Оглавление

[**изучить Fast-API** 1](#_Toc138427372)

[**Команда "alembic init alembic"** 2](#_Toc138427373)

[**Вот пример использования Alembic** 2](#_Toc138427374)

[**В контексте Alembic, "окружение** 4](#_Toc138427375)

[**Десериализация данных** 6](#_Toc138427376)

[**REST API (Representational State Transfer Application Programming Interface)** 6](#_Toc138427377)

[**"Репрезентативное состояние"** 7](#_Toc138427378)

[**Давайте разберем каждую строку в коде: # alembic/env.py** 7](#_Toc138427379)

# **изучить Fast-API**

FastAPI - это современный и быстрый фреймворк для создания веб-приложений на основе Python. Он позволяет легко и эффективно разрабатывать высокопроизводительные API (интерфейсы программного обеспечения) с помощью использования современных подходов и принципов, таких как типизация данных и асинхронное программирование.

Вот некоторые ключевые особенности и преимущества FastAPI:

Быстрота и производительность: FastAPI основан на ASGI-сервере (Asynchronous Server Gateway Interface), что позволяет обрабатывать запросы асинхронно и достигать высокой производительности. Он использует систему типов Python для создания сильно типизированных API, что улучшает производительность и помогает предотвратить ошибки.

Простота использования: FastAPI обладает простым и интуитивно понятным синтаксисом, основанным на аннотациях типов Python. Это позволяет быстро определить структуру данных запросов и ответов API. FastAPI также автоматически генерирует интерактивную документацию API на основе этих аннотаций типов, что делает процесс разработки и отладки более удобным.

Поддержка асинхронности: FastAPI полностью совместим с асинхронным программированием и позволяет создавать эффективные асинхронные приложения. Вы можете использовать асинхронные функции и корутины для обработки запросов, вызова внешних API и выполнения других асинхронных операций.

Валидация данных и автоматическая документация: FastAPI предоставляет встроенные инструменты для автоматической валидации входных данных запросов и генерации документации API. Он использует аннотации типов Python для определения схемы данных запросов и ответов, а также проводит автоматическую проверку типов данных и валидацию входных параметров.

Поддержка авторизации и аутентификации: FastAPI предлагает возможности для реализации системы авторизации и аутентификации в ваших API. Вы можете использовать различные методы аутентификации, такие как JWT (JSON Web Tokens), OAuth и другие, для защиты ваших конечных точек API.

FastAPI является мощным фреймворком для разработки веб-приложений и API на Python. Он позволяет создавать высокопроизводительные и надежные API с минимальным количеством кода. Если вы хотите изучить FastAPI, рекомендуется ознакомиться с его официальной документацией, которая предлагает подробные примеры и руководства по разработке приложений с использованием этого фреймворка.

**Команда "alembic init alembic"** создает папку "alembic" в корневой директории проекта и инициализирует ее как директорию миграций для Alembic - это инструмент для управления миграциями базы данных в Python.

Миграции базы данных - это процесс изменения схемы базы данных, который может включать в себя создание, изменение или удаление таблиц, столбцов, ограничений и других элементов базы данных.

Alembic позволяет разработчикам разрабатывать миграции базы данных в Python, а затем применять их к базе данных, что упрощает процесс управления схемой базы данных и обеспечивает более безопасное и надежное внесение изменений в базу данных.

**Вот пример использования Alembic** для создания миграции базы данных в приложении Flask, использующем SQLAlchemy для работы с базой данных:

1. Установите Alembic и его зависимости:

pip install alembic sqlalchemy

1. Создайте файл alembic.ini в корневой папке вашего проекта:

# alembic.ini

[alembic]

script\_location = alembic

1. Создайте папку alembic в корневой папке вашего проекта и выполните команду:

alembic init alembic

1. Отредактируйте файл alembic/env.py, чтобы в нем была настройка подключения к вашей базе данных:

# alembic/env.py

from sqlalchemy import engine\_from\_config

from sqlalchemy import pool

from alembic import context

from myapp import db # здесь импортируйте свой инстанс SQLAlchemy

# Эта функция вызывается для получения URL-адреса базы данных из конфигурации

config = context.config

config.set\_main\_option('sqlalchemy.url', str(db.engine.url))

# Эта функция вызывается для получения объекта метаданных SQLAlchemy

# для использования в создании миграций

target\_metadata = db.metadata

1. Создайте миграцию, выполнив команду:

alembic revision --autogenerate -m "create users table"

1. Отредактируйте созданную миграцию в папке alembic/versions, чтобы добавить необходимые изменения в базу данных:

# alembic/versions/xxxxxxxxxxxx\_create\_users\_table.py

from alembic import op

import sqlalchemy as sa

# Эта функция вызывается при применении миграции

def upgrade():

op.create\_table('users',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('name', sa.String(length=255), nullable=True),

sa.PrimaryKeyConstraint('id')

)

# Эта функция вызывается при откате миграции

def downgrade():

op.drop\_table('users')

1. Примените миграцию к базе данных, выполнив команду:

alembic upgrade head

1. Проверьте, что таблица была успешно создана в базе данных.

