**Сравнение производительности и удобства использования различных ORM (Object-Relational Mapping) библиотек: Django ORM, SQLAlchemy и Tortoise ORM**: **Разработать приложения для работы с базами данных с использованием Django ORM, SQLAlchemy и Tortoise ORM, сравнить их.**

Оглавление

[Введение 1](#_Toc182397874)

[Обзор ORM 3](#_Toc182397875)

[Обзор проекта 5](#_Toc182397876)

[Создание базы данных посредством Python в PyCharm 6](#_Toc182397877)

[Описание приложения myshop (Django) 8](#_Toc182397878)

[Описание приложения sqlalchemy\_shop (SQLAlchemy) 13](#_Toc182397879)

[Описание приложения tortoise\_shop (Tortoise) 19](#_Toc182397880)

[Подробное описание процесса тестирования и результатов 24](#_Toc182397881)

[Приложение 1. Список необходимых библиотек 29](#_Toc182397882)

# Введение

* **Актуальность темы**

Изучение ORM (Object-Relational Mapping) библиотек имеет несколько ключевых причин, особенно в контексте растущей популярности Python и веб-разработки:

**1. Упрощение работы с базами данных**

ORM библиотеки позволяют разработчикам работать с базами данных, используя объектно-ориентированное программирование (ООП). Это упрощает взаимодействие с базами данных, так как разработчики могут манипулировать данными как с объектами, избегая необходимости писать сложные SQL-запросы. Это особенно полезно для тех, кто не является экспертом в SQL.

**2. Повышение производительности разработки**

Использование ORM позволяет сократить время на написание кода, так как многие задачи, связанные с управлением базами данных, автоматизируются. Это дает возможность сосредоточиться на бизнес-логике приложения, а не на деталях работы с базами данных.

**3. Портируемость и гибкость**

ORM библиотеки зачастую обеспечивают абстракцию над конкретной СУБД (системой управления базами данных), что позволяет легко менять СУБД без необходимости переписывать код. Например, можно перейти с PostgreSQL на MySQL, изменив лишь конфигурацию, а не все SQL-запросы.

**4. Защита от SQL-инъекций**

ORM обеспечивает защиту от SQL-инъекций, так как позволяет использовать параметры и подготовленные запросы. Это значительно повышает безопасность приложения, особенно в условиях веб-разработки, где атаки на базы данных являются распространенной угрозой.

**5. Растущая популярность Python**

Python продолжает набирать популярность благодаря своей простоте, читаемости и мощным библиотекам. Веб-разработка на Python, особенно с использованием фреймворков, таких как Django и Flask, часто включает в себя использование ORM для взаимодействия с базами данных. Знание ORM становится важным навыком для разработчиков, работающих с этими фреймворками.

**6. Веб-разработка и современные архитектуры**

Веб-приложения становятся все более сложными и многоуровневыми. ORM библиотеки помогают эффективно управлять данными на серверной стороне, обеспечивая возможность быстрого создания и модификации моделей данных, что является критически важным для динамичных веб-приложений.

**7. Сообщество и поддержка**

Популярные ORM библиотеки, такие как SQLAlchemy, Django ORM и Tortoise, имеют активное сообщество и хорошую документацию. Это облегчает новичкам изучение и внедрение ORM в свои проекты, а также поиск решений для возникающих проблем.

**Заключение**

Изучение ORM библиотек является важным шагом для разработчиков, стремящихся эффективно работать с базами данных в контексте веб-разработки на Python. Упрощение взаимодействия с данными, повышение безопасности и производительности, а также растущая популярность Python делают знание ORM необходимым для современного разработчика.

* **Цели и задачи проекта**

Предлагается провести сравнение производительности и удобства использования различных ORM (Object-Relational Mapping) библиотек: Django ORM, SQLAlchemy и Tortoise ORM, а также разработать приложения для работы с базами данных с использованием Django ORM, SQLAlchemy и Tortoise ORM, сравнить их.

# Обзор ORM

* **Что такое ORM?**

**Object-Relational Mapping (ORM)** – это методология, позволяющая разработчикам работать с базами данных, используя объекты из объектно-ориентированного программирования (ООП). ORM служит промежуточным слоем между объектами приложения и реляционными базами данных, что позволяет автоматически преобразовывать данные из базы в объекты и обратно.

* **Преимущества ORM**

1. **Упрощение работы с базами данных**: ORM позволяет разработчикам оперировать объектами вместо написания SQL-запросов, что делает код более читаемым и удобным.
2. **Портируемость**: Благодаря абстрагированию от конкретной СУБД, можно легко менять базу данных без значительных изменений в коде.
3. **Безопасность**: ORM автоматически обрабатывает параметры запросов, что снижает риск SQL-инъекций и повышает безопасность приложений.
4. **Ускорение разработки**: Многие рутинные операции автоматизированы, что позволяет быстрее разрабатывать и тестировать приложения.
5. **Управление миграциями**: ORM часто включает инструменты для управления изменениями в структуре базы данных, что упрощает поддержку.

* **Недостатки ORM**

1. **Производительность**: Для сложных запросов производительность может быть ниже, чем при ручной оптимизации SQL, особенно если ORM генерирует неэффективные запросы.
2. **Сложность в отладке**: Генерируемые ORM запросы могут быть сложны для понимания и отладки, что затрудняет выявление ошибок.
3. **Обучение и адаптация**: Несмотря на простоту, изучение ORM может занять время, особенно для новичков в программировании.
4. **Ограниченная гибкость**: Некоторые сложные операции могут быть трудными для реализации с помощью ORM.

* **Краткий обзор выбранных ORM библиотек**

**Django ORM**

**Особенности**:

* Встроен в фреймворк Django и следует принципам DRY (то есть если есть какая-то логика, которая повторяется в разных частях нашего приложения, то вместо того, чтобы писать ее несколько раз, мы пишем ее один раз и используем везде, где нужно).
* Поддерживает автоматическое создание и управление базами данных через модель, определяемую с помощью Python-классов.

**Преимущества**:

* **Интеграция**: Полная интеграция с Django, что делает его удобным для разработчиков, использующих этот фреймворк.
* **Миграции**: Удобные инструменты для управления миграциями базы данных.
* **Панель администрирования**: Автоматически генерируемая панель для работы с моделями данных.

