**Дипломная работа по теме:**

**Сравнение производительности и удобства использования различных ORM (Object-Relational Mapping) библиотек:   
Django ORM, SQLAlchemy и Tortoise ORM**

Автор: Шипулин Илья Дмитриевич

(Группа 30/04, направление «Python-разработчик»)

Оглавление

[Оглавление 2](#_Toc182781938)

[Введение 4](#_Toc182781939)

[Структура базы данных 7](#_Toc182781940)

[Основные таблицы и их поля: 7](#_Toc182781941)

[Опишем отношения между таблицами: 10](#_Toc182781942)

[Причины выбора данной структуры: 10](#_Toc182781943)

[Основные процессы, на которых будет построен анализ. 10](#_Toc182781944)

[Django ORM 12](#_Toc182781945)

[Основные возможности Django ORM 12](#_Toc182781946)

[Работа с нашей базой данных с использованием Django ORM 12](#_Toc182781947)

[Использование моделей 13](#_Toc182781948)

[Запросы с фильтрацией 15](#_Toc182781949)

[Обновление и удаление данных 15](#_Toc182781950)

[Работа с миграциями 15](#_Toc182781951)

[Использование админ-панели Django 16](#_Toc182781952)

[Особенности Django ORM при работе с нашей базой данных 16](#_Toc182781953)

[Преимущества использования Django ORM 16](#_Toc182781954)

[Заключение 17](#_Toc182781955)

[SQLAlchemy ORM 18](#_Toc182781956)

[Основные возможности SQLAlchemy ORM 18](#_Toc182781957)

[Работа с нашей базой данных с использованием SQLAlchemy ORM 18](#_Toc182781958)

[Примеры операций с базой данных 19](#_Toc182781959)

[Управление транзакциями 23](#_Toc182781960)

[Особенности SQLAlchemy ORM при работе с нашей базой данных 23](#_Toc182781961)

[Сравнение с Django ORM 24](#_Toc182781962)

[Управление миграциями с помощью Alembic 24](#_Toc182781963)

[Заключение 24](#_Toc182781964)

[Tortoise ORM 26](#_Toc182781965)

[Основные возможности Tortoise ORM 26](#_Toc182781966)

[Работа с нашей базой данных с использованием Tortoise ORM 26](#_Toc182781967)

[Примеры операций с базой данных 27](#_Toc182781968)

[Управление транзакциями 30](#_Toc182781969)

[Особенности Tortoise ORM при работе с нашей базой данных 30](#_Toc182781970)

[Управление миграциями с помощью Aerich 30](#_Toc182781971)

[Сравнение с Django ORM и SQLAlchemy 31](#_Toc182781972)

[Применение в асинхронных фреймворках 31](#_Toc182781973)

[Заключение 32](#_Toc182781974)

[Анализ и сравнение Django ORM, SQLAlchemy ORM и Tortoise ORM 33](#_Toc182781975)

[Сходства 33](#_Toc182781976)

[Различия 33](#_Toc182781977)

[Преимущества каждого ORM 36](#_Toc182781978)

[Django ORM 36](#_Toc182781979)

[SQLAlchemy ORM 36](#_Toc182781980)

[Tortoise ORM 36](#_Toc182781981)

[Заключение 38](#_Toc182781982)

[Список литературы 39](#_Toc182781983)

Введение

В современном мире информационных технологий разработка веб-приложений становится все более сложной и многоуровневой. Одной из ключевых задач при создании таких приложений является эффективное взаимодействие с базами данных. Для упрощения и ускорения этого процесса используются объектно-реляционные отображения (ORM — Object-Relational Mapping), которые позволяют разработчикам работать с базами данных с использованием объектно-ориентированного подхода, не прибегая к написанию сырых SQL-запросов.

Python, как один из самых популярных языков программирования, предлагает широкий выбор ORM-фреймворков, каждый из которых имеет свои особенности, преимущества и области применения. Среди них выделяются три наиболее распространенных и востребованных ORM:

* **Django ORM** — встроенный компонент популярного веб-фреймворка Django, предоставляющий простой и интуитивно понятный интерфейс для работы с базами данных. Он известен своей интеграцией с другими компонентами Django, что ускоряет процесс разработки и обеспечивает согласованность приложения.
* **SQLAlchemy ORM** — мощный и гибкий ORM, который предоставляет разработчикам глубокий уровень контроля над взаимодействием с базой данных. Его особенностью является разделение на высокоуровневый ORM и низкоуровневое ядро, что позволяет оптимизировать запросы и работать с различными СУБД.
* **Tortoise ORM** — современный асинхронный ORM, разработанный для использования с асинхронными веб-фреймворками, такими как FastAPI и Starlette. Он сочетает в себе простоту синтаксиса, похожего на Django ORM, и преимущества асинхронного ввода-вывода, что делает его привлекательным для разработки высоконагруженных приложений.

**Актуальность исследования** обусловлена тем, что выбор подходящего ORM-фреймворка является критически важным для успешной реализации проекта. Неправильный выбор может привести к проблемам с производительностью, масштабируемостью и поддержкой приложения в будущем. В условиях постоянно растущих требований к скорости разработки и качеству программного обеспечения, важно понимать сильные и слабые стороны каждого инструмента.

**Целью данной дипломной работы** является сравнительный анализ трех ORM-фреймворков для Python: Django ORM, SQLAlchemy ORM и Tortoise ORM. Для достижения этой цели будут решены следующие задачи:

1. **Разработка показательной структуры базы данных**, которая включает различные типы отношений и особенностей, таких как наследование, связи один-к-одному, один-ко-многим и многие-ко-многим.
2. **Реализация данной структуры базы данных** с использованием каждого из рассматриваемых ORM-фреймворков, включая определение моделей, настройку связей и выполнение основных операций CRUD.
3. **Анализ особенностей и возможностей каждого ORM**, включая синтаксис, функциональность, производительность, интеграцию с веб-фреймворками и поддержку миграций.
4. **Проведение практических тестов и сравнений**, чтобы выявить преимущества и недостатки каждого ORM в контексте разработанной базы данных.
5. **Предоставление рекомендаций по выбору ORM-фреймворка** в зависимости от требований и условий конкретного проекта.