Это пример простой миграции базы данных, создающей таблицу пользователей. Alembic позволяет делать более сложные изменения в базе данных, такие как добавление новых столбцов, изменение типов данных и другие.

Конечно, вот пошаговое руководство по созданию экземпляра SQLAlchemy и его подключению к приложению Flask:

1. Установите необходимые зависимости: Flask и Flask-SQLAlchemy.

pip install Flask Flask-SQLAlchemy

1. Импортируйте необходимые модули:

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

1. Создайте экземпляр приложения Flask:
2. app = Flask(\_\_name\_\_)
3. Установите строку подключения к базе данных:

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///myapp.db'

Здесь мы используем SQLite в качестве базы данных, но вы можете использовать любую другую базу данных, поддерживаемую SQLAlchemy.

1. Создайте экземпляр SQLAlchemy:

db = SQLAlchemy(app)

1. Определите модели данных, которые будут использоваться в приложении, например:

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(50))

email = db.Column(db.String(50))

1. Выполните миграции базы данных, чтобы создать необходимые таблицы:

flask db init

flask db migrate

flask db upgrade

Здесь мы используем Flask-Migrate для выполнения миграций базы данных. Если вы не установили Flask-Migrate, выполните следующую команду:

pip install Flask-Migrate

1. Используйте экземпляр SQLAlchemy для выполнения операций базы данных в вашем приложении, например:

@app.route('/users')

def list\_users():

users = User.query.all()

return render\_template('users.html', users=users)

Здесь мы используем метод query.all() для получения всех пользователей из таблицы User и передаем их в шаблон users.html.

Это базовое руководство по созданию экземпляра SQLAlchemy и его подключению к приложению Flask. Вы можете настроить его под свои потребности, добавив дополнительные модели данных и операции базы данных.

**В контексте Alembic, "окружение**" - это способ определения настроек и параметров для миграций базы данных. Окружение Alembic обеспечивает независимость миграций от конкретной базы данных и позволяет использовать один и тот же код миграций для разных баз данных.

Окружение Alembic обычно настраивается в файле alembic.ini, который содержит параметры подключения к базе данных, пути к папкам с миграциями и другие настройки. Например, в файле alembic.ini можно определить следующие параметры:

# alembic.ini

[alembic]

script\_location = alembic

sqlalchemy.url = postgresql://user:password@localhost/mydatabase

def upgrade() -> None:

op.create\_table(

'users',

sa.Column('id', sa.Integer, primary\_key=True),

sa.Column('name', sa.String(50)),

sa.Column('email', sa.String(100), unique=True)

)

def downgrade() -> None:

op.drop\_table('users')

project\_root/

|- alembic.ini

|- myapp.s3db

|- migrations/

|- versions/

|- <migration\_files>.py

**Десериализация данных**

- это процесс преобразования сериализованных данных (например, JSON или бинарных данных) в объекты, которые можно использовать в программном коде.

В зависимости от языка программирования, который вы используете, процесс десериализации может различаться. В Python, например, для десериализации JSON объекта в объект Python используется модуль json. Пример:

import json

# JSON объект

json\_data = '{"name": "John", "age": 30, "city": "New York"}'

# десериализация JSON объекта в объект Python

python\_obj = json.loads(json\_data)

# вывод значений объекта Python

print(python\_obj["name"])

print(python\_obj["age"])

print(python\_obj["city"])

В этом примере мы импортировали модуль json, затем использовали функцию loads для десериализации JSON объекта в объект Python. После этого мы можем обращаться к значениям в объекте Python, используя ключи, так же, как мы это делали в JSON объекте.

**REST API (Representational State Transfer Application Programming Interface)** - это набор принципов и правил для разработки веб-сервисов, которые позволяют клиентам взаимодействовать с сервером через определенные конечные точки (эндпоинты) посредством HTTP-протокола. REST API обеспечивает доступ к ресурсам (например, базе данных) и операции над ними.

Для реализации сервиса с использованием REST API обычно требуется определить следующие эндпоинты:

GET /resource - Получение списка ресурсов. Этот эндпоинт позволяет получить все элементы определенного ресурса.

GET /resource/{id} - Получение конкретного ресурса по его идентификатору. Этот эндпоинт позволяет получить информацию о конкретном элементе ресурса.

POST /resource - Создание нового ресурса. Этот эндпоинт позволяет создать новый элемент ресурса, используя переданные данные.

PUT /resource/{id} - Обновление ресурса. Этот эндпоинт позволяет обновить информацию о конкретном элементе ресурса, используя переданные данные.