**Недостатки**:

* **Проблемы с производительностью**: Может генерировать менее оптимизированные запросы для сложных операций.
* **Ограниченность**: Некоторые особенности баз данных могут быть недоступны или сложно реализуемы.

**SQLAlchemy**

**Особенности**:

* Более низкоуровневый, чем Django ORM, и предоставляет высокую степень гибкости.
* Поддерживает как ORM, так и SQL-выражения, позволяя использовать оба подхода.

**Преимущества**:

* **Гибкость**: Позволяет выбирать между объектно-реляционным подходом и использованием чистого SQL.
* **Интуитивно понятный API**: Предоставляет мощный и выразительный API для работы с базами данных.
* **Поддержка различных СУБД**: Большое количество поддерживаемых баз данных и возможность легкой настройки.

**Недостатки**:

* **Сложность**: Более сложен в освоении по сравнению с другими ORM, особенно для новичков.
* **Настройка**: Требует больше времени и усилий на настройку и конфигурацию.

**Tortoise ORM**

**Особенности**:

* Асинхронная ORM, разработанная для работы с Python 3.6+ и асинхронными библиотеками, такими как asyncio.
* Легко интегрируется с фреймворками, поддерживающими асинхронное программирование.

**Преимущества**:

* **Асинхронность**: Поддержка асинхронных операций, что позволяет увеличивать производительность приложений, работающих с большим количеством запросов.
* **Простота использования**: Интуитивно понятный интерфейс, который делает его доступным для новичков.
* **Поддержка множества баз данных**: Хорошая поддержка различных СУБД, включая PostgreSQL, MySQL и SQLite.

**Недостатки**:

* **Молодая библиотека**: Все еще развивается, поэтому может не иметь всех необходимых функций по сравнению с более зрелыми библиотеками.
* **Ограниченная документация**: Документация может быть менее обширной и подробной, чем у более популярных ORM, таких как Django ORM или SQLAlchemy.

# Обзор проекта

* **Структура проекта**

В этой работе был создан проект PG\_Django (django-admin startproject PG\_Django), далее - три приложения (python manage.py startapp <наименование приложения>): myshop, sqlalchemy\_shop, tortoise\_shop для работы с базой данных с использованием Django ORM, SQLAlchemy и Tortoise ORM соответственно, на примере магазина для лекарственных препаратов. Для этих приложений были установлены необходимые библиотеки. В работе используется Python 3.12 и зависимости:

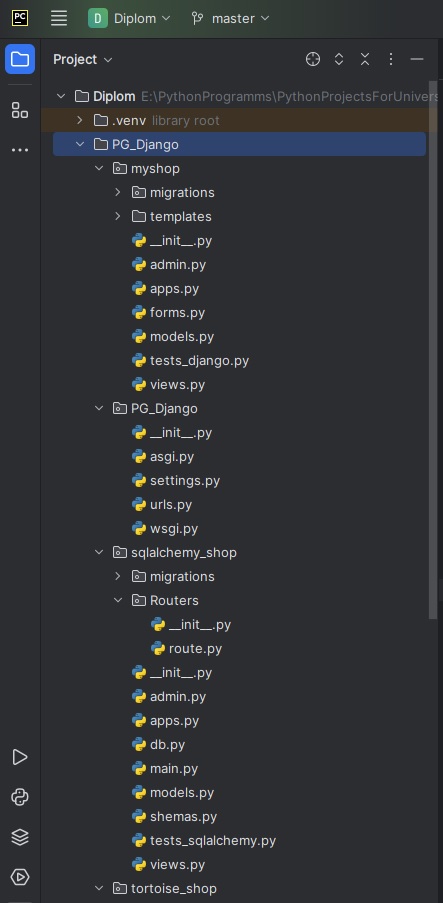
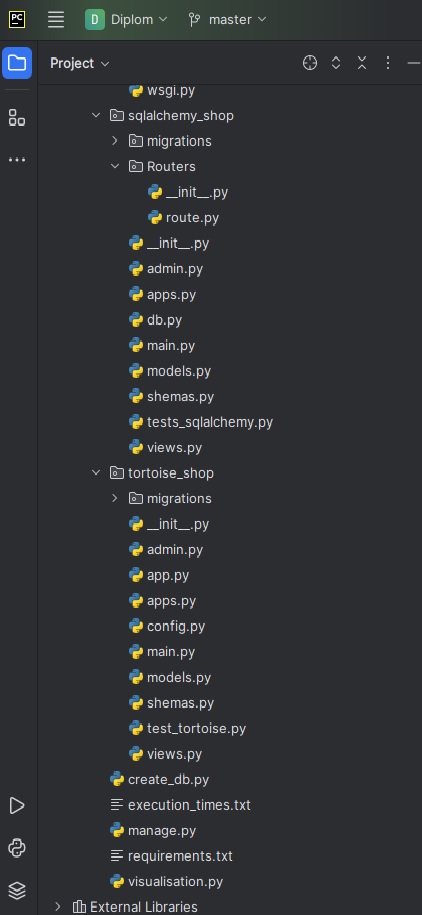
pip install django

pip install sqlalchemy

pip install tortoise-orm

Ниже приведен скриншот структуры проекта:

*Рисунок 1 Структура проекта*

# Создание базы данных посредством Python в PyCharm

Для создания магазина лекарственных препаратов на Python используется база данных. Выбор пал на СУБД PostgreSQL по причине:

1. **Клиент-серверная архитектура**: PostgreSQL использует клиент-серверную архитектуру, что позволяет нескольким клиентам обращаться к базе данных одновременно.
2. **Многопоточность**: PostgreSQL поддерживает многопоточность, что позволяет обрабатывать несколько запросов одновременно.
3. **Широкая поддержка типов данных**: PostgreSQL имеет широкую поддержку типов данных, включая JSON, XML и другие.
4. **Масштабируемость**: PostgreSQL легко масштабируется для обработки больших объемов данных и трафика.
5. **Безопасность:** PostgreSQL имеет встроенную поддержку SSL/TLS и другие функции безопасности.

