دليل قواعد البيانات و Django

مقدمة عن قواعد البيانات

قواعد البيانات هي أنظمة منظمة لتخزين وإدارة واسترجاع البيانات بطريقة فعالة ومنظمة. تعتبر قواعد البيانات العمود الفقري لمعظم التطبيقات الحديثة، حيث توفر طريقة موثوقة وآمنة لحفظ المعلومات والوصول إليها عند الحاجة.

في عالم تطوير الويب، تلعب قواعد البيانات دوراً محورياً في تخزين بيانات المستخدمين، المحتوى، الإعدادات، وكافة المعلومات التي يحتاجها التطبيق للعمل بشكل صحيح. ومع تطور التقنيات، ظهرت أنواع مختلفة من قواعد البيانات لتلبية احتياجات متنوعة.

أهمية قواعد البيانات:

- تنظيم وهيكلة البيانات بطريقة منطقية
- ضمان سلامة وأمان البيانات
- إمكانية الوصول السريع للمعلومات
- دعم العمليات المتزامنة لعدة مستخدمين
- النسخ الاحتياطي والاستعادة

الفرق بين أنواع قواعد البيانات (SQL وNoSQL)

تنقسم قواعد البيانات الحديثة إلى فئتين رئيسيتين: قواعد البيانات العلائقية (SQL) وقواعد البيانات غير العلائقية (NoSQL). لكل نوع خصائصه ومميزاته واستخداماته المحددة.

قواعد البيانات العلائقية (SQL)

قواعد البيانات العلائقية تعتمد على النموذج العلائقي لتنظيم البيانات، حيث يتم تخزين البيانات في جداول مترابطة. تستخدم لغة SQL (Structured Query Language) للتعامل مع البيانات.

خصائص قواعد البيانات العلائقية:

- **الهيكل الثابت:** تتطلب تعريف مخطط محدد مسبقاً
- **العلاقات:** دعم العلاقات المعقدة بين البيانات
- **ACID:** ضمان خصائص الذرية والاتساق والعزل والديمومة
- **الاستعلامات المعقدة:** دعم الاستعلامات المتقدمة والربط

أمثلة على قواعد البيانات العلائقية: MySQL، PostgreSQL، SQLite، Oracle، SQL Server

قواعد البيانات غير العلائقية (NoSQL)

قواعد البيانات غير العلائقية صُممت للتعامل مع البيانات غير المهيكلة أو شبه المهيكلة، وتوفر مرونة أكبر في تخزين البيانات وقابلية توسع أفضل.

أنواع قواعد البيانات غير العلائقية:

- **قواعد البيانات الوثائقية:** MongoDB، CouchDB
- **قواعد البيانات المفتاح-قيمة:** Redis، DynamoDB
- **قواعد البيانات العمودية:** Cassandra، HBase
- **قواعد بيانات الرسوم البيانية:** Neo4j، Amazon Neptune

مقارنة شاملة بين SQL وNoSQL

المعيار	قواعد البيانات العلائقية (SQL)	قواعد البيانات غير العلائقية (NoSQL)
هيكل البيانات	مهيكل، جداول مع صفوف وأعمدة	مرن، وثائق، مفتاح-قيمة، أو هياكل أخرى
المخطط	مخطط ثابت محدد مسبقاً	مخطط مرن أو بدون مخطط
لغة الاستعلام	SQL موحدة	متنوعة حسب النوع
قابلية التوسع	توسع عمودي (عادة)	توسع أفقي
الاتساق	اتساق قوي (ACID)	اتساق نهائي (عادة)
الاستخدام الأمثل	التطبيقات المعقدة، الأنظمة المالية	البيانات الضخمة، التطبيقات السريعة

طرق الوصول إلى قاعدة البيانات باستخدام Django

يوفر Django نظاماً قوياً ومرناً للتعامل مع قواعد البيانات من خلال ORM (Object-Relational Mapping) الخاص به. هذا النظام يسمح للمطورين بالتعامل مع قاعدة البيانات باستخدام كائنات Python بدلاً من كتابة استعلامات SQL مباشرة.

