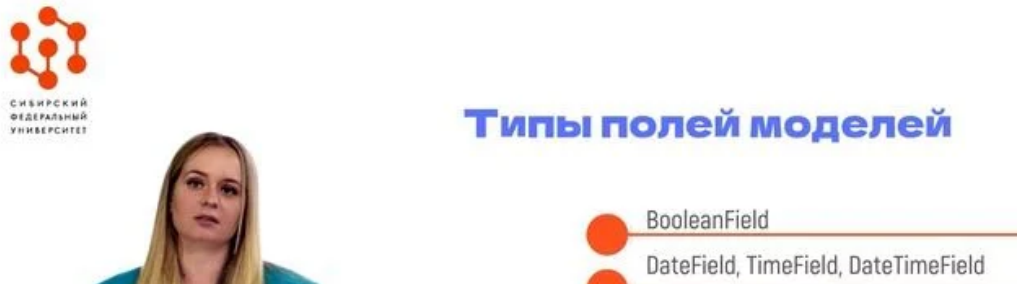
# Разработка программного обеспечения на языке Python



## Лекция 4. Основы баз данных, СУБД

Посмотрите видеоуроки и ответьте на контрольные вопросы после лекции

## Создание базы данных в django. Модели




00:00 / 05:27

В данном разделе рассмотрим понятие модели и его значение при создании таблиц базы данных в приложении Django .

Практически любое веб-приложение, так или иначе, работает с базой данных. Абстракции Object-Relational Mapping (ORM) позволяют большую часть времени работать с сущностями языка программирования. Используя Django ORM, можно создавать таблицы БД и менять в них данные, используя для этого средства языка python - вызовы методов и свойств. При этом Django обеспечивает работу приложения с различными хранилищами данных и СУБД.

Object-Relational Mapping (ORM) - отображение сущностей предметной области и их взаимосвязей в объекты языка программирования

Модели в Django описывают структуру используемых данных

По умолчанию Django в качестве базы данных использует SQLite. Для настройки работы с базами данных в файле settings.py определена переменная DATABASES.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

ЕЕ параметр ENGINE указывает на используемый модуль для доступа к БД. В данном случае это встроенный пакет для sqlite3. Второй параметр - NAME указывает на путь к базе данных. После первого запуска проекта будет создан файл db.sqlite3, который и будет использоваться в качестве базы данных.

Модели в Django описывают структуру используемых таблиц данных. При создании приложения в его каталог добавляется файл models.py, который применяется для определения моделей.

Модель представляет класс, унаследованный от класса models.Model. В качестве примера создадим класс Student, описывающий таблицу студент.

models.py

```
1  from django.db import models
2
3
4  class Student(models.Model):
5      name = models.CharField(max_length=20)
6      age = models.IntegerField()
7
```

В модели определены два поля для описания столбцов таблицы - имя и возраст. Каждая модель сопоставляется с определенной таблицей в базе данных.

Для создания в бд таблицы, которая хранит объекты модели, требуется выполнить миграцию. Необходимо создать миграцию с помощью команды

**python manage.py makemigrations**

В результате в проекте появится файл, описывающий создание таблиц на основе моделей.

```
(venv) C:\djangoExample\example_project>python manage.py makemigrations
Migrations for 'example_app':
  example_app\migrations\0001_initial.py
    - Create model Student
```

Теперь надо выполнить данную миграцию. Для этого выполняется команда

**python manage.py migrate**

После чего в базе данных будут созданы таблицы.

```
(venv) C:\djangoExample\example_project>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, example_app, sessions
```

Для определения моделей мы можем использовать следующие **типы полей**:

- BooleanField: хранит значение True или False.
- Поля DateField, TimeField, DateTimeField: хранят дату и время.
- Поле AutoField: хранит целочисленное значение, которое заполняется автоматически.
- Поля FloatField, IntegerField: хранят числа различного формата.
- Поля CharField, TextField для хранения строковых значений.
- EmailField хранит email-адрес.
- Поля FileField, ImageField хранят данные о файлах и изображении.

Далее рассмотрим организацию связей между таблицами в модели данных. При связывании двух таблиц выделяют основную и дополнительную. Логическое связывание таблиц производится с помощью ключа.

**Существуют 3 основные вида связи:**

- отношения один к одному
- один ко многим
- многие ко многим.

Отношение **один к одному** предполагает, что одна строка из одной таблицы может быть связана только с одной сущностью из другой таблицы. Например, таблица студент. Всю базовую информацию о нем, например, имя, возраст, можно выделить в одну модель, а учетные данные - логин, пароль, и т.д. - в другую.

```
from django.db import models


class Student(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()


class Account(models.Model):
    login = models.CharField(max_length=20)
    password = models.CharField(max_length=20)
    student = models.OneToOneField(Student, on_delete=models.CASCADE, primary_key=True)
```

Для создания отношения один к одному применяется конструктор OneToOneField. Его первый параметр указывает, с какой моделью будет ассоциирована данная сущность (в данном случае ассоциация с моделью Student). Второй параметр on_delete = models.CASCADE говорит, что данные текущей модели (Account) будут автоматически удаляться в случае удаления связанного объекта главной модели (Student).

Параметр **on_delete** указывает на то, какие действия будут происходить с записями в зависимой таблице в случае удаления связанных записей из главной таблицы. Он можем следующие следующие значения:

- models.CASCADE указывает на то, что будет удаляться строка из зависимой таблицы.
- PROTECT: блокирует удаление строки из главной таблицы.
- SET_NULL устанавливает значение NULL.
- SET_DEFAULT: устанавливает значение по умолчанию.
- DO_NOTHING - не производится никаких действий.

При организации связи **один ко многим**, одна главная сущность может быть связаны с несколькими зависимыми сущностями. Например, одна компания может выпускать несколько товаров.

```
from django.db import models


class Company(models.Model):
    name = models.CharField(max_length=30)


class Product(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    name = models.CharField(max_length=30)
    price = models.IntegerField()
```

В данном случае модель Company является главной моделью, а модель Product представляет товар компании и является зависимой моделью. Конструктор типа models.ForeignKey настраивает связь с главной сущностью. Первый параметр указывает, с какой моделью будет создаваться связь - в нашем примере это модель Company.

Связь **многие ко многим** описывает ситуацию, одна строка из каждой из двух связанных таблиц связана с несколькими строками из другой таблицы. Например, один студент может посещать несколько курсов, а один курс могут посещать несколько студентов. Для создания отношения многие ко многим применяется тип ManyToManyField.

```
from django.db import models


class Course(models.Model):
    name = models.CharField(max_length=30)


class Student(models.Model):
    name = models.CharField(max_length=30)
    courses = models.ManyToManyField(Course)
```

Подведем итоги, в данной теме мы рассмотрели создание таблиц базы данных, которые будут использоваться в веб-приложении. Django включает набор инструментов ORM, позволяющих создавать таблицы через модели, представляющие собой классы на языке python. Далее будем изучать способы взаимодействия с данными этих таблиц.

Манипуляция с данными в Django

ПРЕДЫДУЩИЙ ЭЛЕМЕНТ КУРСА

◄ Задание 5. Добавление шаблонов страниц веб-приложения

Перейти на...

СЛЕДУЮЩИЙ ЭЛЕМЕНТ КУРСА

Задание 6. Создание базы данных ►