База данных создается программно. Для этого была установлена библиотека psycopg2:

* pip install psycopg2

Это одна из самых популярных библиотек для работы с базами данных PostgreSQL в Python. Она позволяет разработчикам выполнять SQL-запросы, управлять транзакциями и взаимодействовать с PostgreSQL из своих приложений на Python.

Для работы Tortoise ORM с PostgreSQL необходим модуль asyncpg. Этот модуль является драйвером для работы с PostgreSQL в асинхронном режиме.

* pip install asyncpg

В проекте создан модуль create\_db.py в котором:

* Импортируется библиотека psycopg2 для работы с PostgreSQL.
* Импортируется модуль sql из psycopg2 для безопасного создания SQL-запросов.
* Задаются параметры подключения к серверу PostgreSQL:

HOST: адрес сервера "localhost".

PORT: порт сервера (по умолчанию 5432).

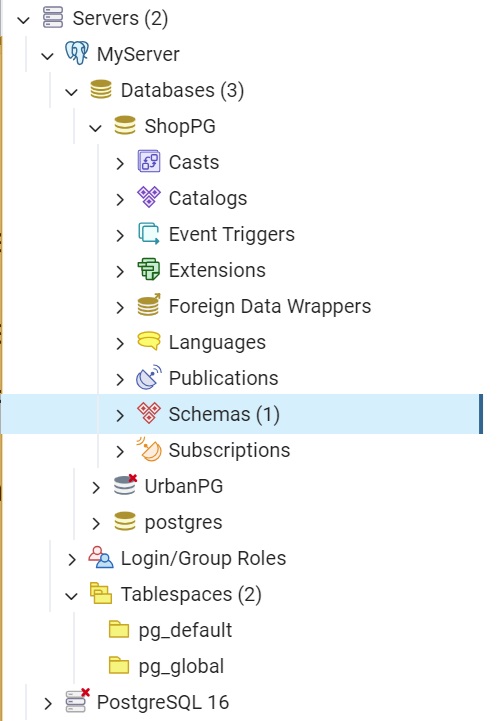
USER: имя пользователя для подключения "postgres".

PASSWORD: пароль пользователя "Zaq12wsX".

* Используется метод connect для установления соединения с сервером PostgreSQL с указанными параметрами. Если соединение успешно, переменная conn содержит объект соединения.
* Включается автоматическое подтверждение транзакций. Это означает, что каждая выполненная SQL-команда будет автоматически зафиксирована в базе данных.
* Создается курсор, который используется для выполнения SQL-запросов. Курсор позволяет взаимодействовать с базой данных, отправлять запросы и получать результаты.
* Задается имя базы данных, которую нужно создать.
* Создается SQL-запрос для создания базы данных с использованием sql.SQL и sql.Identifier. Это предотвращает SQL-инъекции, правильно экранируя идентификаторы (имя базы данных и имя пользователя).
* Запрос выполняется с помощью метода cursor.execute().
* Обрабатываются возможные ошибки с помощью try-except.
* В блоке finally закрывается курсор и соединение с базой данных, чтобы освободить ресурсы.

После запуска на выполнение модуля создается база данных PostgreSQL с именем "ShopPG".

*Рисунок2 База данных*



# Описание приложения myshop (Django)

Приложение Django состоит из нескольких ключевых компонентов, включая модели данных, административный интерфейс и тесты.

* **Модели (models.py)**

Модуль models.py определяет структуру базы данных приложения. В нем описаны три основные модели: Buyer, Medicine и Purchase.

1. **Buyer:**

* name: Имя покупателя (строка, максимум 100 символов).
* balance: Баланс покупателя (десятичное число, всего 10 цифр, из которых 2 после запятой).
* age: Возраст покупателя (целое число, ограниченное диапазоном от 0 до 100).

1. **Medicine:**

* title: Название лекарства (строка, максимум 200 символов).
* category: Категория лекарства (строка, максимум 100 символов).
* cost: Стоимость лекарства (десятичное число, всего 6 цифр, из которых 2 после запятой).
* description: Описание лекарства (текст без ограничений по длине).
* age\_limited: Флаг, указывающий, есть ли возрастное ограничение (булево поле, по умолчанию False).

1. **Purchase:**

* buyer: Ссылка на покупателя (внешний ключ, связанный с моделью Buyer).
* medicine: Ссылка на лекарство (внешний ключ, связанный с моделью Medicine).
* date: Дата покупки (дата и время, автоматически добавляется при создании записи).
* **Административный интерфейс (admin.py)**

Модуль admin.py регистрирует модели для использования в административном интерфейсе Django. Это позволяет нам управлять данными через веб-интерфейс.

* Medicine: Регистрация модели Medicine.
* Buyer: Регистрация модели Buyer.
* Purchase: Регистрация модели Purchase.
* **Тестирование (tests\_django.py)**

Модуль tests\_django.py содержит тестовый случай MyShopTests, который проверяет производительность операций CRUD (Create, Read, Update, Delete) с использованием моделей Buyer, Medicine и Purchase.

1. **Создание (Create):**

* Приложение создает 1000 записей для каждого типа объекта: Buyer, Medicine и Purchase.
* Для каждого покупателя, лекарства и покупки используются фиксированные значения (например, имя "Иван Иванов", баланс 1234, возраст 33, название "Лекарство" и т.д.).
* Проверяется, что количество созданных записей соответствует ожидаемому (1000 для каждой модели).

1. **Чтение (Read):**

* Загружаются все записи для каждого типа объекта, чтобы убедиться, что они корректно сохранены.

1. **Обновление (Update):**

* Обновляется баланс всех покупателей на новое значение (2000).

1. **Удаление (Delete):**

* Все записи для каждого типа объекта удаляются из базы данных.

Другой метод тестирует более сложные операции:

1. **CREATE:** Создание 1000 покупателей с рандомными данными и 1000 лекарств с рандомными данными.
2. **Проверки:** Проверяется количество созданных записей, как и в предыдущем тесте.
3. **SORTING:** Сортировка покупателей по возрасту.
4. **FILTERING:** Фильтрация покупателей с балансом больше 1500.
5. **GROUPING**: Группировка лекарств по категории и подсчет количества в каждой группе.
6. **Измерение времени выполнения:**

* Измеряется время выполнения всех операций.
* Результаты записываются в файл execution\_times.txt.
* **Настройки подключения к базе данных (settings.py)**

В приложении Django настройки подключения к базе данных находятся в файле settings.py.