نماذج Django (Models)

النماذج في Django هي فئات Python تمثل جداول قاعدة البيانات. كل نموذج يرث من `django.db.models.Model` ويحدد حقول الجدول وسلوكياته.

مثال على نموذج بسيط:

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    age = models.IntegerField()
    created_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.name

    class Meta:
        verbose_name = "طالب"
        verbose_name_plural = "الطلاب"
```

أنواع الحقول الشائعة في Django

- **CharField:** للنصوص القصيرة مع حد أقصى للطول
- **TextField:** للنصوص الطويلة
- **IntegerField:** للأرقام الصحيحة
- **FloatField:** للأرقام العشرية
- **BooleanField:** للقيم المنطقية (صح/خطأ)
- **DateField:** للتواريخ
- **DateTimeField:** للتاريخ والوقت
- **EmailField:** للبريد الإلكتروني مع التحقق
- **ForeignKey:** للعلاقات واحد لمتعدد
- **ManyToManyField:** للعلاقات متعدد لمتعدد

مجموعات الاستعلام (QuerySets)

QuerySets هي الطريقة الأساسية للاستعلام عن البيانات في Django. تمثل مجموعة من الكائنات من قاعدة البيانات ويمكن تنقيحها وترشيحها وترتيبها.

خصائص QuerySets:

- **التقييم المؤجل:** لا يتم تنفيذ الاستعلام حتى الحاجة الفعلية للبيانات
- **القابلية للتسلسل:** يمكن ربط عدة عمليات معاً
- **التخزين المؤقت:** النتائج تُخزن مؤقتاً لتحسين الأداء

مدير النماذج (Model Manager)

كل نموذج في Django يحتوي على مدير افتراضي يسمى `objects` والذي يوفر واجهة للتعامل مع قاعدة البيانات. يمكن إنشاء مديرين مخصصين

لإضافة وظائف خاصة.

```
class StudentManager(models.Manager):
    def adults(self):
        return self.filter(age__gte=18)

    def by_email_domain(self, domain):
        return self.filter(email__endswith=f'@{domain}')

class Student(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    age = models.IntegerField()

    objects = StudentManager()  # المدير المخصص
```

أمثلة على استخدام Django للوصول إلى قاعدة البيانات

إنشاء وحفظ البيانات

الطريقة الأولى: إنشاء كائن وحفظه

```
# إنشاء طالب جديد
student = Student()
student.name = "أحمد محمد"
student.email = "ahmed@example.com"
student.age = 20
student.save()

# أو باستخدام المنشئ
student = Student(
    name="فاطمة علي",
    email="fatima@example.com",
    age=19
)
student.save()
```

الطريقة الثانية: استخدام create()

```
# إنشاء وحفظ في خطوة واحدة
student = Student.objects.create(
    name="سارة أحمد",
    email="sara@example.com",
    age=21
)

# إنشاء متعدد
Student.objects.bulk_create([
    Student(name="محمد حسن", email="mohamed@example.com", age=22)
```

```
    Student(name="نور الدين", email="nour@example.com", age=20),
])
```

استعلام وجلب البيانات

الاستعلامات الأساسية:

```
# جلب جميع الطلاب
all_students = Student.objects.all()

# جلب طالب واحد بالمعرف
student = Student.objects.get(id=1)

# جلب أول طالب أو None إذا لم يوجد
student = Student.objects.first()

# عدد الطلاب
count = Student.objects.count()

# التحقق من وجود طلاب
exists = Student.objects.exists()
```

التصفية والبحث:

```
# تصفية بالعمر
adults = Student.objects.filter(age__gte=18)
young = Student.objects.filter(age__lt=20)

# البحث بالاسم
students_named_ahmed = Student.objects.filter(name__icontains="مد

# تصفية بالبريد الإلكتروني
gmail_users = Student.objects.filter(email__endswith="@gmail.com"
```

```
# تصفيات متعددة
filtered_students = Student.objects.filter(
    age__gte=18,
    email__icontains="university"
).exclude(name__startswith="محمد")
```

ترتيب البيانات