**Объектом исследования** являются ORM-фреймворки Django ORM, SQLAlchemy ORM и Tortoise ORM.

**Предмет исследования** — методы и инструменты, обеспечивающие взаимодействие между объектно-ориентированным кодом на Python и реляционными базами данных посредством ORM.

**Практическая значимость работы** заключается в предоставлении разработчикам и архитекторам программного обеспечения детального анализа и сравнительной характеристики популярных ORM-фреймворков для Python. Результаты исследования могут быть использованы при выборе наиболее подходящего инструмента для конкретных задач разработки, что способствует повышению эффективности и качества создаваемых приложений.

**Структура работы** состоит из введения, трех основных разделов, заключения и приложений. В первом разделе описывается теоретическая основа ORM, а также подробно рассматривается разработанная структура базы данных. Во втором разделе проводится практическая реализация базы данных с использованием каждого ORM, включая анализ синтаксических и функциональных особенностей. В третьем разделе осуществляется сравнительный анализ ORM-фреймворков, включая тестирование производительности и обсуждение результатов. В заключении подводятся итоги работы и формулируются рекомендации.

Таким образом, данная дипломная работа направлена на углубленное изучение и сравнение трех ключевых ORM-фреймворков для Python, что позволит разработчикам принимать обоснованные решения при выборе инструментов для своих проектов.

Структура базы данных

Для сравнения ORM-фреймворков нам понадобится база данных со сложной структурой, которая включает различные типы отношений: *один-к-одному, один-ко-многим, многие-ко-многим*, а также элементы наследования и полиморфизма. Это позволит продемонстрировать возможности и особенности каждого ORM.

Возьмем для примера следующую структуру базы данных.

Система управления обучением в университете.

Основные таблицы и их поля:

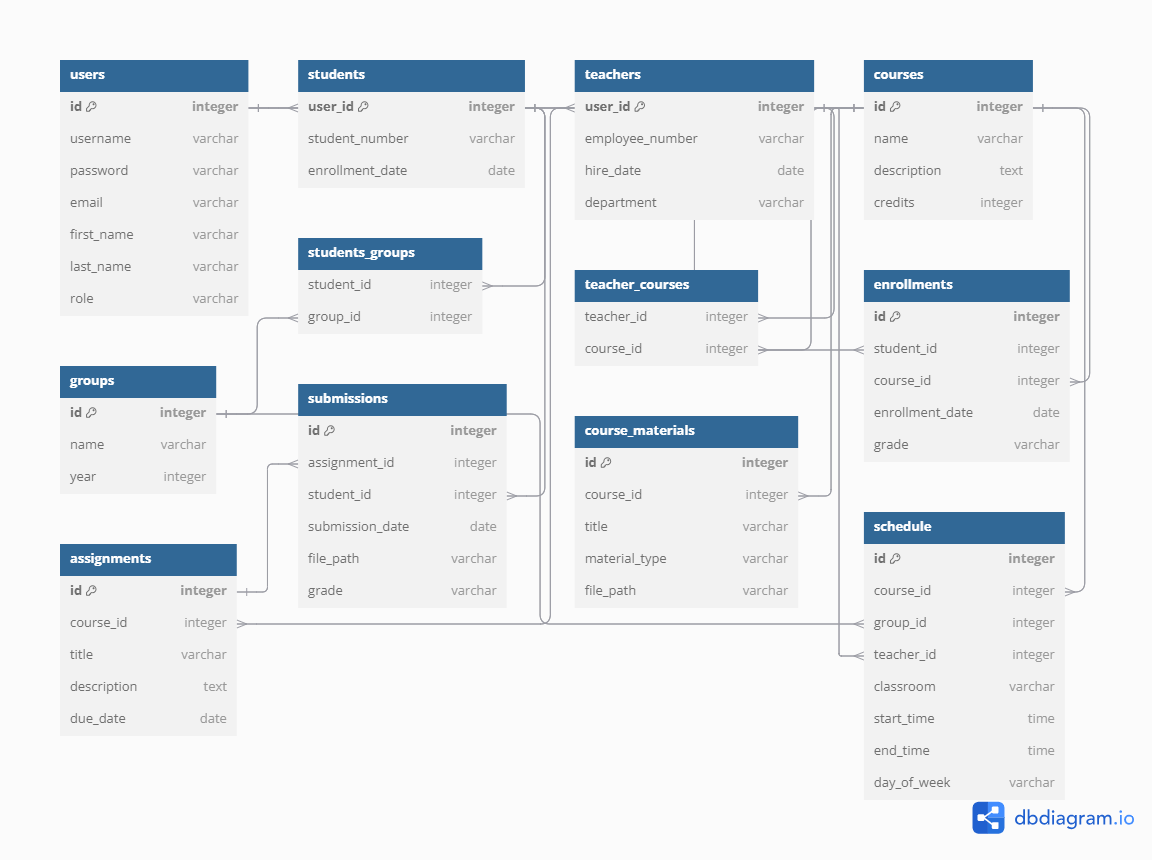
1. **Пользователи (Users)**
   * id (первичный ключ)
   * username (уникальное)
   * password
   * email (уникальное)
   * first\_name
   * last\_name
   * role (студент, преподаватель, администратор)
2. **Студенты (Students)** (*наследует от Пользователи*)
   * user\_id (внешний ключ на Users)
   * student\_number (уникальное)
   * enrollment\_date
3. **Преподаватели (Teachers)** (*наследует от Пользователи*)
   * user\_id (внешний ключ на Users)
   * employee\_number (уникальное)
   * hire\_date
   * department
4. **Курсы (Courses)**
   * id (первичный ключ)
   * name
   * description
   * credits
5. **Группы (Groups)**
   * id (первичный ключ)
   * name
   * year
6. **Записи студентов в группы (StudentsGroups)** (*многие-ко-многим между Студенты и Группы*)
   * student\_id (внешний ключ на Students)
   * group\_id (внешний ключ на Groups)
7. **Преподаватели и Курсы (TeacherCourses)** *(многие-ко-многим между Преподаватели и Курсы)*
   * teacher\_id (внешний ключ на Teachers)
   * course\_id (внешний ключ на Courses)
8. **Записи студентов на курсы (Enrollments)**
   * id (первичный ключ)
   * student\_id (внешний ключ на Students)
   * course\_id (внешний ключ на Courses)
   * enrollment\_date
   * grade
9. **Задания (Assignments)**
   * id (первичный ключ)
   * course\_id (внешний ключ на Courses)
   * title
   * description
   * due\_date
10. **Ответы на задания (Submissions)**
    * id (первичный ключ)
    * assignment\_id (внешний ключ на Assignments)
    * student\_id (внешний ключ на Students)
    * submission\_date
    * file\_path
    * grade
11. **Материалы курса (CourseMaterials)**
    * id (первичный ключ)
    * course\_id (внешний ключ на Courses)
    * title
    * material\_type (лекция, презентация, документ)
    * file\_path
12. **Расписание (Schedule)**
    * id (первичный ключ)
    * course\_id (внешний ключ на Courses)
    * group\_id (внешний ключ на Groups)
    * teacher\_id (внешний ключ на Teachers)
    * classroom
    * start\_time
    * end\_time
    * ****day\_of\_week

