becerilerini, özellikle de etkileşimli uygulamalar için çok önemli olan CRUD (Oluştur, Oku, Güncelle, Sil) işlemlerini göstermek için temel bir projedir. Kapsamlı bir Python çatısı olan Django, karmaşık geliştirme görevlerini kolaylaştıran zengin özellikleri nedeniyle bu projeler için son derece uygundur. Bu kılavuz, kurulum, gönderilerin uygulanması, kullanıcı yorumları ve kimlik doğrulama süreçlerini kapsayan Django ile basit bir blog oluşturma sürecini detaylandıracaktır.

İlk Django Proje Kurulumu

İlk olarak, Django'nun pip kullanılarak yüklendiğinden emin olun:

```
pip install django
```

Django yüklendikten sonra projenizi başlatın:

```
django-admin startproject myblog
```

Proje dizininize girin ve blogunuz için bir uygulama oluşturun:

```
cd myblog
python manage.py startapp blog
```

Model Tanımları

Bir blog tipik olarak makaleler veya gönderiler için bir model gerektirir. İsteğe bağlı olarak, kullanıcı profilleri ve yorumlar için modeller etkileşimi artırabilir. Aşağıda bir **Post** modeli örneği verilmiştir:

```python
from django.db import models
from django.contrib.auth.models import User

class Post(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    body = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

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

Bu model, gönderinin başlığı, gövdesi, yazarı ve oluşturma ve güncellemeler için zaman damgaları için alanlar içeren bir yapı sağlar.

Görünümleri ve Şablonları Uygulama

Django görünümleri şablonlara veri aktarımını yönetir ve şablonlar da verileri işler. İşte bir blog için temel görünümler:

```python
from django.views.generic import ListView, DetailView
from .models import Post

class PostListView(ListView):
    model = Post
    template_name = 'blog/post_list.html'
    context_object_name = 'posts'

class PostDetailView(DetailView):
    model = Post
    template_name = 'blog/post_detail.html'
```

Bu görünümler için ilgili şablonların ayarlanması gerekir, örneğin **post_list.html**:

```
{% extends "base.html" %}

{% block content %}
    <h1>Blog Posts</h1>
    <ul>
        {% for post in posts %}
        <li>
            <a href="{% url 'post_detail' post.pk %}">{{ post.title }}</a>
            <p>Written by {{ post.author }} on {{ post.created_at }}</p>
        </li>
        {% endfor %}
    </ul>
{% endblock %}
```

URL Kalıplarını Yapılandırma

Django'daki URL kalıpları, gelen istekleri uygun görünümlere yönlendirir. URL yapılandırmalarını **urls.py** dosyanızda ayarlayın:

```
from django.urls import path
from .views import PostListView, PostDetailView

urlpatterns = [
    path('', PostListView.as_view(), name='post_list'),
    path('post/<int:pk>/', PostDetailView.as_view(), name='post_detail'),
]
```

Kullanıcı Kimlik Doğrulamasını Ayarlama

Etkileşimli özelliklere sahip bloglar için kullanıcı kimlik doğrulaması gereklidir. Oturum açma ve kapatma işlemleri için Django'nun yerleşik sistemini kullanın:

```
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

Sonuç

Django kullanarak bir blog oluşturmak, çerçevenin veritabanı yönetiminden kullanıcı arayüzü ve kimlik doğrulamaya kadar bir web uygulamasının gerekli tüm yönlerini destekleme yeteneğini gösterir. Bu adımları izleyerek, yorumlar ve etiketler gibi daha fazla işlevle genişletilebilecek temel bir blog oluşturabilirsiniz. Django sadece sağlam web uygulaması geliştirmeyi kolaylaştırmakla kalmaz, aynı zamanda basitleştirerek geliştiricilerin en iyi kullanıcı deneyimini yaratmaya odaklanmalarını sağlar.

SORULAR VE CEVAPLAR

1-CRUD kısaltmasının açılımı nedir ve web geliştirmede ne anlama gelir?

CRUD:
Create (Oluştur)
Read (Oku)
Update (Güncelle)
Delete (Sil)
Web geliştirmede, veritabanı ile etkileşim kurmanın dört temel işlemini ifade eder.

2-Django'da "yumuşak silme" (soft delete) nedir ve neden kullanılır?

Yumuşak Silme: Kaydı veritabanından tamamen silmek yerine, bir "silindi" işareti (genellikle bir tarih alanı) ile işaretlenerek pasif hale getirilmesidir. Veri kurtarma veya geçmişe dönük raporlama gerektiğinde kullanılır.

3-Django'da bir model nesnesini veritabanına kaydetmek için hangi metot kullanılır?

.save() metodu kullanılır