```
# ترتيب حسب العمر (تصاعدي)
students_by_age = Student.objects.order_by('age')

# ترتيب حسب العمر (تنازلي)
students_by_age_desc = Student.objects.order_by('-age')

# ترتيب متعدد المستويات
students_sorted = Student.objects.order_by('age', '-name')

# ترتيب عشوائي
random_students = Student.objects.order_by('?')
```

تحديث البيانات

تحديث كائن واحد:

```
# الطريقة الأولى
student = Student.objects.get(id=1)
student.age = 21
student.save()

# الطريقة الثانية: update_or_create
student, created = Student.objects.update_or_create(
    email="ahmed@example.com",
```

```
        defaults={'age': 21, 'name': 'أحمد محمد الجديد'}
    )
```

تحديث متعدد:

```
# تحديث جميع الطلاب فوق سن معينة
Student.objects.filter(age__gte=20).update(
    status='متقدم'
)

# تحديث باستخدام F expressions
from django.db.models import F
Student.objects.all().update(
    age=F('age') + 1  # زيادة عمر جميع الطلاب بسنة
)
```

حذف البيانات

```
# حذف كائن واحد
student = Student.objects.get(id=1)
student.delete()

# حذف متعدد بالتصفية
Student.objects.filter(age__lt=18).delete()

# حذف جميع البيانات
Student.objects.all().delete()
```

ملاحظة مهمة: *عند حذف البيانات، تأكد من أنك تفهم تأثير ذلك على العلاقات والبيانات المرتبطة. استخدم الحذف بحذر خاصة في بيئة الإنتاج.*

الاستعلامات المتقدمة

استخدام Q objects للاستعلامات المعقدة:

```
from django.db.models import Q

# البحث بشروط OR
students = Student.objects.filter(
    Q(name__icontains="أحمد") | Q(name__icontains="محمد")
)

# شروط معقدة
complex_query = Student.objects.filter(
    Q(age__gte=18) & (Q(email__endswith="@gmail.com") | Q(email__
)
```

التجميع والإحصائيات:

```
from django.db.models import Count, Avg, Max, Min

# إحصائيات أساسية
stats = Student.objects.aggregate(
    total_students=Count('id'),
    average_age=Avg('age'),
    max_age=Max('age'),
    min_age=Min('age')
)

# التجميع حسب حقل معين
age_groups = Student.objects.values('age').annotate(
    count=Count('id')
).order_by('age')
```

العلاقات والربط:

```
# افترض وجود نموذج Course مرتبط بـ Student
class Course(models.Model):
    name = models.CharField(max_length=100)
    students = models.ManyToManyField(Student)

# الوصول للعلاقات
course = Course.objects.get(id=1)
students_in_course = course.students.all()

# الربط العكسي
student = Student.objects.get(id=1)
student_courses = student.course_set.all()

# استخدام select_related و prefetch_related لتحسين الأداء
students_with_courses = Student.objects.prefetch_related('course_
```

نصائح للأداء:

- استخدم `select_related()` للعلاقات واحد لواحد وواحد لمتعدد
- استخدم `prefetch_related()` للعلاقات متعدد لمتعدد
- استخدم `only()` و `defer()` لجلب حقول محددة فقط
- استخدم `bulk_create()` و `bulk_update()` للعمليات الكبيرة

خلاصة

يوفر Django ORM نظاماً شاملاً وقوياً للتعامل مع قواعد البيانات، مما يجعل عملية تطوير التطبيقات أسرع وأكثر أماناً. من خلال استخدام النماذج وQuerySets والمديرين المخصصين، يمكن للمطورين بناء تطبيقات معقدة دون الحاجة لكتابة SQL مباشرة، مع الاستفادة من ميزات الأمان والأداء المدمجة في Django.

الفهم الجيد لهذه المفاهيم والأدوات يمكّن المطور من بناء تطبيقات ويب فعالة وقابلة للتوسع، مع الحفاظ على جودة الكود وسهولة الصيانة.