Таблица 1. Графическое отображение отношений таблиц

Опишем отношения между таблицами:

* **Один-ко-многим:**
  + Пользователь → Студент/Преподаватель (наследование)
  + Курс → Задания
  + Курс → Материалы курса
  + Преподаватель → Расписание
  + Группа → Расписание
* **Многие-ко-многим:**
  + Студенты ↔ Группы (StudentGroups)
  + Преподаватели ↔ Курсы (TeacherCourses)
  + Студенты ↔ Курсы (Enrollments)

Причины выбора данной структуры:

* **Сложные связи и отношения:** позволяет протестировать возможности ORM в работе с разными типами отношений.
* **Наследование:** Пользователи разделены на подтипы (Студенты, Преподаватели), что позволит проверить поддержку наследования.
* **Работа с датами и временем:** Поля для дат и времени помогут оценить работу ORM с соответствующими типами данных.
* **Файловые поля:** Поля для хранения путей к файлам дадут возможность проверить работу с файловой системой.
* **Уникальные и составные ключи:** Наличие уникальных полей и составных ключей в таблицах связей.

Основные процессы, на которых будет построен анализ.

1. **Создание моделей:** Реализуем модели данных для каждой таблицы с учетом наследования и связей.
2. **Миграции:** Проверим, как каждый ORM справляется с созданием и обновлением схемы базы данных.
3. **CRUD операции:** Реализуем операции создания, чтения, обновления и удаления для основных сущностей.
4. **Запросы с фильтрацией и сортировкой:** Сравним, как каждый ORM позволяет выполнять сложные запросы.
5. **Работа с транзакциями:** Оценим поддержку транзакций и методов отката изменений.
6. **Производительность:** Проведем тесты производительности для массовых операций и сложных запросов.

Используя предложенную структуру базы данных, мы сможем полноценно сравнить Django ORM, SQLAlchemy ORM и Tortoise ORM по различным критериям: функциональность, удобство использования, производительность и другие аспекты, важные для вашей дипломной работы.

Django ORM

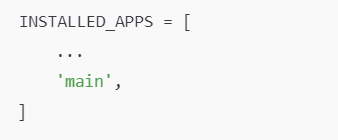
Django ORM (Object-Relational Mapping) — это компонент фреймворка Django, который предоставляет удобный и интуитивно понятный способ взаимодействия с базой данных. Вместо написания сырых SQL-запросов, разработчики могут использовать объекты Python для выполнения операций создания, чтения, обновления и удаления (CRUD) данных.

Основные возможности Django ORM

* **Модели как представление таблиц:** Каждая модель в Django соответствует таблице в базе данных.
* **Управление миграциями:** Django автоматически генерирует и применяет миграции для изменения схемы базы данных.
* **Абстракция запросов:** Позволяет писать сложные запросы с использованием удобного API на Python.
* **Поддержка различных СУБД:** SQLite, PostgreSQL, MySQL, Oracle и другие.
* **Интеграция с админ-панелью Django:** Быстрое создание интерфейса для управления данными.

Работа с нашей базой данных с использованием Django ORM

Давайте рассмотрим, как реализовать предложенную структуру базы данных с помощью Django ORM.

1. **Настройка проекта Django**
   1. Установите Django: pip install Django
   2. Создайте новый проект: django-admin startproject university
   3. Перейдите в директорию проекта и создайте приложение: python manage.py startapp main
2. **Определение моделей**
   1. В файле main/models.py определим модели для каждой таблицы. Полный код представлен в файле models.py
3. **Настройка моделей**
   1. В файле university/settings.py добавьте приложение в INSTALLED\_APPS:   
      
4. **Создание и применение миграций**
   1. Создайте миграции командой: python manage.py makemigrations
   2. Примените миграции командой: python manage.py migrate

Использование моделей

Теперь можно использовать модели для взаимодействия с базой данных.

Примеры операций:

1. **Создание пользователя и студента**

from main.models import User, Student

from django.utils import timezone

# Создание пользователя

user = User.objects.create\_user(username='student1', password='password123', email='student1@example.com', first\_name='Иван', last\_name='Иванов', role='student')

# Создание студента

student = Student.objects.create(user=user, student\_number='S1001', enrollment\_date=timezone.now())

1. **Создание преподавателя**

from main.models import User, Teacher

# Создание пользователя

user = User.objects.create\_user(username='teacher1', password='password123', email='teacher1@example.com', first\_name='Петр', last\_name='Петров', role='teacher')

# Создание преподавателя

teacher = Teacher.objects.create(user=user, employee\_number='E2001', hire\_date=timezone.now(), department='Математика')

1. **Создание курса и добавление преподавателя к курсу**

from main.models import Course, TeacherCourse

# Создание курса

course = Course.objects.create(name='Алгебра', description='Курс по линейной алгебре', credits=5)

