Лекция современная бэкенд-разработка (FastAPI, SQLAlchemy, Alembic)

Репозиторий к этой лекции, в котором вы можете увидеть весь код
https://github.com/G00gleKid/backend_lecture

1. Введение

Что такое бэкенд?

Бэкенд — это серверная часть веб-приложения, отвечающая за обработку запросов, работу с базой данных и бизнес-логику.

Клиент (обычно браузер или мобильное приложение) отправляет запросы на сервер, а сервер возвращает данные.

Архитектура клиент-сервер

- Клиент отправляет HTTP-запрос.
- Сервер обрабатывает запрос и взаимодействует с базой данных.
- Сервер отправляет ответ клиенту в виде JSON.

Что такое REST?

REST (Representational State Transfer) — это архитектурный стиль взаимодействия с веб-сервисами, который основан на работе с ресурсами. Основные концепции REST:

- Использование стандартных HTTP-методов (GET, POST, PUT, DELETE и др.).
- Идентификация ресурсов через URL (например, `/users/1`).
- Отсутствие состояния (stateless) — каждый запрос должен содержать всю необходимую информацию.
- Поддержка разных форматов данных, чаще всего JSON.

Когда стоит выбирать REST?

- Когда важно соблюдение стандартов HTTP.
- Если требуется простая и понятная интеграция с клиентами.
- Когда нужно легко кешировать ответы.

---

2. HTTP-методы и работа с объектами

Что такое эндпоинт?

Эндпоинт (endpoint) — это конкретный URL, к которому клиент может отправлять HTTP-запросы для взаимодействия с сервером. Эндпоинты определяют, какие ресурсы доступны и какие действия можно с ними выполнять. Например, `GET /users/{user_id}` — это эндпоинт для получения информации о пользователе.

Основные HTTP-методы

- **GET** — получение данных (без изменения состояния сервера).
- **POST** — создание новых данных.
- **PUT** — полное обновление ресурса.
- **PATCH** — частичное обновление ресурса.
- **DELETE** — удаление ресурса.

Инициализация приложения и пример простого эндпоинта

```
from fastapi import FastAPI

app = FastAPI(title='My App')

@app.get('/hello')
def hello_world():
    return {'message': 'Hello World'}
```