Разберем каждую из этих настроек:

1. **'ENGINE':**

* Указывает, какой движок базы данных использовать. В данном случае используется PostgreSQL.
* Значение: 'django.db.backends.postgresql'.

1. **'NAME':**

* Имя базы данных, к которой мы подключаемся.
* Значение: 'ShopPG'.

1. **'USER':**

* Имя пользователя для подключения к базе данных.
* Значение: 'postgres'.

1. **'PASSWORD':**

* Пароль для указанного пользователя.
* Значение: 'Zaq12wsX'.

1. **'HOST':**

* Хост, на котором находится база данных. В нашем случае это локальный хост.
* Значение: '127.0.0.1'.

1. **'PORT':**

* Порт, на котором работает сервер базы данных. Для PostgreSQL стандартный порт — 5432.
* Значение: '5432'.
* **Выполнение миграций**

Миграции в Django позволяют синхронизировать изменения в ваших моделях с базой данных.

1. **Создание миграций:**

* После того, как мы определили или изменили модели, нам нужно создать миграции. Это делается с помощью команды makemigrations. Переходим в директорию cd PG\_Django.
* python manage.py makemigrations
* Эта команда создаст файлы миграций в папке migrations внутри нашего приложения.

1. **Применение миграций:**

* После создания миграций их нужно применить к базе данных. Это делается с помощью команды migrate.
* python manage.py migrate
* Эта команда применит все необработанные миграции к базе данных, создавая таблицы, добавляя столбцы, индексы и т.д.
* **Создание суперпользователя**

Создание суперпользователя (superuser) в Django — это важный шаг, так как суперпользователь имеет полные права доступа к административному интерфейсу и может управлять всеми аспектами приложения.

* Для создания суперпользователя используем команду createsuperuser.
* python manage.py createsuperuser
* После выполнения команды createsuperuser вводим следующие данные:

Username: admin

Email address:

Password: Zaq12wsX

Password (again): Zaq12wsX

Теперь у нас есть суперпользователь, который может войти в административный интерфейс.

Мы настроили подключение к базе данных PostgreSQL и выполнили миграции, чтобы синхронизировать структуру базы данных с нашими моделями, а также создали суперпользователя. Теперь наше приложение готово к работе с данными.

* **Вход в административный интерфейс**
* Запускаем сервер разработки Django:
* python manage.py runserver
* Открываем браузер и переходим по адресу http://127.0.0.1:8000/admin/. Увидим страницу входа в административный интерфейс. Вводим имя пользователя и пароль, которые мы указали при создании суперпользователя, и нажиаем кнопку "Войти".
* **Заключение**

Приложение Django является простым, но эффективным примером использования моделей, административного интерфейса и тестирования. Оно демонстрирует, как можно быстро создавать, читать, обновлять и удалять данные, а также измерять производительность этих операций.

# Описание приложения sqlalchemy\_shop (SQLAlchemy)

Данное приложение представляет собой API, созданное с использованием фреймворка FastAPI и ORM SQLAlchemy для управления данными о покупателях, лекарствах и покупках в аптеке.

* **Модуль main:**

Это основной файл, который запускает наше FastAPI приложение. Здесь происходит настройка приложения, включение маршрутов и определение корневого маршрута.

* В этом модуле создается основное приложение FastAPI.
* Импортируются роутеры из модуля route для покупателей (buyer\_router), лекарств (medicine\_router) и покупок (purchase\_router).
* Определяется экземпляр приложения FastAPI с названием "Аптека API", описанием и версией.
* Добавлен маршрут для корневого пути '/', который возвращает приветственное сообщение.
* Маршруты из импортированных роутеров включаются в приложение с помощью метода include\_router.
* **Модуль admin:**

Этот модуль содержит функцию, которая возвращает сессию базы данных.

* Этот модуль содержит функцию get\_db, которая предоставляет сессию базы данных для использования в других модулях.
* Функция использует SessionLocal из модуля db для создания сессии и возвращает ее с помощью генератора.
* **Модуль db:**

Этот модуль содержит функции для взаимодействия с базой данных.

* Создается пользовательский базовый класс Base, от которого наследуются все модели.
* Создается движок базы данных engine и сессионный менеджер SessionLocal.
* Создаются таблицы в базе данных после определения моделей.
* **Модуль models:**

Этот модуль содержит определение моделей для покупателя, лекарства и покупки.

**Модель Buyer**

Класс Buyer представляет собой модель для покупателя. Он содержит следующие поля:

* id: целое число, представляющее уникальный идентификатор покупателя.
* name: строка, представляющая имя покупателя.
* balance: дробное число, представляющее счет покупателя.
* age: целое число, представляющее возраст покупателя.
* slug: строка, представляющая уникальную строку для покупателя.
* purchases: отношение с покупками, представляющее список покупок покупателя.

**Модель Medicine**

Класс Medicine представляет собой модель для лекарства. Он содержит следующие поля:

* id: целое число, представляющее уникальный идентификатор лекарства.
* title: строка, представляющая название лекарства.
* category: строка, представляющая категорию лекарства.
* cost: дробное число, представляющее стоимость лекарства.
* description: строка, представляющая описание лекарства.
* age\_limited: булевое значение, представляющее ограничение по возрасту.
* slug: строка, представляющая уникальную строку для лекарства.
* purchases: отношение с покупками, представляющее список покупок лекарства.

**Модель Purchase**

Класс Purchase представляет собой модель для покупки. Он содержит следующие поля:

* id: целое число, представляющее уникальный идентификатор покупки.
* buyer\_id: целое число, представляющее ID покупателя.
* medicine\_id: целое число, представляющее ID лекарства.
* date: дата и время, представляющие дату покупки.
* buyer: отношение с покупателем, представляющее покупателя, который сделал покупку.
* medicine: отношение с лекарством, представляющее лекарство, которое было куплено.

**Создание таблиц в базе данных**

После определения моделей создается таблицы в базе данных с помощью метода create\_all из класса Base.

Base.metadata.create\_all(engine)

Этот код создаст таблицы sqlalchemy\_buyer, sqlalchemy\_medicine и sqlalchemy\_purchase в базе данных, если они еще не существуют.

