ТЕХНОАТОМ

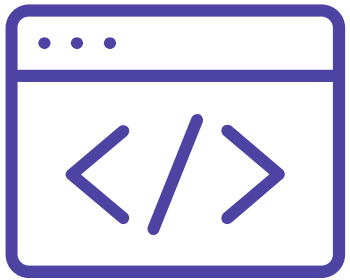
Веб-программирование Python

Лекция 4. Базы данных, модели Django
Часть 1

Михалев Олег

Сегодня

- Базы данных и основы SQL
- Объектно-реляционное отображение
- Модели Django
- Формирование запросов



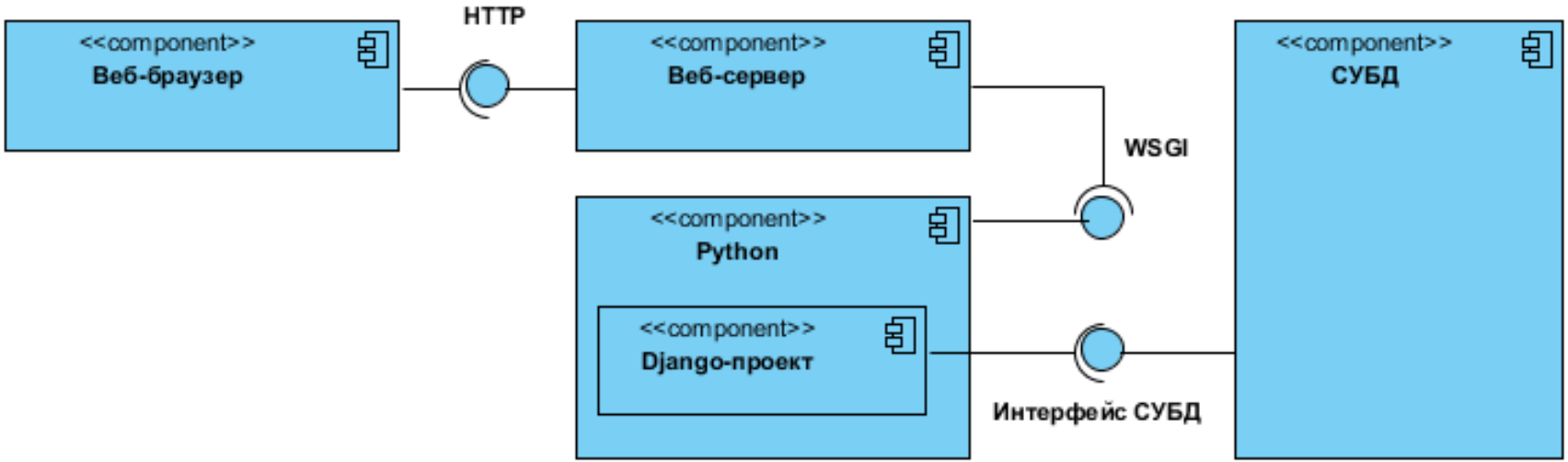
Мы умеем

получать запрос пользователя
обрабатывать поступившие данные
отдавать ответ

Мы хотим

сохранять поступившие данные
управлять сохраненными данными

Базы данных


<<component>>
Веб-браузер
HTTP
<<component>>
Веб-сервер
<<component>>
СУБД
WSGI
<<component>>
Python
<<component>>
Django-проект
Интерфейс СУБД

ТЕХНОАТОМ

БД - совокупность систематизированных данных
СУБД - совокупность средств управления данными

Функции СУБД

- организация журналирования
- организация транзакций
- организация операций над данными
- организация языка запросов

- организация секционирования
- организация кластеризации

СУБД основаны на моделях данных

Модель данных диктует способ организации операций над данными

Реляционная модель

- строгая структура
- операции на основе реляционной алгебры
- кортежи данных

		Заголовок			
		Поле 1	Поле 2	Поле 3	Поле 4
		Имя файла	**Тип**	**Дата модификации**	**Размер**
Тело	Запись 1	1.pdf	application/pdf	2016-09-12	3 457 Кб
	Запись 2	2.pdf	application/pdf	2016-09-19	2 712 Кб
	Запись 3	hw1.zip	application/zip	2016-09-21	473 Кб

Каждое поле характеризуется своим именем и типом
Каждая запись представляет собой кортеж данных