Запустить этот код, можно командой
`fastapi dev`

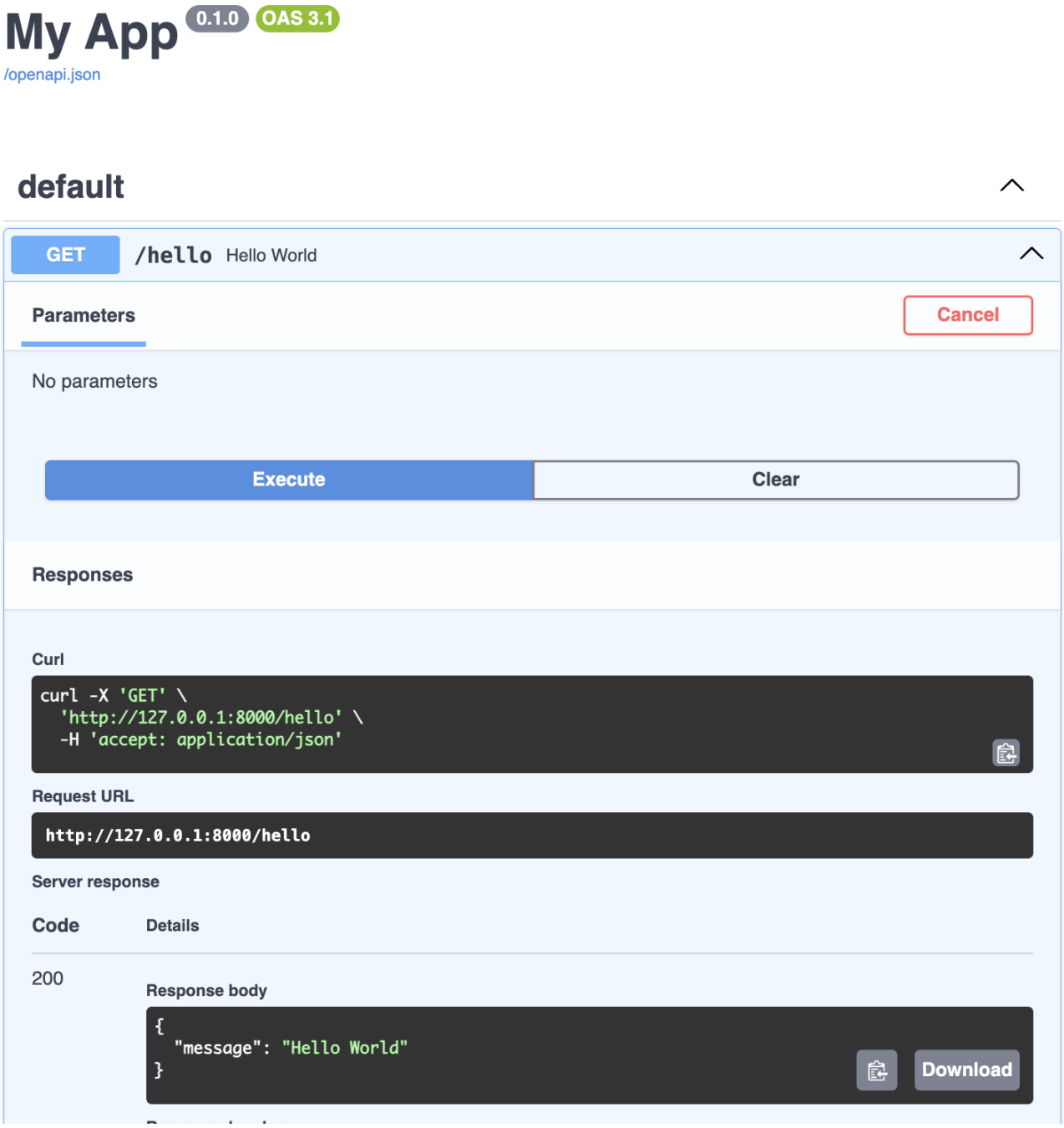
А для того чтобы проверить, как работает этот эндпоинт — надо послать на него http запрос с методом GET по адрессу `http://127.0.0.1:8000/hello`.
В ответ получим:

```
{"message": "Hello World"}
```

Вуаля!

Чтобы сделать это удобно, FastAPI предлагает воспользоваться документацией (swagger), которую фреймворк сам генерирует, когда вы запускаете приложение. Чтобы найти эту документацию, нужно

просто добавить в url `/docs` к адресу вашего приложения и перейти по ней в браузере.
И вот так она будет выглядеть:



Для того чтобы послать запрос — нужно нажать кнопку `Execute` и в response body, мы увидим что нам отвечает сервер.

Отлично, мы научились делать простой эндпоинт и посылать на него запрос. Но если бы каждый энпоинт, возвращал каждый раз одни и те же данные (например "Hello, World!"), в них было бы мало смысла. Потому теперь мы перейдём к тому, как серверу отдавать данные, в зависимости от того что запросил пользователей.

Path-параметры

Один из таких способов — `path` параметры. Представьте, что у нас приложение интернет магазина. Когда пользователь нажимает на товар, он хочет перейти к нему на страницу и узнать больше об этом товаре. Для этого, клиенту (фронтенду), нужно запросить данные по конкретному товару, прислав айди этого товара.

Path-параметры передаются в URL и используются для получения данных о конкретном ресурсе. Например, `GET /items/1` вернет информацию о товаре с ID = 1. Для того чтобы FastAPI понял, что вы хотите объявить path параметр, нужно заключить его название в фигурные скобки, а затем объявить как параметр в функции.