* **Модуль shemas.py**

Этот модуль содержит определение схем для создания и обновления покупателя, лекарства и покупки. Схемы используются в модуле route.py для валидации входных данных и обеспечения того, чтобы данные соответствовали ожидаемому формату.

**Схема для создания покупателя**

Класс CreateBuyer представляет собой схему для создания покупателя. Он содержит поля:

* name: строка, представляющая имя покупателя.
* balance: дробное число, представляющее счет покупателя.
* age: целое число, представляющее возраст покупателя.

**Схема для обновления покупателя**

Класс UpdateBuyer представляет собой схему для обновления покупателя. Он содержит такие же поля как и класс CreateBuyer.

**Схема для создания лекарства**

Класс CreateMedicine представляет собой схему для создания лекарства. Он содержит поля:

* title: строка, представляющая название лекарства.
* category: строка, представляющая категорию лекарства.
* cost: дробное число, представляющее стоимость лекарства.
* description: строка, представляющая описание лекарства.
* age\_limited: булевое значение, представляющее ограничение по возрасту.

**Схема для обновления лекарства**

Класс UpdateMedicine представляет собой схему для обновления лекарства. Он содержит такие же поля как и класс CreateMedicine.

**Схема для создания покупки**

Класс CreatePurchase представляет собой схему для создания покупки. Он содержит поля:

* buyer\_id: целое число, представляющее ID покупателя.
* medicine\_id: целое число, представляющее ID лекарства.
* date: дата и время, представляющие дату покупки.

**Схема для обновления покупки**

Класс UpdatePurchase представляет собой схему для обновления покупки. Он содержит такие же поля как и класс CreatePurchase.

* **Модуль route:**

Этот модуль содержит определение маршрутов для покупателей, лекарств и покупок. Каждый роутер имеет префикс и тег для организации маршрутов. Используется зависимость get\_db для получения сессии базы данных. Маршруты используют методы SQLAlchemy для взаимодействия с базой данных.

**Маршруты для покупателей**

* all\_buyers: Получение всех покупателей.
* buyer\_by\_id: Получение покупателя по ID.
* create\_buyer: Создание нового покупателя.
* update\_buyer: Обновление покупателя по ID.
* delete\_buyer: Удаление покупателя по ID.
* get\_buyer\_medicines: Маршрут для получения всех лекарств определенного покупателя

**Маршруты для лекарств**

* all\_medicines: Получение всех лекарств.
* medicine\_by\_id: Получение лекарства по ID.
* create\_medicine: Создание нового лекарства.
* update\_medicine: Обновление лекарства по ID.
* delete\_medicine: Удаление лекарства по ID.

**Маршруты для покупок**

* all\_purchases: Получение всех покупок.
* purchase\_by\_id: Получение покупки по ID.
* create\_purchase: Создание новой покупки.
* update\_purchase: Обновление покупки по ID.
* delete\_purchase: Удаление покупки по ID.
* **Модуль test\_sqlalchemy:**

Этот модуль содержит тесты для проверки функциональности SQLAlchemy. В тесте  выполняются операции CRUD (создание, чтение, обновление, удаление) с большими объемами данных. Время выполнения теста записывается в файл execution\_times.txt.

**setUp и tearDown методы**

* Методы setUp и tearDown используются для очистки данных перед и после каждого теста.

**Тест test\_sqlalchemy**

Этот тест проверяет функциональность SQLAlchemy.

* CREATE - Создание 1000 покупателей, 1000 лекарств и 1000 покупок.
* READ - Получение всех покупателей, лекарств и покупок.
* UPDATE - Обновление баланса всех покупателей.
* DELETE - Удаление всех покупок, лекарств и покупателей.

Другой метод тестирует более сложные операции:

* CREATE: Создание 1000 покупателей с рандомными данными и 1000 лекарств с рандомными данными.
* Проверки: Проверяется количество созданных записей, как и в предыдущем тесте.
* SORTING: Сортировка покупателей по возрасту.
* FILTERING: Фильтрация покупателей с балансом больше 1500.
* GROUPING: Группировка лекарств по категории и подсчет количества в каждой группе.
* Измерение времени выполнения:

o Измеряется время выполнения всех операций.

o Результаты записываются в файл execution\_times.txt.

* **Установка и запуск Swagger**

Чтобы использовать Swagger в нашем приложении, нам необходимо установить библиотеку FastAPI, а затем запустить приложение с помощью Uvicorn.

**Запуск приложения с Swagger**

Чтобы запустить проект и войти в Swagger, необходимо использовать следующую команду:

uvicorn PG\_Django.sqlalchemy\_shop.main:app --reload

В этой команде:

* uvicorn - это сервер ASGI, который используется для запуска приложения FastAPI.
* main - это имя файла, в котором находится приложение FastAPI.
* app - это имя переменной, которая содержит экземпляр приложения FastAPI ( app = FastAPI() ).
* --reload - это флаг, который позволяет серверу автоматически перезагружаться при изменении кода.

После запуска команды, открываем браузер и переходим по адресу http://localhost:8000/docs, чтобы войти в Swagger.

Мы увидим страницу Swagger, на которой можно просматривать API-интерфейсы и тестировать запросы.

**Функции Swagger**

Swagger предоставляет следующие функции:

1. **API-интерфейсы**: Swagger отображает все API-интерфейсы, которые определены в вашем приложении, включая методы запроса, ответы и параметры.
2. **Тестирование запросов**: Swagger позволяет тестировать запросы к API-интерфейсам, чтобы проверить их работу и получить ответы.

**Навигация по Swagger**

В Swagger можно найти API-интерфейсы, используя навигационную панель вверху страницы. Можно выбрать раздел, который интересует нас и просмотреть все API-интерфейсы в этом разделе.

* **Функциональность приложения:**
* Приложение позволяет управлять данными о покупателях, лекарствах и их покупках.
* Пользователи могут получать списки покупателей, лекарств и покупок, а также получать информацию о конкретных записях по идентификатору.
* Возможна создание новых записей, обновление существующих и удаление записей.
* Отношения между моделями реализованы с помощью SQLAlchemy ORM.
* Тестовый модуль обеспечивает проверку производительности и правильности работы с базой данных.