Таблицы - простейшая интерпретация реляционной модели

Язык запросов предоставляет интерфейс для манипуляции данными

SQL - язык запросов
SQL ориентирован специально на реляционные базы данных

Лирическое отступление

NoSQL - совокупность подходов к построению **основанных-не-на-реляционной-модели** баз данных

Ключ-значение
Tarantool, Redis, MemcacheDB

Деревья
MongoDB, CouchDB

Графы
Neo4j, OrientDB

NoSQL имеет ряд преимуществ

- горизонтальное масштабирование
- гибкая структура данных
- производительность (в некоторых случаях)

Классические реляционные СУБД облегчают работу с динамически изменяющейся структурой базы данных, в то время как **использование NoSQL-решений возлагает на плечи разработчика нетривиальные задачи «стратегического» проектирования** (со всеми вытекающими рисками).

SQL databases are like automatic transmission
NoSQL databases are like manual transmission

Dare Obasanjo, 2010

Мы будем использовать классические реляционные СУБД

SQLite, MySQL, PostgreSQL

Мы будем использовать SQL

Определение данных
Манипуляция данными
Управление транзакциями

Определение доступа к данным

Определение данных

- создание объекта
- изменение объекта
- удаление объекта

```
1.CREATE DATABASE atom;

2.CREATE TABLE students (
3.    login VARCHAR(65) NOT NULL
4.    name VARCHAR(255),
5.    summary INTEGER NOT NULL DEFAULT 0,
6.    UNIQUE login_unique (login),
7.    INDEX summary_key (summary)
8.);
```

```
1.ALTER TABLE students
2.     ADD COLUMN points DECIMAL(10, 2) NOT NULL DEFAULT 0,
3.     ADD INDEX points_key (points),
4.     MODIFY COLUMN name VARCHAR(300) NOT NULL,
5.     DROP COLUMN summary;
```

```
1.DROP TABLE students;
```

Манипуляция данными

- **C**reate (создание)
- **R**ead (чтение)
- **U**pdate (обновление)
- **D**elete (удаление)

```
1.INSERT INTO students (login, name)
2.    ("sparrow", "Jack Sparrow");
```

Видите ошибку?

```
1.SELECT * FROM students
2.     WHERE points > 0
3.     ORDER BY points DESC;

4.SELECT points FROM students
5.     WHERE name = "Jack Sparrow";
```

```
1.UPDATE students SET
2.     points = points + 10
3.     WHERE login = "granger";
```

```
1.DELETE FROM students
2.     WHERE login = "sparrow";
```

Управление транзакциями

- BEGIN/START TRANSACTION
- COMMIT
- ROLLBACK

- SAVEPOINT

Транзакция - логическая единица операций с данными

Последовательность операций в транзакции воспринимается целостно

```
1.START TRANSACTION;
2.SELECT @points_copy:=points FROM students
3.     WHERE login = 'alexey';
4.UPDATE students SET
5.     points=@points_copy
6.     WHERE login = 'vasiliy';
7.COMMIT;
```

Требования к транзакциям

- **A**tomicity (атомарность)
- **C**onsistency (согласованность)
- **I**solation (изолированность)
- **D**urability (долговечность)