Обратите внимание, что теперь, если надо запросить информацию о фрукте с id=1 url будет выглядеть так:
`http://127.0.0.1:8000/fruits/1`

```
fruits = [{'id': 1, 'name': 'banana'}, {'id': 2, 'name': 'apple'}]

@app.get('/fruits/{fruit_id}')
def get_fruit(fruit_id: int):
    for fruit in fruits:
        if fruit['id'] == fruit_id:
            return fruit

    return None
```

Возвращаемое значение в данном случае будет:

```
{ "id": 1, "name": "banana" }
```

Query-параметры

Query-параметры, схожи по назначению с параметрами пути, только они, передаются в URL после знака `?` , а разделяются знаком &

URL:
`http://127.0.0.1:8000/fruits?fruit_name=apple`

```
@app.get('/fruits')
async def get_fruit_by_name(fruit_name: str):
    for fruit in fruits:
        if fruit['name'] == fruit_name:
            return fruit

    return None
```

Как быть если мы хотим использовать более сложные структуры данных? Не передавать же их всех через параметры. Для этого в http-методах, например таких как POST и PUT, есть тело запроса — `body` . Чтобы удобно передавать данные в теле запроса, сама документация `FastAPI` рекомендует нам использовать библиотеку `Pydantic`

Что такое Pydantic и зачем он нужен?

`Pydantic` — это библиотека для валидации и сериализации данных в Python. Он проверяет получаемые значения на соответствие типам, прежде чем эти данные попадут дальше в ваше приложение. Также `pydantic` используется и внутри приложения, в качестве удобного способа передачи данных.

Если говорить о возвращаемых данных, то эта библиотека позволяет делать сразу несколько полезных вещей:

- **Убирать избыточные данные**. То есть если в объекте `pydantic` окажется больше данных, чем объявлено в этом объекте, `pydantic` их просто не будет использовать и передавать дальше по приложению.
- **Сериализовать данные**. Преобразовывать объекты `pydantic` в JSON объекты, которые затем могут быть переданы по сети.

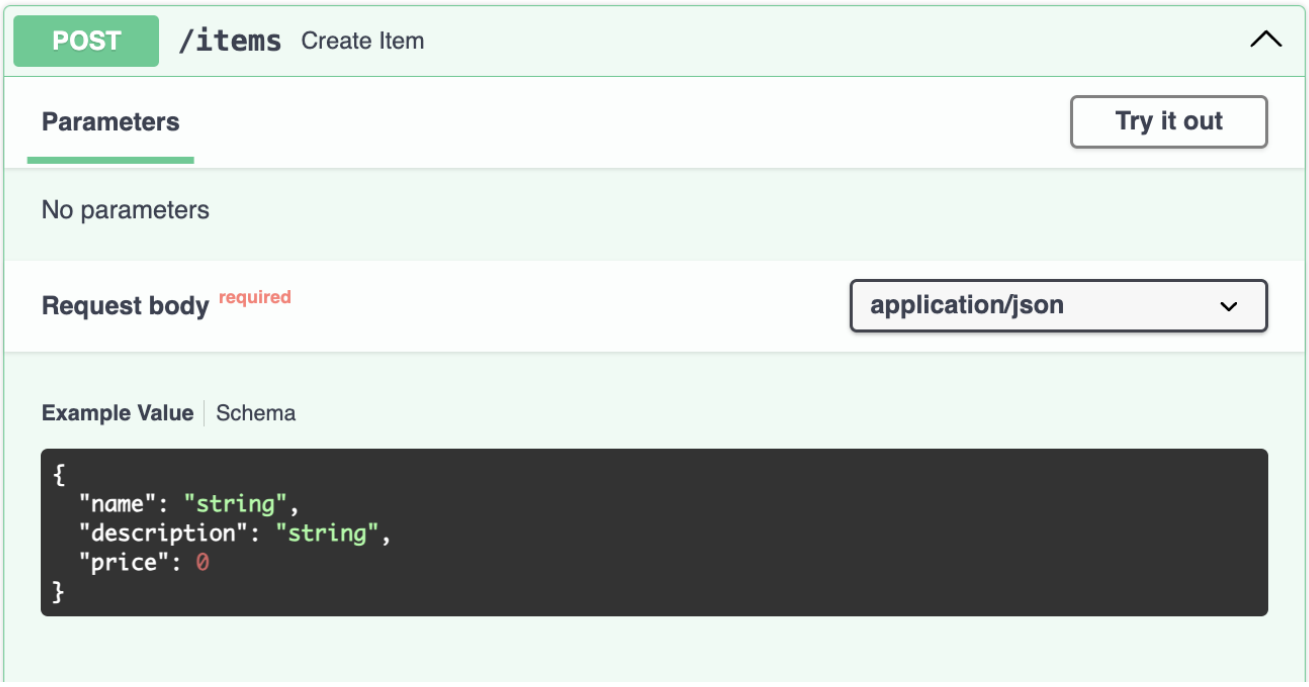
Пример использования Pydantic-схемы:

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str | None = None
    price: float

@app.post("/items")
def create_item(item: Item):
    return item
```

Вот как будет выглядеть документация:



Как видно из скриншота, поскольку мы использовали Pydantic для определения схемы данных, все

необходимые поля отображаются в документации, включая те, которые являются необязательными. Это помогает разработчикам понять, какие именно данные ожидаются от клиента.

Response Model

Когда мы определяем маршруты в `FastAPI`, мы можем использовать Pydantic модели не только для валидации входных данных, но и для описания формата ответа. Это достигается с помощью параметра `response_model`, который гарантирует, что ответ будет соответствовать указанной модели.

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str | None = None
    price: int

class ShortItem(BaseModel):
    name: str
    price: int

item = Item(name='Laptop', description='description', price=1000)

@app.get('/items', response_model=ShortItem)
def read_short_item():
    return item
```

Несмотря на то, что `Item` и `ShortItem` — это две разные модели в `Pydantic`, `Item` содержит все необходимые поля для создания объекта `ShortItem`. В результате, когда мы делаем запрос к `/items`, `FastAPI` автоматически преобразует объект `Item` в формат `ShortItem`, и мы получим следующий JSON-ответ:

```
{
    "name": "Laptop",
    "price": 1000
}
```

Роутеры

Для того чтобы упростить управление маршрутами и логически разделять разные части приложения, в FastAPI есть **роутеры**.

Как вы могли заметить, при описании маршрутов для работы с элементами `items`, мы несколько раз прописывали один и тот же путь `/items`:

```
@app.post('/items')
# код
```

```
@app.get('/items', response_model=ShortItem)
# код
```

С использованием роутеров, этот код выглядел бы так:

```
from fastapi import APIRouter

# Создаем роутер с общим префиксом для всех маршрутов
items_router = APIRouter(prefix='/items')

@items_router.post('')
# код

@items_router.get('', response_model=ShortItem)
# код
```

Чтобы FastAPI приложение увидело этот роутер, нужно включить его в главное приложение:

```
from fastapi import FastAPI

app = FastAPI(title='My App')

app.include_router(items_router, tags=['Items'])
```

---

3. Работа с базой данных

Что такое SQLAlchemy?

`SQLAlchemy` — это библиотека для работы с реляционными базами данных на языке Python. Она предоставляет `ORM` (Object Relational Mapping), который позволяет взаимодействовать с базой данных, используя объекты, а не напрямую SQL-запросы. Это упрощает написание кода и делает его более читаемым и поддерживаемым.

Давайте опишем модели, которые будут символизировать таблицы в базе данных. У нас будет две таблички — юзеры и адреса. Для того чтобы это сделать, нужно импортировать класс `DeclarativeBase`, от которого будут наследоваться пользовательские классы. Этот класс хранит в себе все метаданные о существующих моделях Алхимии.

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
from sqlalchemy import ForeignKey

class Base(DeclarativeBase):
    pass

class User(Base):
    __tablename__ = 'users'
```

```
    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str]
    fullname: Mapped[str]

class Address(Base):
    __tablename__ = 'addresses'