# Связь преподавателя с курсом

TeacherCourse.objects.create(teacher=teacher, course=course)

1. **Запись студента на курс**

from main.models import Enrollment

# Запись студента на курс

Enrollment.objects.create(student=student, course=course, enrollment\_date=timezone.now())

1. **Создание задания и добавление материала курса**

from main.models import Assignment, CourseMaterial

# Создание задания

assignment = Assignment.objects.create(course=course, title='Домашнее задание 1', description='Решить задачи из учебника', due\_date='2023-12-01')

# Добавление материала курса

material = CourseMaterial.objects.create(course=course, title='Лекция 1', material\_type='lecture', file='path/to/file.pdf')

1. **Создание расписания**

from main.models import Schedule, Group

# Создание группы

group = Group.objects.create(name='Группа А', year=2023)

# Создание расписания

schedule = Schedule.objects.create(

    course=course,

    group=group,

    teacher=teacher,

    classroom='Аудитория 101',

    start\_time='09:00',

    end\_time='10:30',

    day\_of\_week='Monday'

)

Запросы с фильтрацией

* Получить всех студентов, записанных на определенный курс

students\_in\_course = Student.objects.filter(enrollment\_\_course=course)

* Получить все задания с приближающимся сроком сдачи

from django.utils import timezone

upcoming\_assignments = Assignment.objects.filter(due\_date\_\_gt=timezone.now())

* Получить расписание для определенной группы на понедельник

monday\_schedule = Schedule.objects.filter(group=group, day\_of\_week='Monday')

Обновление и удаление данных

* Обновить оценку студента за курс

enrollment = Enrollment.objects.get(student=student, course=course)

enrollment.grade = 90.5

enrollment.save()

* Удалить материал курса

material = CourseMaterial.objects.get(id=1)

material.delete()

Работа с миграциями

Django ORM упрощает управление схемой базы данных с помощью системы миграций.

* **Создание новых полей или моделей:** Добавьте изменения в models.py и выполните python manage.py makemigrations и python manage.py migrate.
* **Изменение существующих полей:** Например, изменение типа поля или добавление параметров null=True.

Использование админ-панели Django

Админ-панель позволяет управлять данными через веб-интерфейс.

* Регистрация моделей в main/admin.py

from django.contrib import admin

from .models import User, Student, Teacher, Course, Group, Enrollment, Assignment, Submission, CourseMaterial, Schedule

admin.site.register(User)

admin.site.register(Student)

admin.site.register(Teacher)

admin.site.register(Course)

admin.site.register(Group)

admin.site.register(Enrollment)

admin.site.register(Assignment)

admin.site.register(Submission)

admin.site.register(CourseMaterial)

admin.site.register(Schedule)

* Создание суперпользователя

python manage.py createsuperuser

* Запуск сервера и доступ к админ-панели

python manage.py runserver

Перейдите по адресу http://127.0.0.1:8000/admin/ и войдите с учетными данными суперпользователя.

Особенности Django ORM при работе с нашей базой данных

* **Наследование моделей:** Мы использовали AbstractUser для расширения модели пользователя.
* **Связи многие-ко-многим:** Реализованы через промежуточные модели (StudentGroup, TeacherCourse), что позволяет добавлять дополнительные поля и управлять уникальностью.
* **Уникальные ограничения:** Использованы unique\_together в Meta для обеспечения уникальности пар значений.
* **Работа с файлами:** Используются поля FileField для загрузки и хранения файлов.

Преимущества использования Django ORM

* **Скорость разработки**: Меньше кода для взаимодействия с базой данных.
* **Безопасность**: Защита от SQL-инъекций.
* **Поддержка транзакций**: Автоматическое управление транзакциями в блоках атомарности.
* **Интеграция с другими компонентами Django**: Простое использование моделей в представлениях и шаблонах.

Заключение

Используя Django ORM, мы можем эффективно работать с любой структурой базы данных, пользуясь всеми преимуществами высокоуровневого API. Это позволяет сфокусироваться на логике приложения, не отвлекаясь на детали реализации взаимодействия с базой данных.

SQLAlchemy ORM

SQLAlchemy ORM — это популярная библиотека для Python, которая предоставляет полный набор инструментов для работы с реляционными базами данных с использованием объектно-реляционного отображения (ORM). SQLAlchemy предлагает гибкий и мощный способ взаимодействия с базой данных, позволяя разработчикам писать код на Python вместо SQL-запросов.

Основные возможности SQLAlchemy ORM

* **Декларативные модели:** Определение моделей с использованием декларативного стиля, который похож на описание классов.
* **Гибкая конфигурация связей:** Поддержка различных типов отношений между таблицами (один-к-одному, один-ко-многим, многие-ко-многим).
* **Продвинутые возможности запросов:** Предоставляет мощный язык запросов, который позволяет создавать сложные запросы.
* **Поддержка различных СУБД:** Совместим с SQLite, PostgreSQL, MySQL, Oracle и другими.
* **Разделение между ORM и ядром SQLAlchemy:** Позволяет использовать как высокоуровневый ORM, так и низкоуровневые SQL-конструкции.

Работа с нашей базой данных с использованием SQLAlchemy ORM

Давайте рассмотрим, как реализовать структуру данных с помощью SQLAlchemy ORM.