Атомарность

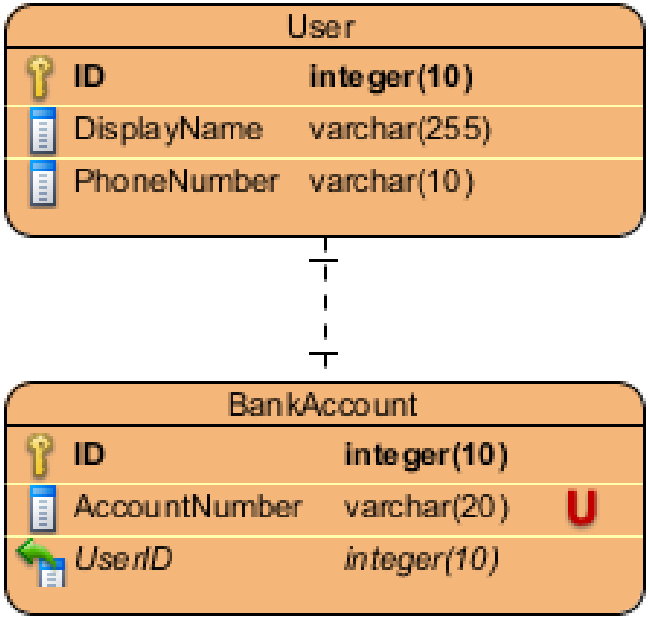
Транзакции не применяются частично

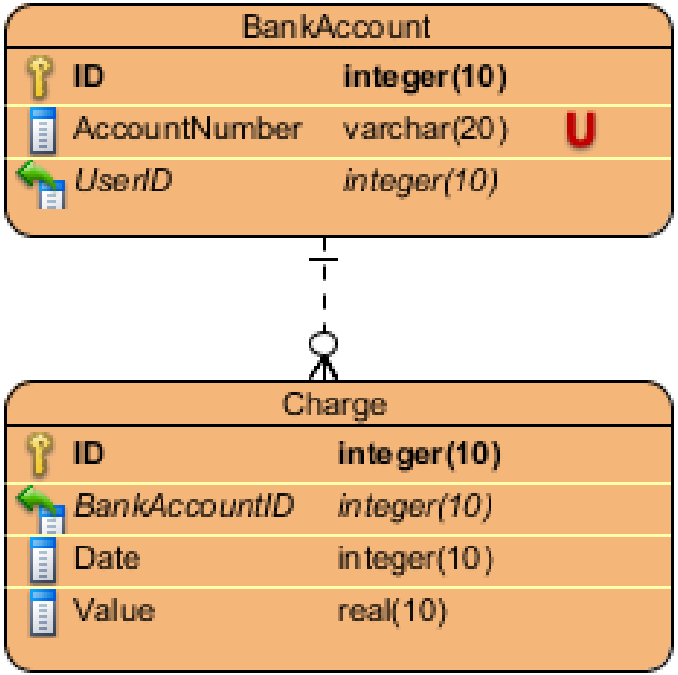
Согласованность

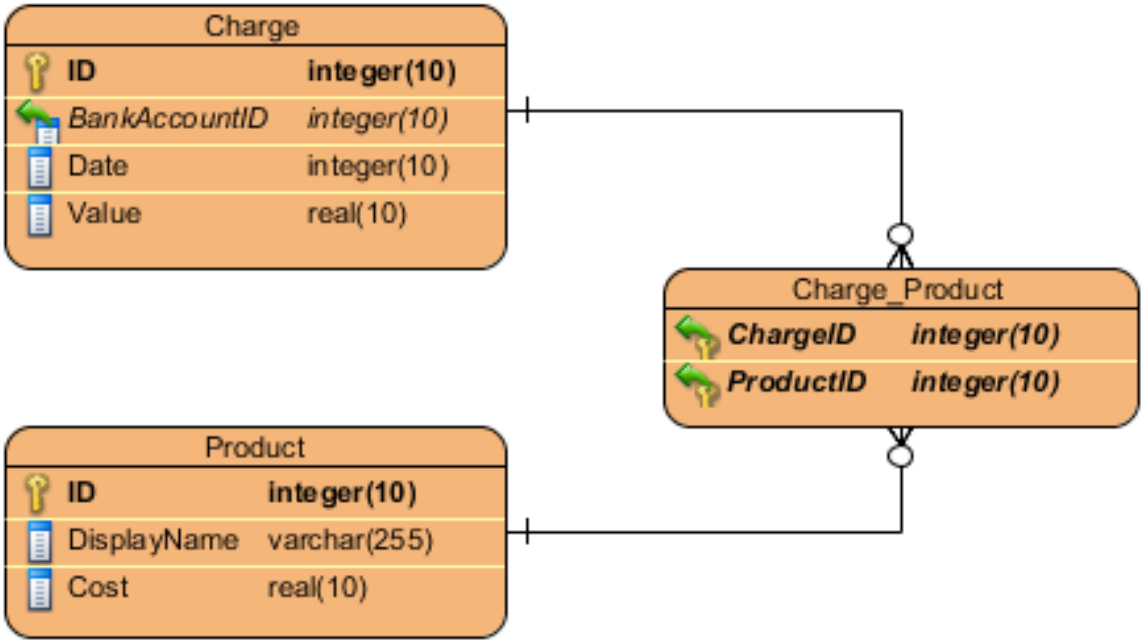
Транзакции не нарушают целостность данных

Говоря о согласованности, нельзя не вспомнить о том, что между различными таблицами возможны связи, и многие СУБД позволяют поддерживать согласованность штатными методами