    id: Mapped[int] = mapped_column(primary_key=True)
    email: Mapped[str] = mapped_column(nullable=False)
    user_id: Mapped[int] = mapped_column(ForeignKey('users.id'))
```

`__tablename__ = 'users'` — указывает на то, как таблица будет называться в базе данных.
Далее, каждое название переменной — обозначает название колонки в БД.

Для понимания этого кода, достаточно знать основные термины в работе базы данных. Но некоторые моменты я всё же поясню.

Типы данных которые используют python и алхимия — разные. Например, если у питона есть тип int, то в алхимии это будет свой тип — `Integer`. Также в Алхимии есть типы, которые соответсвуют типам в БД, но которых нет в питоне, например — `BigInteger`. В `Mapped` указывается питоновский тип, на основе которого (если не указать иначе в `mapped_column`) Алхимия поймет, какой **свой** тип назначить этой колонке, а также `python` использует это как аннотацию, чтобы подсказывать разработчику, как можно обрабатывать это значение. Если требуется поставить иной тип данных, которого нет в `python`, первым аргументом в `mapped_column` можно указать тип из `SQLAlchemy` Следом указываются дополнительные атрибуты колонки, такие как `primary_key` (первичный ключ), `nullable` (флаг допустимости NULL), `ForeignKey` (внешний ключ) и т. д.

Отлично! Мы создали наши таблицы, но как перенести их в базу данных? Руками писать SQL в БД чтобы он соответствовал тому что мы описали здесь? — Об этом разработчики тоже позаботились и создали инструмент который занимается всем этим — `Alembic`

Зачем нужен Alembic?

Для начала давайте обсудим вводное понятие — **миграция**.

Миграция в базе данных — это процесс изменения структуры базы данных, который позволяет добавить, изменить или удалить таблицы, столбцы и другие элементы без потери данных. Миграции обычно описываются в виде отдельных файлов или скриптов, которые содержат инструкции по изменению структуры базы данных.

Зачем они нужны? — Представим что вы гениальный системный архитектор и разработали схему базы данных для сервиса мониторинга задач, которая хорошо себя зарекомендовала и вот уже как целый год вы не меняете состояние вашей БД, потому что в ней и так всё есть. Но в один прекрасный день, приходит пользователь, и говорит: «Мне недостаточно статусов задачи "to do", "in progress" и "finished"! Я хочу статус "blocked"!!». И вот вся ваша продуманная архитектура, как и ЧСВ, летит коту под хвост. Что поделать, пользователю ведь не откажешь.

Итого, без использования миграций, вам придётся сначала изменить код моделей Алхимии, затем руками прописывать изменения в базах данных на всех средах разработки, что запросто может привести к ошибкам и несоответствиям.

А если бы вы использовали миграции, то процесс обновления структуры базы данных был бы гораздо проще, безопаснее и удобнее. Вместо ручного внесения изменений во всех средах вам достаточно было бы создать один файл миграции, в котором чётко прописаны изменения — добавление нового статуса в таблицу задач. Затем этот файл можно было бы применить автоматически на всех окружениях (локальном, тестовом, продакшен).

Кроме того, миграции гарантируют сохранность данных, позволяют легко отслеживать то, какие изменения вносились в базу данных, гарантируют безопасность, например если ваш разработчик не очень умело может писать sql, а также позволяют лекго сделать откат, если что-то не приживётся.

`Alembic` — это инструмент для управления миграциями в базе данных. Миграции позволяют отслеживать изменения в структуре базы данных и применять их, не теряя данных. Это особенно полезно, когда приложение развивается и схема базы данных изменяется.

Для того чтобы сделать миграцию, нужно прописать команду
`alembic revision --autogenerate -m "<название миграции>"`

Флаг `--autogenerate`, указывается для того, чтобы alembic сам сгенерировал миграции, на основе состояния вашей базы данных и моделей алхимии. К коду который я привёл выше создаётся следующая миграция:

```
"""create users and addresses tables

Revision ID: ef0ce29c678f
Revises:
Create Date: 2025-02-16 17:57:35.392845

"""
from typing import Sequence, Union

from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision: str = 'ef0ce29c678f'
down_revision: Union[str, None] = None
branch_labels: Union[str, Sequence[str], None] = None
depends_on: Union[str, Sequence[str], None] = None


def upgrade() -> None:
    op.create_table(
        'users',
        sa.Column('id', sa.Integer(), nullable=False),
```