# Описание приложения tortoise\_shop (Tortoise)

Приложение представляет собой API-интерфейс для управления аптекой, реализованное с помощью фреймворка FastAPI и ORM Tortoise. Оно состоит из нескольких модулей, которые обеспечивают функции CRUD (Create, Read, Update, Delete) для покупателей, лекарств и покупок.

* **Модуль config:**

Модуль config является файлом конфигурации, в котором хранятся настройки соединения с базой данных PostgreSQL. В нем определена переменная TORTOISE\_ORM, которая представляет собой словарь с конфигурацией для подключения к базе данных.

Конфигурация включает в себя следующие параметры:

* connections: словарь с параметрами подключения к базе данных. В данном случае, подключение осуществляется к базе данных PostgreSQL на локальном хосте (localhost) с именем пользователя postgres и паролем Zaq12wsX. База данных имеет имя ShopPG.
* apps: словарь с параметрами приложения. В данном случае, приложение имеет имя models и использует модели из модуля models.
* **Модуль main:**

Модуль main.py является основным модулем приложения. В нем определена функция run, которая выполняется при запуске приложения.

В этом модуле выполняются следующие действия:

* run: функция, которая инициализирует Tortoise ORM с использованием конфигурации из модуля config.py и генерирует схемы в базе данных.
* Tortoise.init: функция, которая инициализирует Tortoise ORM с использованием конфигурации.
* Tortoise.generate\_schemas: функция, которая генерирует схемы в базе данных.
* Buyer.all: функция, которая возвращает все покупателей из базы данных.
* Tortoise.close\_connections: функция, которая закрывает соединения с базой данных.
* **Модуль models:**

Модуль models содержит определения моделей данных для приложения. В нем определены три модели: Buyer (покупатель), Medicine (лекарство) и Purchase (покупка).

Каждая модель имеет следующие атрибуты:

* id: уникальный идентификатор объекта.
* name (для Buyer): имя покупателя.
* balance (для Buyer): баланс покупателя.
* age (для Buyer): возраст покупателя.
* title (для Medicine): название лекарства.
* category (для Medicine): категория лекарства.
* cost (для Medicine): стоимость лекарства.
* description (для Medicine): описание лекарства.
* age\_limited (для Medicine): флаг, указывающий, ограничено ли лекарство по возрасту.
* buyer (для Purchase): ссылка на покупателя, совершившего покупку.
* medicine (для Purchase): ссылка на лекарство, приобретенное в покупке.
* date (для Purchase): дата покупки.

Каждая модель также имеет метод \_\_str\_\_, который возвращает строковое представление объекта.

* **Модуль shemas:**

Модуль shemas содержит определения схем данных для приложения. В нем определены схемы для создания и обновления покупателей, лекарств и покупок.

Схемы используются для валидации данных, поступающих от клиента. Каждая схема имеет следующие атрибуты:

* name (для CreateBuyer и UpdateBuyer): имя покупателя.
* balance (для CreateBuyer и UpdateBuyer): баланс покупателя.
* age (для CreateBuyer и UpdateBuyer): возраст покупателя.
* title (для CreateMedicine и UpdateMedicine): название лекарства.
* category (для CreateMedicine и UpdateMedicine): категория лекарства.
* cost (для CreateMedicine и UpdateMedicine): стоимость лекарства.
* description (для CreateMedicine и UpdateMedicine): описание лекарства.
* age\_limited (для CreateMedicine и UpdateMedicine): флаг, указывающий, ограничено ли лекарство по возрасту.
* buyer\_id (для CreatePurchase и UpdatePurchase): идентификатор покупателя, совершившего покупку.
* medicine\_id (для CreatePurchase и UpdatePurchase): идентификатор лекарства, приобретенного в покупке.
* date (для CreatePurchase и UpdatePurchase): дата покупки.
* **Модуль app:**

Модуль app.py содержит определения роутеров для покупателей, лекарств и покупок.

Маршруты разделены на несколько категорий:

Маршруты для покупателей (/buyer):

* POST /buyer/create: создание нового покупателя.
* GET /buyer: получение всех покупателей.
* GET /buyer/{buyer\_id}: получение покупателя по идентификатору.
* PUT /buyer/update/{buyer\_id}: обновление покупателя.
* DELETE /buyer/delete/{buyer\_id}: удаление покупателя.
* GET/{buyer\_id}/medicines: получения лекарств определенного покупателя

Маршруты для лекарств (/medicine):

* POST /medicine/create: создание нового лекарства.
* GET /medicine: получение всех лекарств.
* GET /medicine/{medicine\_id}: получение лекарства по идентификатору.
* PUT /medicine/update/{medicine\_id}: обновление лекарства.
* DELETE /medicine/delete/{medicine\_id}: удаление лекарства.

Маршруты для покупок (/purchase):

* POST /purchase: создание новой покупки.
* GET /purchase: получение всех покупок.
* GET /purchase/{purchase\_id}: получение покупки по идентификатору.
* PUT /purchase/{purchase\_id}: обновление покупки.
* DELETE /purchase/{purchase\_id}: удаление покупки.

Каждый маршрут имеет свою собственную функцию-обработчик, которая выполняет необходимые действия для обработки запроса.

* **Модуль test\_tortoise**

Этот модуль представляет собой набор тестов для проверки работы с базой данных с использованием Tortoise ORM. Импортируются необходимые модули, включая tortoise, модели данных (Buyer, Medicine, Purchase), а также утилиты для тестирования (IsolatedAsyncioTestCase, unittest).

**Класс тестов TortoiseShopTests:**

Этот класс наследуется от IsolatedAsyncioTestCase, что означает, что тесты будут выполняться асинхронно и изолированно.

**В методе asyncSetUp** выполняется инициализация Tortoise ORM, создание схем базы данных и очистка таблиц перед началом тестов.

**В методе asyncTearDown** закрываются соединения с базой данных после завершения тестов.

**Тестовый метод test\_tortoise:**

Здесь выполняются основные тестовые операции с базой данных:

Создание (CREATE): Создаются списки объектов Buyer, Medicine и Purchase, которые затем сохраняются в базе данных с помощью метода bulk\_create. Производится проверка, что количество записей в каждой таблице равно 1000.