- связь один-к-одному
- связь один-ко-многим
- связь многие-ко-многим


User
ID
integer(10)
DisplayName
varchar(255)
PhoneNumber
varchar(10)
BankAccount
ID
integer(10)
AccountNumber
varchar(20)
U
UserID
integer(10)


BankAccount
ID
integer(10)
AccountNumber
varchar(20)
U
UserID
integer(10)
Charge
ID
integer(10)
BankAccountID
integer(10)
Date
integer(10)
Value
real(10)


Charge
ID integer(10)
BankAccountID integer(10)
Date integer(10)
Value real(10)
Charge_Product
ChargeID integer(10)
ProductID integer(10)
Product
ID integer(10)
DisplayName varchar(255)
Cost real(10)

Изолированность

Транзакции не влияют друг на друга

Долговечность

Транзакции гарантируют
восстановление после сбоя

Многие реляционные СУБД расширяют базовые возможности SQL

(особенно в части процедур и транзакций)

Большинство реляционных СУБД представлено в Python библиотеками-коннекторами

```
1. >>> from sqlite3 import connect
2. >>> with connect(':memory:') as connection:
3. ...     cursor = connection.cursor()
4. ...     cursor.execute('CREATE TABLE students (name TEXT, points REAL);')
5. ...     cursor.execute('INSERT INTO students VALUES ("sparrow", 5);')
6. ...     for row in cursor.execute('SELECT * FROM students;'):
7. ...             print(row)
8. ...
9. ('sparrow', 5.0)
```

Проблемы такого подхода

- Маппинги кортежей
- Приведение типов данных
- Смешивание SQL-кода и Python

Базы данных + ООП = ORM

ORM

- Примитивы CRUD
- ООП и отсутствие SQL
- Печеньки

SQLAlchemy

https://pypi.python.org/pypi/SQLAlchemy/

http://docs.sqlalchemy.org/en/latest/

SQLAlchemy все еще поддерживает прямые SQL-запросы

(агрегируя в себя все установленные коннекторы)

```
1. >>> from sqlalchemy import create_engine
2. >>> engine = create_engine('sqlite:///:memory:')
3. >>> engine.execute('CREATE TABLE students (name TEXT, points REAL);')
4. >>> engine.execute('INSERT INTO students VALUES ("sparrow", 5);')
5. >>> for row in engine.execute('SELECT * FROM students;'):
6. ...     print(row)
7. ...
8. ('sparrow', 5.0)
```

```
1.>>> from sqlalchemy import *
2.>>> engine = create_engine('sqlite:///:memory:')
3.>>> meta = MetaData(engine)
4.>>> students = Table(
5....     'students',
6....     meta,
7....     Column('name', Text),
8....     Column('points', Numeric)
9.... )
10.>>> students.create()
```

```
1. >>> create_primitive = students.insert()
2. >>> create_primitive.execute(name='sparrow', points=5)
3. >>> read_primitive = students.select(students.c.points > 0)
4. >>> for row in read_primitive.execute():
5. ...     print(row)
6. ('sparrow', Decimal('5.0000000000'))
```

Хм...

```
1. >>> for student in read_primitive.execute():
2. ...     print(student.name)
3. ...     print('%.2f' % student.points)
4. ...
5. sparrow
6. 5.00
```

Хорошо, но можно уж наконец-то полноценные классы и ORM?

```
1.>>> from sqlalchemy import *
2.>>> from sqlalchemy.ext.declarative import declarative_base

3.>>> Model = declarative_base()

4.>>> class Student(Model):
5....     __tablename__ = 'students'
6....     name = Column(Text, primary_key=True)
7....     points = Column(Numeric)
8....
9.>>> engine = create_engine('sqlite:///:memory:')
10.>>> Model.metadata.create_all(engine)
```

```
1. >>> from sqlalchemy.orm import sessionmaker

2. >>> Session = sessionmaker(bind=engine)
3. >>> session = Session()

4. >>> student = Student(name='sparrow', points=5)
5. >>> session.add(student)
6. >>> session.commit()

7. >>> session.query(Student).filter(Student.points > 0).first()
8. <__main__.Student object at 0x033D4770>
```