```
        sa.Column('name', sa.String(), nullable=False),
        sa.Column('fullname', sa.String(), nullable=False),
        sa.PrimaryKeyConstraint('id')
    )
    op.create_table(
        'addresses',
        sa.Column('id', sa.Integer(), nullable=False),
        sa.Column('email', sa.String(), nullable=False),
        sa.Column('user_id', sa.Integer(), nullable=True),
        sa.ForeignKeyConstraint(['user_id'], ['users.id'], ),
        sa.PrimaryKeyConstraint('id')
    )


def downgrade() -> None:
    op.drop_table('addresses')
    op.drop_table('users')
```

Сверху файла в комментарии содержится вспомогательная информация, `Revision ID` — это айди этой миграции (миграция и ревизия — однозначные слова). `Revises` — айди предыдущей миграции, так как все миграции идут последовательно. Если кто-то склонирует ваш проект, и захочет накатить актуальное состояние базы данных, то каждая миграция применится последовательно, от самых старых к самым новым. Тут, в `Revises` пусто, потому что это первая миграция. Та же самая информация указана ниже кодом до функции.

В функциях `upgrade` и `downgrade` указываются инструкции, которые нужно сделать при `upgrade` (применении миграции к БД) и `downgrade` (откате миграции) соответсвенно.

Для того чтобы применить миграции к БД до самой последней, нужно выполнить команду — `alembic upgrade head`, для отката последней миграции — `alembic downgrade -1` (ну или `alembic downgrade base`, если вы хотите откатить все миграции)

Основы работы с SQLAlchemy

Как уже упоминалось ранее, SQLAlchemy — это ORM фреймворк, который позволяет разработчикам работать с сущностями БД как с объектами и классами Python, что упрощает внедрение БД в ваш код.

Давайте теперь создадим простые функции, которые покажут как может работать алхимия

```
from sqlalchemy import insert, select

async def get_users(db_session: AsyncSession):
    return (await db_session.execute(select(User))).scalars().all()


async def get_user(db_session: AsyncSession, user_id: int):
    return (await db_session.scalar(select(User).where(User.id == user_id)))


async def create_user(db_session: AsyncSession, name: str, fullname: str):
```

```
    await db_session.execute(insert(User).values(name=name, fullname=fullname))

    await db_session.commit()
```

* что такое db_session (сессия) поговорим позднее; на данный момент, представьте, что это просто соединение с базой данных

Для того чтобы выполнить sql, внутрь `db_session.execute` нужно поместить выражения алхимии. В первой функции выражение `select(User)` указывает, что мы хотим получить все строки из таблицы users. Метод `.scalars()` преобразует их в объекты Python, ну а `.all()` заворачивает всё это в массив. На выходе получается массив из объектов `User`.

В функции `get_user`, добавляется условие