Чтение (READ): Извлекаются все записи из таблиц Buyer, Medicine и Purchase.

Обновление (UPDATE): Обновляется поле balance для всех покупателей, демонстрируя процесс изменения данных.

Удаление (DELETE): Удаляются все записи из таблиц базы данных.

Другой метод тестирует более сложные операции:

CREATE: Создание 1000 покупателей с рандомными данными и 1000 лекарств с рандомными данными.

Проверки: Проверяется количество созданных записей, как и в предыдущем тесте.

SORTING: Сортировка покупателей по возрасту.

FILTERING: Фильтрация покупателей с балансом больше 1500.

GROUPING: Группировка лекарств по категории и подсчет количества в каждой группе.

Измерение времени выполнения:

o Измеряется время выполнения всех операций.

o Результаты записываются в файл execution\_times.txt.

* **Установка и запуск Swagger**

Чтобы использовать Swagger в нашем приложении, нам необходимо перейти в директорию cd PG\_Django/tortoise\_shop, а затем запустить приложение с помощью Uvicorn.

**Запуск приложения с Swagger**

Чтобы запустить проект и войти в Swagger, необходимо использовать следующую команду:

uvicorn app:app --reload

В этой команде:

* uvicorn - это сервер ASGI, который используется для запуска приложения FastAPI.
* app - это имя файла, в котором находится приложение FastAPI.
* app - это имя переменной, которая содержит экземпляр приложения FastAPI ( app = FastAPI() ).
* --reload - это флаг, который позволяет серверу автоматически перезагружаться при изменении кода.

После запуска команды, открываем браузер и переходим по адресу http://localhost:8000/docs, чтобы войти в Swagger.

Мы увидим страницу Swagger, на которой можно просматривать API-интерфейсы и тестировать запросы.

**Функции Swagger**

Swagger предоставляет следующие функции:

1. **API-интерфейсы**: Swagger отображает все API-интерфейсы, которые определены в вашем приложении, включая методы запроса, ответы и параметры.
2. **Тестирование запросов**: Swagger позволяет тестировать запросы к API-интерфейсам, чтобы проверить их работу и получить ответы.

**Навигация по Swagger**

В Swagger можно найти API-интерфейсы, используя навигационную панель вверху страницы. Можно выбрать раздел, который интересует нас и просмотреть все API-интерфейсы в этом разделе.

# Подробное описание процесса тестирования и результатов

* **Настройка тестовой среды**

Тесты были написаны для трех различных ORM-фреймворков: Django ORM, SQLAlchemy и Tortoise ORM. Каждый фреймворк имеет свой модуль тестирования:

1. tests\_django.py для Django ORM
2. tests\_sqlalchemy.py для SQLAlchemy
3. tests\_tortoise.py для Tortoise ORM

Все тесты выполняют следующие одинаковые операции, но с использованием различного синтаксиса :

* **CREATE:** Создание 1000 записей для каждой модели (Buyer, Medicine, Purchase).
* **READ:** Чтение всех записей из базы данных.
* **UPDATE:** Обновление баланса для всех покупателей.
* **DELETE:** Удаление всех записей из базы данных.

Также есть расширенные тесты (test\_\*\*\*\_advanced), которые выполняют:

* **Создание** объектов со случайными данными для проведения теститрования
* **Сортировка** покупателей по возрасту.
* **Фильтрация** покупателей с балансом больше 1500.
* **Группировка** лекарств по категории с подсчетом количества в каждой группе.
* **Процесс тестирования**

Тесты запускаются командой:

**PG\_Django> python manage.py test**

При этом выполняются все модули тестов, и результаты выполнения записываются в файл execution\_times.txt.

* **Сбор и визуализация результатов**

После выполнения тестов, модуль visualisation.py считывает данные из файла execution\_times.txt, вычисляет среднее время выполнения для каждого фреймворка и строит столбчатую диаграмму, отображающую сравнение времени выполнения тестов. Гистограмма, построенная с помощью этого модуля, помогает быстро оценить общую производительность каждого ORM-фреймворка как для базовых, так и для усложненных операций.