1. **Установка и настройка SQLAlchemy**
   * Установим SQLAlchemy командой pip install sqlalchemy
   * Установим драйвер для выбранной СУБД (Выберем PostgreSQL) pip install psycopg2
2. **Настройка подключения к базе данных**

Создадим файл database.py для настройки подключения

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

# Настройте строку подключения к вашей базе данных

DATABASE\_URI = 'postgresql+psycopg2://user:password@localhost/university\_db'

engine = create\_engine(DATABASE\_URI)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

1. **Определение базовой модели**

Создадим базовый класс для моделей:

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

1. **Определение моделей**

В файле models.py определим модели для каждой таблицы:

from sqlalchemy import (

    Column, String, Integer, Date, DateTime, ForeignKey, Table, Time, Enum, UniqueConstraint

)

from sqlalchemy.orm import relationship

from database import Base

import enum

# Определение перечислений

class UserRole(enum.Enum):

    student = 'student'

    teacher = 'teacher'

    administrator = 'administrator'

Полный код представлен в models.py

1. **Создание базы данных**

После определения моделей необходимо создать таблицы в базе данных:

from database import engine

from models import Base

Base.metadata.create\_all(bind=engine)

Поместим этот код в файл create\_tables.py и запустим командой python create\_tables.py

1. **Использование сессий для взаимодействия с базой данных**

Создадим сессию для выполнения операций:

from database import SessionLocal

session = SessionLocal()

Примеры операций с базой данных

1. Создание пользователя и студента

from models import User, Student

from datetime import date

# Создание пользователя

new\_user = User(

    username='student1',

    password='password123',  # В реальном приложении используйте хэширование

    email='student1@example.com',

    first\_name='Иван',

    last\_name='Иванов',

    role=UserRole.student

)

# Добавление в сессию

session.add(new\_user)

session.commit()

# Создание студента

new\_student = Student(

    user\_id=new\_user.id,

    student\_number='S1001',

    enrollment\_date=date.today()

)

session.add(new\_student)

session.commit()

1. Создание преподавателя

from models import User, Teacher

# Создание пользователя

new\_teacher\_user = User(

    username='teacher1',

    password='password123',

    email='teacher1@example.com',

    first\_name='Петр',

    last\_name='Петров',

    role=UserRole.teacher

)

session.add(new\_teacher\_user)

session.commit()

# Создание преподавателя

new\_teacher = Teacher(

    user\_id=new\_teacher\_user.id,

    employee\_number='E2001',

    hire\_date=date.today(),

    department='Математика'

)

session.add(new\_teacher)

session.commit()

1. Создание курса и добавление преподавателя к курсу

from models import Course, TeacherCourse

# Создание курса

new\_course = Course(

    name='Алгебра',

    description='Курс по линейной алгебре',

    credits=5

)

session.add(new\_course)

session.commit()

# Связь преподавателя с курсом

teacher\_course = TeacherCourse(

    teacher\_id=new\_teacher.user\_id,

    course\_id=new\_course.id

)

session.add(teacher\_course)

session.commit()

1. Запись студента на курс

from models import Enrollment

# Запись студента на курс

enrollment = Enrollment(

    student\_id=new\_student.user\_id,

    course\_id=new\_course.id,

    enrollment\_date=date.today()

)

session.add(enrollment)

session.commit()

1. Создание задания и добавление материала курса

from models import Assignment, CourseMaterial, MaterialType

# Создание задания

assignment = Assignment(

    course\_id=new\_course.id,

    title='Домашнее задание 1',

    description='Решить задачи из учебника',

    due\_date=date(2023, 12, 1)

)

session.add(assignment)

session.commit()

# Добавление материала курса

material = CourseMaterial(

    course\_id=new\_course.id,

    title='Лекция 1',

    material\_type=MaterialType.lecture,

    file\_path='path/to/file.pdf'

)

session.add(material)

session.commit()

1. Создание расписания

from models import Schedule, Group, DayOfWeek

# Создание группы

new\_group = Group(

    name='Группа А',

    year=2023

)

session.add(new\_group)

session.commit()

# Создание расписания

schedule = Schedule(

    course\_id=new\_course.id,

    group\_id=new\_group.id,

    teacher\_id=new\_teacher.user\_id,

    classroom='Аудитория 101',

    start\_time='09:00',

    end\_time='10:30',

    day\_of\_week=DayOfWeek.Monday

)

session.add(schedule)

session.commit()

1. Запросы с фильтрацией

* Получить всех студентов, записанных на определенный курс:

from models import Student

students\_in\_course = session.query(Student).join(Enrollment).filter(Enrollment.course\_id == new\_course.id).all()

* Получить все задания с приближающимся сроком сдачи:

from models import Assignment

from datetime import date

upcoming\_assignments = session.query(Assignment).filter(Assignment.due\_date > date.today()).all()

* Получить расписание для определенной группы на понедельник:

from models import Schedule

monday\_schedule = session.query(Schedule).filter(

    Schedule.group\_id == new\_group.id,

    Schedule.day\_of\_week == DayOfWeek.Monday

).all()

1. Обновление и удаление данных

* Обновить оценку студента за курс:

enrollment = session.query(Enrollment).filter\_by(

    student\_id=new\_student.user\_id,

    course\_id=new\_course.id

).first()

enrollment.grade = 90

session.commit()

* Удалить материал курса

material\_to\_delete = session.query(CourseMaterial).filter\_by(id=material.id).first()

session.delete(material\_to\_delete)

session.commit()

Управление транзакциями

SQLAlchemy автоматически управляет транзакциями через сессии. Если возникает ошибка во время выполнения операций, вы можете откатить изменения:

try:

    # Ваши операции

    session.commit()

except Exception as e:

    session.rollback()

    print(f"Ошибка: {e}")

Особенности SQLAlchemy ORM при работе с нашей базой данных

* **Гибкость в определении отношений:** SQLAlchemy предоставляет детальный контроль над связями между моделями.
* **Уникальные ограничения:** Использование UniqueConstraint в \_\_table\_args\_\_ для обеспечения уникальности пар значений.
* **Работа с перечислениями:** Использование Enum для хранения ограниченного набора значений.
* **Промежуточные таблицы:** Для связей многие-ко-многим можно использовать таблицы (например, student\_groups) или классы (например, TeacherCourse).
* **Отложенная загрузка данных:** SQLAlchemy поддерживает различные стратегии загрузки связанных данных (eager loading, lazy loading).

Сравнение с Django ORM

* **Более низкий уровень абстракции:** SQLAlchemy предоставляет более низкоуровневый доступ к базе данных, что может быть полезно для сложных запросов.
* **Отделение ORM от фреймворка:** SQLAlchemy независим от веб-фреймворков, что делает его гибким для использования в разных проектах.
* **Миграции:** В отличие от Django, SQLAlchemy не предоставляет встроенной системы миграций. Для управления миграциями можно использовать сторонние инструменты, такие как Alembic.