```
.where(User.id == user_id)
```

которое соответствует WHERE из SQL, и ставит условие, по которому выбираться будут только те строчки, у которых id равен заданному.

для сокращения, можно вместо вызова `db_session.execute()`, и последующего оборачивания в `scalars()`, можно сразу вызывать `db_session.scalar()` или `db_session.scalars()` в зависимости от того, ожидается получить одно значение или несколько.

Ну а в функции `create_user` показана вставка новой строки в таблицу. По аналогии с синтаксисом SQL. Только в отличии от предыдущих запросов, так как происходит изменение таблицы базы данных, в данном случае добавление строчки, нужно обязательно сделать `await db_session.commit()` чтобы транзакция применилась к БД.

Как вы могли заметить, код выше, очень похож на синтаксис языка SQL и такой подход в SQLAlchemy называется **`Core`**. Он более гибкий, полезен для выполнения более сложных операций, требующих тонкого контроля, немного быстрее по производительности, но в удобности уступает следующему подходу — **`ORM`**.

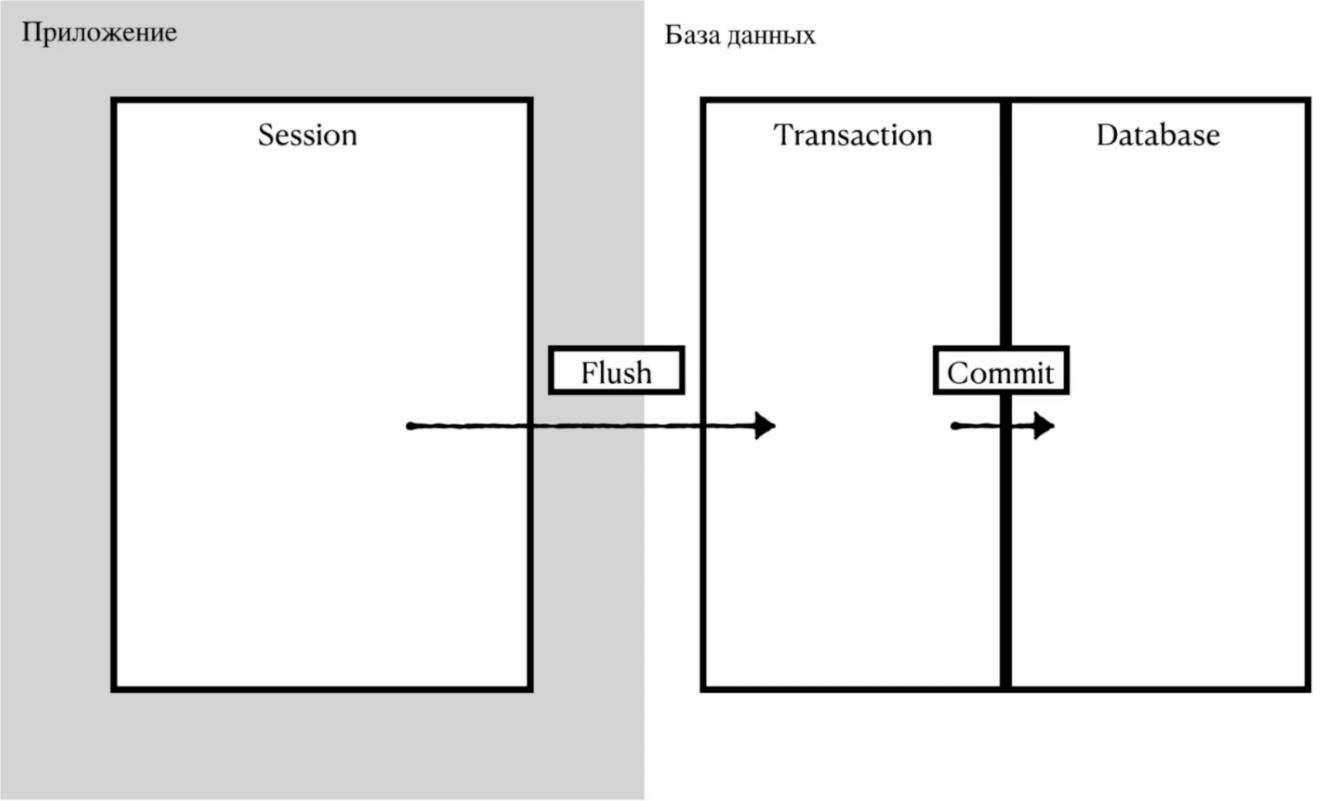
Как уже упоминалось ранее, ORM, позволяет работать с сущностями базы данных, как с объектами и классами языка программирования. Но для понимания этой концепции, уже не обойтись бе понимания что такое сессия в SQLAlchemy

Сессия (Session) в SQLAlchemy — это объект, который управляет соединением с базой данных и выполняет запросы к ней. Она играет роль “промежуточного слоя” между кодом приложения и базой данных, позволяя эффективно работать с транзакциями и изменениями данных.

При выполнении операций в базе данных важно понимать разницу между сессией и транзакцией:

- **Транзакция** — это логическая единица работы с базой данных, которая может быть зафиксирована (commit()) или отменена (rollback()).
- **Сессия** — это объект в SQLAlchemy, который управляет состоянием объектов и запросами к базе. Она может содержать одну или несколько транзакций.

SQLAlchemy автоматически объединяет несколько операций в рамках одной транзакции, что снижает нагрузку на базу данных. Например, если добавить несколько объектов в сессию перед вызовом commit(), то они будут записаны в базу одной транзакцией.



Теперь давайте рассмотрим, как использовать ORM подход на примере с таблицей адресов:

```
async def create_address(db_session: AsyncSession, email: str, user_id: int):
    new_address = Address(
        email=email,
        user_id=user_id
    )

    db_session.add(new_address)

    await db_session.commit()
    await db_session.refresh(new_address)

    return new_address
```

Для начала, вместо того чтобы писать sql-like код, мы создаём объект `new_address` класса `Address`. В Алхимии, такой объект помечается как **`transient`** (временный). Таким состоянием помечаются те экземпляры, которые не находятся в сесси и не сохранены в базе данных (не имеет идентификатора базы данных). Единственная его связь с `ORM` заключается в том, что его класс помечен как модель алхимии.

Следующим шагом, вызовом метода `add()`, объект добавляется в сессию Алхимии, и переходит в состояние **`pending`** (ожидающий), то есть Алхимия помечает этот объект, как тот, который со

следующим сбрасыванием изменений в транзакцию (`flush`) будет добавлен в базу данных и перейдёт в состояние **`persistent`** .

После `commit()` (применяя который, также происходит `flush`), все изменения из транзакции попадают в базу данных, и объект `new_address` , переходит в состояние **`persistent`** (постоянный), которое означает, что этот объект имеет идентификатор базы данных (т.е. первичный ключ), а также в настоящее время связан с сессией.

Методом `refresh()` , подтягиваются актуальные данные об этом объекте из базы данных. В нашем случае, мы получили id объекта, который генерируется на стороне базы данных.

Если вы ещё не запутались в состояниях, поговорим о других двух, которые мы не упомянули :)

`deleted` (удалённый)— противоположность **`pending`** . Таким состоянием помечаются объекты которые были удалены в рамках `flush` , но транзакция ещё не совершилось, и объект ещё находится в базе данных. Когда произойдёт `commit` и транзакция завершится, этот объект перейдёт в состояние

`detached` (отсоединённый) — экземпляр, который соответствует или ранее соответствовал записи в базе данных, но в настоящее время не находится ни в одном сеансе. Отсоединенный объект будет содержать маркер идентификации базы данных, однако, поскольку он не связан с сеансом, неизвестно, существует ли эта идентификация в базе данных. Например если в коде выше, после коммита закрыть сессию, то объект перейдёт в состояние **`detached`**

Теперь со всеми нашими знаниями давайте посмотрим на ещё одну прелесть использования ORM стиля.

Добавим в модель `User` отношение с адресами, указывается связь с помощью `relationship` .Но в данном случае addresses — это не колонка в таблице базы данных, а атрибут класса SQLAlchemy ORM, который используется для установления связи между моделями.

```
from sqlalchemy.orm import relationship

class User(Base):
    __tablename__ = 'users'

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str]
    fullname: Mapped[str]

    addresses: Mapped[list['Address']] = relationship('Address')


class Address(Base):
    __tablename__ = 'addresses'

    id: Mapped[int] = mapped_column(primary_key=True)
    email: Mapped[str] = mapped_column(nullable=False)
    user_id: Mapped[int] = mapped_column(ForeignKey('users.id'))
```

`Mapped[list['Address']]` — указывает python, что обратившись у модели юзера к полю addresses, тип данных будет массив из объектов `Address`. А так как у адресов, есть поле user_id, алхимия сама найдет адреса которые связаны юзером.

Давайте опишем эндпоинт FastAPI и вызовем его. Представим, что мы хотим получить пользователя и все адреса привязанные к пользователю. Используя Core Алхимию, нам пришлось бы делать select пользователя, а затем join-ить к нему строчки из таблицы адресов, где user_id равен айди пользователя. В ORM это делается в две строчки

```
async def get_user_addresses(db_session: AsyncSession, user_id: int):
    user = await db_session.get(User, user_id)

    return await user.awaitable_attrs.addresses
```

* awaitable_attrs — используется для того чтобы такая возможность работала при асинхронном подключении

Сначала с помощью `get()`, по первичному ключу получаем пользователя, а затем просто обращаемся к его атрибуту! Вот и всё, мы получили все адреса пользователя, теперь только осталось вызвать эту функцию из эндпоинта

```
@users_router.get('/{user_id}/addresses')
async def get_user_addresses_api(
    user_id: int,
    db_session: AsyncSession = Depends(get_session)
):
    return await get_user_addresses(db_session, user_id)
```

Допустим мы хотим получить все адреса пользователя с айди=1. Отправляем GET запрос на url `http://127.0.0.1:8000/users/1/addresses`. Результат:

```
[
  {
    "email": "example@gmail.com",
    "user_id": 1,
    "id": 1
  },
  {
    "email": "petya2020@gmail.com",
    "user_id": 1,
    "id": 2
  }
]
```