Все еще неочевидно и местами сложно

У Django есть собственный ORM и модели

Django Models
https://docs.djangoproject.com/en/1.10/topics/db/models/

Django Model Fields
https://docs.djangoproject.com/en/1.10/ref/models/fields/

Модели должны располагаться
в модуле **models** приложения

Подключение к СУБД должно быть настроено в
переменной **DATABASES** из **settings**

TEXHOATOM

```
1. from django.db import models

2. class Student(models.Model):

3.     name = models.CharField(max_length=300)
4.     points = models.DecimalField()

5.     class Meta:
6.         db_table = 'students'
```

При этом модели сразу можно использовать с формами

Django Model Forms
https://docs.djangoproject.com/el/1.10/topics/forms/modelforms/

```
1. from django.forms import ModelForm

2. class StudentForm(ModelForm):

3.     class Meta:
4.         model = Student
5.         fields = ['name', 'points']
```

```
1. form = StudentForm(request.POST)
2. if form.is_valid():
3.     student = form.save()
```

Django позволяет объединять модели связями

```
1. class Teacher(models.Model):
2.     ...

3. class Course(models.Model):
4.     ...
5.     teacher = models.OneToOneField(Teacher, related_name='cource')
6.     start_date = models.DateField()
7.     end_date = models.DateField()

8. class Student(models.Model):
9.     ...
10.    course = models.ForeignKey(Cource, related_name='student')
```

В простейшем случае для связи многие-ко-многим дополнительная таблица будет создана автоматически

```
class Student(models.Model):
        ...
        courses = models.ManyToManyField(
          Course, related_name='students'
        )
```

```
1. >>> from example.models import Student
2. >>> student = Student(name='sparrow', points=5)
3. >>> student.save()
4. >>> for student in Student.objects.all():
5. ...     print(student.name)
6. ...
7. sparrow
```

В результате запроса ORM Django, который представлен множеством значений (методы **all/filter/exclude**), будет сформирован экземпляр класса **QuerySet**

Реальный запроса в СУБД произойдет только в случае обращения к данным **QuerySet**

```
1. >>> student = Student.objects.get(name='sparrow')
2. >>> student.points += 10
3. >>> student.save()
```

```
1. >>> Student.objects.filter(name='sparrow').update(points=0)
2. >>> Student.objects.exclude(name='sparrow').delete()
```

Методы **filter/exclude** могут принимать целое множество именованных аргументов

Некоторые из них - **lookup**

Операторы сравнения
%fieldname%__lt , __gt, __lte, __gte

Операторы строковых функций
%fieldname%__exact, __contains
%fieldname%__startswith, __endswith

Оператор поиска альтернатив
%fieldname%__in

Поля связанных моделей
teacher__name__exact

Часто в методах чтения (для фильтрации) и обновления данных требуется ссылаться на уже сохраненное значение

```
>>> from django.db.models import F
>>> from datetime import timedelta
>>> Course.objects.filter(
...     end_date__gt=F('start_date') + timedelta(weeks=1)
... )
```

Иногда необходимо использовать более сложную логику запросов

```
1. >>> from django.db.models import Q
2. >>> query = ~Q(name='sparrow') | Q(points__lte=0)
3. >>> Student.objects.filter(query)
```

Q(first_condition) & Q(second_condition)
Q(first_condition, second_condition)

first_condition AND second_condition

Q(first_condition) | Q(second_condition)
first_condition OR second_condition

~Q(condition)
NOT(condition)

Продолжение следует

Stand-alone альтернатива, очень похожа на Django ORM

Peewee

https://pypi.python.org/pypi/peewee

http://docs.peewee-orm.com/en/latest/

Спасибо за внимание!

Михалев Олег
mailto:mhalairt@gmail.com

Домашнее задание

ТЕХНОАТОМ

Создать модели **Account** и **Charge** (аналогичные сущностям, используемым в предыдущих домашних заданиях), учесть что **Account** относится к **Charge** как один-ко-многим. Создать формы для этих моделей, используя **Django Model Forms**.

Реализовать представления

Создание банковского счета

Просмотр банковского счета, содержащее:

- кнопку "Добавить приход/расход"
- списки транзакций по счету "Приход"/"Расход«

(запись и чтение в СУБД)

Создание транзакции с привязкой с счету
(ссылок на счет не должно быть на форме)

Django Migrations
https://docs.djangoproject.com/el/1.10/topics/migrations/

Создаем/изменяем модели
Запускаем обновление миграций
python manage.py makemigrations

Обновляем структуру в СУБД
python manage.py migrate