Управление миграциями с помощью Alembic

Установим Alembic командой pip install alembic

Инициализируем Alembic командой alembic init alembic

Это создаст директорию alembic с настройками.

**Настройка Alembic:**

В файле alembic.ini настроим строку подключения:

sqlalchemy.url = postgresql+psycopg2://user:password@localhost/university\_db

В файле alembic/env.py импортируем модели:

from models import Base

target\_metadata = Base.metadata

Создание и применение миграций:

* Создание миграции после изменения моделей

alembic revision --autogenerate -m "Описание изменений"

* Применение миграций

alembic upgrade head

Заключение

Используя SQLAlchemy ORM, мы получаем мощный инструмент для взаимодействия с базой данных, который предоставляет большую гибкость и контроль над запросами и схемой данных.

Сравним Django ORM и SQLAlchemy ORM.

* **Сравнение подхода к определению моделей и связей:** Django ORM и SQLAlchemy имеют разные синтаксисы и возможности в этом аспекте.
* **Оценка управления миграциями:** Django предоставляет встроенные миграции, тогда как в SQLAlchemy необходимо использовать Alembic.
* **Возможность кастомизации запросов:** SQLAlchemy может предоставить более тонкий контроль над генерируемыми SQL-запросами.
* **Интеграция с веб-фреймворками:** SQLAlchemy можно использовать с различными фреймворками, такими как Flask или FastAPI.

Tortoise ORM

Tortoise ORM — это современная **асинхронная** объектно-реляционная модель для Python, разработанная с учетом синтаксиса, похожего на Django ORM, но ориентированная на использование с асинхронными фреймворками, такими как FastAPI и Starlette. Она поддерживает асинхронный ввод-вывод, что позволяет эффективно работать с базой данных без блокировки основного потока приложения.

Основные возможности Tortoise ORM

* **Асинхронность:** Полная поддержка асинхронных операций с базой данных.
* **Декларативные модели:** Определение моделей данных с использованием декларативного стиля.
* **Поддержка различных СУБД:** Совместима с SQLite, PostgreSQL, MySQL и другими.
* **Поддержка отношений:** Один-к-одному, один-ко-многим, многие-ко-многим.
* **Миграции:** Встроенная поддержка миграций через сторонний инструмент Aerich.

**Работа с нашей базой данных с использованием Tortoise ORM**

Рассмотрим, как реализовать предложенную структуру базы данных с помощью Tortoise ORM.

1. **Установка и настройка Tortoise ORM**

* Установим Tortoise ORM и нужный драйвер для выбранной СУБД (например PostgreSQL)

pip install tortoise-orm[asyncpg]

1. **Настройка подключения к базе данных**

В файле нашего приложения, например app.py, настроим Tortoise ORM:

from tortoise import Tortoise, run\_async

async def init():

    await Tortoise.init(

        db\_url='postgres://user:password@localhost:5432/university\_db',

        modules={'models': ['models']}

    )

    await Tortoise.generate\_schemas()

1. **Определение моделей**

Создадим файл models.py и определите модели для каждой таблицы:

from tortoise import fields, models

import enum

# Определение перечислений

class UserRole(enum.Enum):

    student = 'student'

    teacher = 'teacher'

    administrator = 'administrator'

Полный код представлен в models.py

1. **Инициализация схемы базы данных**

В основном файле вашего приложения, после определения моделей, вызовите функцию инициализации:

import asyncio

async def main():

    await init()

    # Ваш код работы с базой данных

if \_\_name\_\_ == '\_\_main\_\_':

    asyncio.run(main())

Примеры операций с базой данных

1. Создание пользователя и студента

from models import User, Student, UserRole

from datetime import date

async def create\_student():

    # Создание пользователя

    user = await User.create(

        username='student1',

        password='password123',  # В реальном приложении используйте хэширование паролей

        email='student1@example.com',

        first\_name='Иван',

        last\_name='Иванов',

        role=UserRole.student

    )

    # Создание студента

    student = await Student.create(

        user=user,

        student\_number='S1001',

        enrollment\_date=date.today()

    )

1. Создание преподавателя

from models import User, Teacher, UserRole

async def create\_teacher():

    # Создание пользователя

    user = await User.create(

        username='teacher1',

        password='password123',

        email='teacher1@example.com',

        first\_name='Петр',

        last\_name='Петров',

        role=UserRole.teacher

    )

    # Создание преподавателя

    teacher = await Teacher.create(

        user=user,

        employee\_number='E2001',

        hire\_date=date.today(),

        department='Математика'

    )

1. Создание курса и добавление преподавателя к курсу

from models import Course, TeacherCourse

async def create\_course\_and\_assign\_teacher(teacher):

    # Создание курса

    course = await Course.create(

        name='Алгебра',

        description='Курс по линейной алгебре',

        credits=5

    )

    # Связь преподавателя с курсом

    await TeacherCourse.create(

        teacher=teacher,

        course=course

    )

1. Запись студента на курс

from models import Enrollment

async def enroll\_student\_to\_course(student, course):

    await Enrollment.create(

        student=student,

        course=course,

        enrollment\_date=date.today()

    )

1. Создание задания и добавление материала курса

from models import Assignment, CourseMaterial, MaterialType

async def create\_assignment\_and\_material(course):

    # Создание задания

    assignment = await Assignment.create(

        course=course,

        title='Домашнее задание 1',

        description='Решить задачи из учебника',

        due\_date=date(2023, 12, 1)

    )

    # Добавление материала курса

    material = await CourseMaterial.create(

        course=course,

        title='Лекция 1',

        material\_type=MaterialType.lecture,

        file\_path='path/to/file.pdf'

    )

1. Создание расписания

from models import Schedule, Group, DayOfWeek

async def create\_schedule(course, group, teacher):

    schedule = await Schedule.create(

        course=course,

        group=group,

        teacher=teacher,

        classroom='Аудитория 101',

        start\_time='09:00',

        end\_time='10:30',

        day\_of\_week=DayOfWeek.Monday

    )

1. Запросы с фильтрацией

* Получить всех студентов, записанных на определенный курс

students\_in\_course = await Student.filter(enrollments\_\_course=course).distinct()