* **Результаты тестирования**

**Результаты тестирования представлены в следующем формате:**

django: 0.7823026180267334

django\_advanced: 0.9388089179992676

sqlalchemy: 0.9376182556152344

sqlalchemy\_advanced: 0.8295187950134277

tortoise: 5.779613733291626

tortoise\_advanced: 0.1875772476196289

django: 0.8418958187103271

django\_advanced: 0.9535961151123047

sqlalchemy: 0.9225645065307617

sqlalchemy\_advanced: 1.2236669063568115

tortoise: 5.849228620529175

tortoise\_advanced: 0.16270923614501953

django: 0.7745625972747803

django\_advanced: 1.0289251804351807

sqlalchemy: 1.0148372650146484

sqlalchemy\_advanced: 0.8568634986877441

tortoise: 5.892135143280029

tortoise\_advanced: 0.17196416854858398

django: 0.7734427452087402

django\_advanced: 0.9379479885101318

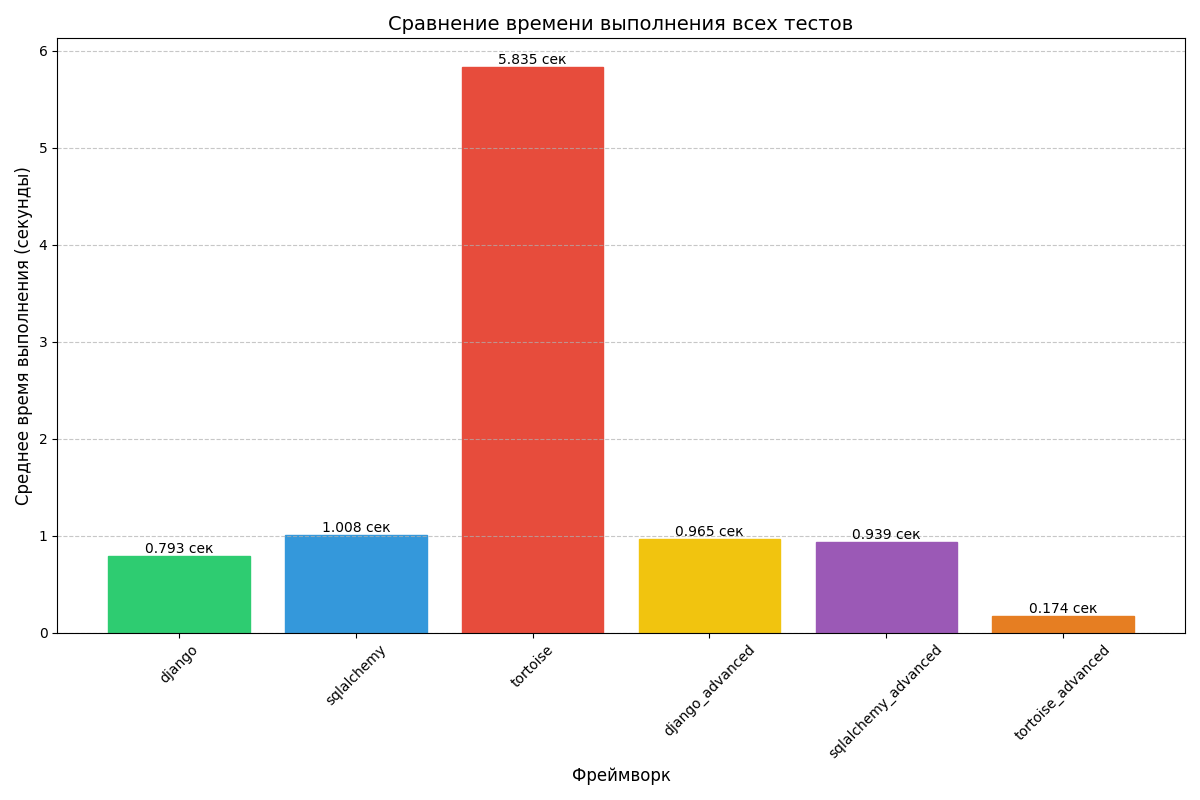
sqlalchemy: 1.1575844287872314

sqlalchemy\_advanced: 0.8452191352844238

tortoise: 5.819587707519531

tortoise\_advanced: 0.1718752384185791

Рисунок3 Результаты тестирования



* **Анализ результатов**

**Django:**

* Обычный тест: среднее время около 0.793 секунд
* Расширенный тест: среднее время около 0.965 секунд

**SQLAlchemy:**

* Обычный тест: среднее время около 1.008 секунд
* Расширенный тест: среднее время около 0.939 секунд

**Tortoise:**

* Обычный тест: среднее время около 5.835 секунд
* Расширенный тест: среднее время около 0.174 секунд
* **Выводы**
* Django ORM является самым быстрым и эффективным фреймворком в обычных тестах, немного опережая SQLAlchemy. Это может быть связано с тем, что Django ORM оптимизирован для работы с базами данных в контексте веб-приложений.
* SQLAlchemy показывает стабильную производительность, близкую к Django, но немного уступает ему в обычных тестах. Это может быть связано с тем, что SQLAlchemy является более универсальным инструментом и предоставляет больше возможностей, что может сказываться на производительности.
* Tortoise показывает самую низкую производительность в обычных тестах. Это может быть связано с тем, что Tortoise ORM является асинхронным фреймворком и может иметь дополнительные накладные расходы при выполнении операций.
* Расширенные тесты для Django и SQLAlchemy выполняются приблизительно за тоже время, что и обычные. Это может быть связано с оптимизациями при работе с большими объемами данных или с особенностями реализации сложных запросов.
* Tortoise показывает значительное улучшение производительности в расширенных тестах. Вероятно, асинхронная природа может давать преимущества в данных сценариях использования.
* **Заключение**

Django и SQLAlchemy показывают схожую и стабильную производительность, в то время как Tortoise демонстрирует неоднозначные результаты, которые могут зависеть от конкретных сценариев использования. Однако выбор фреймворка зависит от конкретных требований проекта и предпочтений разработчика.

* **Для высокопроизводительных CRUD-операций:** Django ORM является отличным выбором благодаря своей сбалансированной производительности и встроенной оптимизации.
* **Для гибких и сложных запросов:** SQLAlchemy предоставляет мощные инструменты для работы с запросами, что может быть полезно в более сложных сценариях.
* **Для асинхронных приложений:** Tortoise ORM подходит для проектов, требующих асинхронных операций и высокой производительности при обработке больших объемов данных.

# Приложение 1. Список необходимых библиотек

aiofiles==24.1.0

aioredis==2.0.1

aiosqlite==0.20.0

annotated-types==0.7.0

anyio==4.6.2.post1

asgiref==3.8.1

async-timeout==4.0.3

asyncpg==0.30.0

atlastk==0.13.3

babel==2.16.0

bcrypt==4.2.0

click==8.1.7

colorama==0.4.6

contourpy==1.3.0

cycler==0.12.1

Django==5.1.2

fastapi==0.115.3

fastapi-admin==1.0.4

fonttools==4.54.1

greenlet==3.1.1

h11==0.14.0

httptools==0.6.4

idna==3.10

iniconfig==2.0.0

iso8601==2.1.0

Jinja2==3.1.4

kiwisolver==1.4.7

lxml==5.3.0

MarkupSafe==3.0.2

matplotlib==3.9.2

numpy==2.1.2

packaging==24.1

pendulum==3.0.0

pillow==11.0.0

pluggy==1.5.0

psycopg2==2.9.10

pydantic==2.9.2

pydantic\_core==2.23.4

pyparsing==3.2.0

pypika-tortoise==0.2.1

pytest==8.3.3

python-dateutil==2.9.0.post0

python-dotenv==1.0.1

python-multipart==0.0.12

python-slugify==8.0.4

pytz==2024.2

PyYAML==6.0.2

six==1.16.0

sniffio==1.3.1

SQLAlchemy==2.0.36

sqlparse==0.5.1

starlette==0.41.0

text-unidecode==1.3

time-machine==2.16.0

tortoise-orm==0.21.7

typing\_extensions==4.12.2

tzdata==2024.2

unittest-xml-reporting==3.2.0

uvicorn==0.32.0

watchfiles==0.24.0

websockets==13.1