* Получить все задания с приближающимся сроком сдачи

from datetime import date

upcoming\_assignments = await Assignment.filter(due\_date\_\_gt=date.today())

* Получить расписание для определенной группы на понедельник

monday\_schedule = await Schedule.filter(

    group=group,

    day\_of\_week=DayOfWeek.Monday

)

1. Обновление и удаление данных

* Обновить оценку студента за курс

enrollment = await Enrollment.get(student=student, course=course)

enrollment.grade = 90.5

await enrollment.save()

* Удалить материал курса

material = await CourseMaterial.get(id=1)

await material.delete()

Управление транзакциями

Tortoise ORM поддерживает управление транзакциями через контекстные менеджеры.

from tortoise.transactions import in\_transaction

async def transactional\_operation():

    async with in\_transaction():

        # Ваши операции

        pass

Особенности Tortoise ORM при работе с нашей базой данных

* **Асинхронность:** Все операции являются асинхронными, что позволяет эффективно использовать ресурсы и улучшать производительность при большом количестве запросов.
* **Поддержка связей многие-ко-многим:** Реализуется через промежуточные таблицы, которые могут быть созданы автоматически или определены явно.
* **Уникальные ограничения:** Использование unique\_together в Meta для обеспечения уникальности комбинаций полей.
* **Работа с перечислениями:** Поддержка хранения перечислений через CharEnumField.
* **Миграции:** Для управления миграциями в Tortoise ORM используется сторонний инструмент Aerich.

Управление миграциями с помощью Aerich

Aerich — это инструмент для управления миграциями, специально разработанный для Tortoise ORM.

Установим Aerich командой pip install aerich

Выполним инициализацию Aerich командой aerich init -t app.TORTOISE\_ORM

Где app.TORTOISE\_ORM — путь к настройкам Tortoise ORM в вашем приложении.

**Настройка конфигурации**

Добавим в app.py словарь с конфигурацией:

TORTOISE\_ORM = {

    'connections': {'default': 'postgres://user:password@localhost:5432/university\_db'},

    'apps': {

        'models': {

            'models': ['models', 'aerich.models'],

            'default\_connection': 'default',

        }

    }

}

Создание и применение миграций:

* Создание начальной миграции

aerich init-db

* После изменения моделей создадим миграции

aerich migrate --name "описание изменений"

* Применение миграций

aerich upgrade

Сравнение с Django ORM и SQLAlchemy

* **Асинхронность:** В отличие от Django ORM и стандартного SQLAlchemy ORM, Tortoise ORM предоставляет асинхронные операции из коробки.
* **Синтаксис, похожий на Django ORM:** Если вы знакомы с Django ORM, вам будет легко перейти на Tortoise ORM.
* **Миграции:** Использует сторонний инструмент Aerich для управления миграциями, аналогично тому, как SQLAlchemy использует Alembic.
* **Производительность:** Асинхронные операции могут дать преимущества в производительности при высоких нагрузках.

Применение в асинхронных фреймворках

Tortoise ORM отлично интегрируется с такими асинхронными веб-фреймворками, как FastAPI.

Пример интеграции с FastApi:

from fastapi import FastAPI

from tortoise.contrib.fastapi import register\_tortoise

app = FastAPI()

register\_tortoise(

    app,

    db\_url='postgres://user:password@localhost:5432/university\_db',

    modules={'models': ['models']},

    generate\_schemas=False,

    add\_exception\_handlers=True,

)

Заключение

Используя Tortoise ORM, мы можем эффективно работать с любой структурой базы данных в асинхронных приложениях. Это позволяет:

* **Использовать преимущества асинхронности:** Улучшение производительности при работе с большим количеством запросов к базе данных.
* **Писать код в знакомом стиле:** Синтаксис, похожий на Django ORM, упрощает освоение.
* **Управлять миграциями:** Aerich обеспечивает простой способ управления изменениями в схеме базы данных.

Анализ и сравнение Django ORM, SQLAlchemy ORM и Tortoise ORM

Django ORM, SQLAlchemy ORM и Tortoise ORM — это три популярные библиотеки для взаимодействия с реляционными базами данных в Python. Они предоставляют разработчикам удобные инструменты для работы с базами данных, абстрагируя детали SQL и позволяя работать с данными как с объектами Python. Однако у них есть свои особенности, преимущества и области применения. В данном анализе мы рассмотрим основные сходства и различия между ними, а также их преимущества.

Сходства

1. **Объектно-реляционное отображение (ORM):**
   * Все три ORM позволяют определять модели данных как классы Python, которые соответствуют таблицам в базе данных.
   * Поддерживают основные операции CRUD (Create, Read, Update, Delete).
2. **Поддержка различных СУБД:**
   * Поддерживают работу с популярными базами данных, такими как SQLite, PostgreSQL, MySQL и другими.
3. **Определение связей между моделями:**
   * Предоставляют возможность определять отношения один-к-одному, один-ко-многим и многие-ко-многим.
4. **Работа с транзакциями:**
   * Обеспечивают механизм управления транзакциями для обеспечения целостности данных.
5. **Использование декларативного стиля:**
   * Модели определяются с использованием классов и атрибутов, что делает код читабельным и понятным.

Различия

1. **Асинхронность:**
   * **Django ORM:** Синхронный, не поддерживает асинхронные операции из коробки.
   * **SQLAlchemy ORM:** По умолчанию синхронный, но есть отдельная асинхронная версия — SQLAlchemy AsyncIO.
   * **Tortoise ORM:** Изначально разработан как асинхронный ORM, оптимизирован для работы с async-фреймворками.
2. **Интеграция с веб-фреймворками:**
   * **Django ORM:** Плотно интегрирован с фреймворком Django, но может использоваться отдельно с некоторыми ограничениями.
   * **SQLAlchemy ORM:** Независим от веб-фреймворков, широко используется с Flask, FastAPI и другими.
   * **Tortoise ORM:** Разработан для асинхронных фреймворков, таких как FastAPI, Starlette и Sanic.
3. **Гибкость и мощность:**
   * **Django ORM:** Предоставляет высокоуровневый API, удобен для быстрого старта, но имеет ограничения в сложных запросах.
   * **SQLAlchemy ORM:** Очень гибкий и мощный, позволяет писать сложные запросы и использовать низкоуровневые SQL-конструкции.
   * **Tortoise ORM:** Предлагает баланс между простотой Django ORM и асинхронностью, но может уступать SQLAlchemy в гибкости.
4. **Миграции:**
   * **Django ORM:** Встроенная система миграций с автоматическим обнаружением изменений моделей.
   * **SQLAlchemy ORM:** Требует использования стороннего инструмента Alembic для управления миграциями.
   * **Tortoise ORM:** Использует сторонний инструмент Aerich для миграций.
5. **Синтаксис и стиль кодирования:**
   * **Django ORM:** Синтаксис простой и интуитивно понятный, хорошо подходит для начинающих.
   * **SQLAlchemy ORM:** Более сложный синтаксис, требует понимания архитектуры, но предоставляет больше возможностей.
   * **Tortoise ORM:** Синтаксис похож на Django ORM, что облегчает переход для разработчиков.
6. **Сообщество и документация:**
   * **Django ORM:** Большое сообщество, обширная документация и множество сторонних пакетов.
   * **SQLAlchemy ORM:** Активное сообщество, подробная документация, используется в крупных проектах.
   * **Tortoise ORM:** Относительно молодое решение, сообщество растет, документация развивается.
7. **Производительность:**
   * **Django ORM:** Хорошая производительность для большинства задач, но может быть менее эффективен в сложных запросах.
   * **SQLAlchemy ORM:** Высокая производительность, особенно при оптимизации запросов.
   * **Tortoise ORM:** Асинхронность позволяет улучшить производительность в I/O операциях, что важно для высоконагруженных приложений.
8. **Кривая обучения:**
   * **Django ORM:** Низкий порог входа, особенно для тех, кто уже знаком с Django.
   * **SQLAlchemy ORM:** Круче кривая обучения, требует больше времени на освоение.
   * **Tortoise ORM:** Легок в освоении для тех, кто знаком с Django ORM, но требует понимания асинхронного программирования.

Преимущества каждого ORM

Django ORM

* **Быстрый старт и простота использования:** Идеален для быстрых разработок и прототипирования.
* **Интеграция с Django:** Полная совместимость с другими компонентами фреймворка, такими как админ-панель, формы и т.д.
* **Сообщество и экосистема:** Большое количество плагинов и расширений.

SQLAlchemy ORM

* **Гибкость и мощность:** Подходит для сложных и масштабируемых проектов, где требуется тонкий контроль над запросами.
* **Отделение от фреймворка:** Можно использовать с любым веб-фреймворком или вообще без него.
* **Продвинутые возможности:** Поддержка сложных типов данных, кастомизация маппинга, использование сырого SQL при необходимости.

Tortoise ORM

* **Асинхронность:** Отлично подходит для современных асинхронных приложений, обеспечивает эффективное использование ресурсов.
* **Синтаксис, похожий на Django ORM:** Упрощает переход для разработчиков, знакомых с Django.
* **Простота интеграции с асинхронными фреймворками:** Легко интегрируется с FastAPI, Starlette и другими.

|  |  |  |  |
| --- | --- | --- | --- |
| Критерий | Django ORM | SQLAlchemy ORM | Tortoise ORM |
| Асинхронность | Нет | Ограниченно (Доступно через SQLAlchemy AsyncIO) | Да |
| Интеграция с фреймворками | Тесно интегрирован с Django | Независим, используется с Flaskm FastApi и др. | Оптимизирован для асинхронных фреймворков |
| Миграции | Встроенные | Через Alembic | Через Aerich |
| Синтаксис | Простой, декларативный | Более сложный, требует глубинного понимания | Похож на Django ORM |
| Гибкость запросов | Ограничена | Очень высокая | Средняя |
| Сообщество | Большое, активное | Большое, активное | Растущее |
| Производительность | Хорошая | Высокая, особенно при оптимизации | Высокая в асинхронных приложениях |
| Поддержка СУБД | SQLite, PostgreSQL, MySQL, Oracle и др. | Широкий спектр, включая NoSQL через доп. пакеты | SQLite, PostgreSQL, MySQL и др. |
| Транзакции | Да | Да | Да (асинхронные) |
| Кривая обучения | Пологая | Крутая | Пологая для знакомых с Django, но требует знаний async |
| Документация | Обширная, подробная | Подробная, но требует времени на изучение | Хорошая, но может быть менее полной |
| Поддержка асинхронных фреймворков | Нет | Ограничена | Да |

Заключение

Выбор ORM зависит от конкретных потребностей проекта и предпочтений команды разработчиков.

* **Django ORM** подходит для проектов на Django, где важны скорость разработки и простота использования. Он идеален для небольших и средних проектов, где нет необходимости в сложных оптимизациях запросов.
* **SQLAlchemy ORM** рекомендуется для крупных проектов, требующих высокой гибкости и производительности. Его стоит выбирать, если необходимо работать с несколькими СУБД или требуется тонкая настройка запросов.
* **Tortoise ORM** является отличным выбором для асинхронных приложений, особенно если вы используете фреймворки вроде FastAPI. Он сочетает в себе простоту Django ORM и преимущества асинхронности.

При выборе ORM важно учитывать требования проекта, опыт команды и планируемую архитектуру приложения.

Список литературы

1. Документация Django <https://www.djangoproject.com/>
2. Примеры использования Django <https://github.com/topics/django-example>
3. Документация SQLAlchemy: <https://docs.sqlalchemy.org/en/14/>
4. Руководство по Alembic: <https://alembic.sqlalchemy.org/en/latest/>
5. Примеры использования SQLAlchemy: <https://github.com/sqlalchemy/sqlalchemy/tree/main/examples>
6. Документация Tortoise ORM: <https://tortoise.github.io/>
7. Документация Aerich: <https://github.com/tortoise/aerich>
8. Примеры использования Tortoise ORM: <https://github.com/tortoise/tortoise-orm/tree/develop/examples>