DELETE /resource/{id} - Удаление ресурса. Этот эндпоинт позволяет удалить конкретный элемент ресурса по его идентификатору.

GET /resource/{id}/subresource - Получение связанных ресурсов. Этот эндпоинт позволяет получить связанные элементы другого ресурса.

POST /resource/{id}/action - Выполнение действия над ресурсом. Этот эндпоинт позволяет выполнить определенное действие или операцию над конкретным элементом ресурса.

Кроме того, REST API может включать дополнительные эндпоинты и ресурсы в зависимости от специфики сервиса и требований приложения. Например, если ваш сервис предоставляет аутентификацию и авторизацию, могут быть эндпоинты для регистрации, входа и управления пользователями.

Важно помнить, что REST API должен быть разработан согласно принципам REST, включая использование соответствующих HTTP методов (GET, POST, PUT, DELETE) для выполнения операций над ресурсами, правильную обработку кодов состояния HTTP, верное использование маршрутов (URL) и правильную структуру данных в формате JSON или XML для запросов и ответов.

Общая архитектура REST API должна следовать принципам масштабируемости, удобства использования, безопасности и совместимости.

REST (Representational State Transfer) - это архитектурный стиль, используемый для проектирования и создания веб-сервисов. Он определяет набор ограничений и рекомендаций для создания распределенных систем, которые могут быть легко масштабированы, управляемы и расширяемы.

Основные принципы REST:

1. Клиент-серверная архитектура: клиенты и серверы должны быть независимы друг от друга, чтобы их можно было разрабатывать и масштабировать независимо друг от друга.
2. Без состояния (Stateless): каждый запрос должен содержать всю необходимую информацию для его выполнения, без сохранения состояния на сервере.
3. Кэширование (Cacheable): сервер должен явно указывать, можно ли кэшировать ответ на запрос.
4. Единообразный интерфейс (Uniform Interface): интерфейс должен быть единообразным для всех клиентов, что упрощает его использование и понимание.
5. Слои (Layered System): клиенты не должны знать о слоях, которые находятся между ними и сервером, что позволяет улучшить масштабируемость системы.
6. Код по требованию (Code on Demand) (необязательный): сервер может отправлять код клиентам для выполнения на их стороне, что позволяет расширять функциональность клиента.

REST использует стандартные HTTP-методы, такие как GET, POST, PUT, DELETE, для выполнения операций с ресурсами. Каждый ресурс имеет свой уникальный идентификатор (URI), который используется для доступа к нему. Ресурсы могут быть представлены в различных форматах, таких как XML, JSON или HTML.

REST является широко используемым и популярным архитектурным стилем для создания веб-сервисов и API.

**"Репрезентативное состояние"** - это понятие, которое используется в контексте управления состоянием веб-приложений. Оно описывает состояние приложения, которое наиболее точно отражает текущее состояние приложения и может быть использовано для восстановления состояния приложения в случае его сбоя или перезагрузки.

Веб-приложения могут иметь множество состояний, которые могут изменяться в зависимости от действий пользователя и других факторов. Репрезентативное состояние - это состояние, которое содержит все необходимые данные, чтобы восстановить состояние приложения в любой момент времени. Это может включать в себя информацию о текущем пользователе, выбранных настройках, текущих задачах и т.д.

Репрезентативное состояние должно быть хранено в безопасном месте и обновляться при каждом изменении состояния приложения. Это обеспечит возможность восстановления состояния приложения в случае его сбоя или перезагрузки.

В целом, репрезентативное состояние является важным аспектом управления состоянием веб-приложений и должно быть учитывано при разработке и тестировании приложений.

# **Давайте разберем каждую строку в коде: # alembic/env.py**

1. **from sqlalchemy import engine\_from\_config, pool**: Импортируются функции **engine\_from\_config** и **pool** из пакета **sqlalchemy**. Эти функции используются для создания и настройки соединения с базой данных.
2. **import sys**: Импортируется модуль **sys**, который предоставляет доступ к некоторым переменным и функциям, связанным с интерпретатором Python.
3. **from sqlalchemy import create\_engine**: Импортируется функция **create\_engine** из пакета **sqlalchemy**. Она используется для создания экземпляра движка SQLAlchemy, который представляет соединение с базой данных.
4. **import logging**: Импортируется модуль **logging**, который предоставляет функциональность для ведения журнала (логирования).
5. **from alembic import context**: Импортируется модуль **context** из пакета **alembic**, который предоставляет контекст выполнения миграций.
6. **from practi import db**: Импортируется объект **db** из модуля **practi**. Этот объект представляет инстанс SQLAlchemy и содержит информацию о базе данных и метаданных моделей.
7. **config = context.config**: Создается переменная **config**, которая получает конфигурацию Alembic.
8. **config.set\_main\_option('sqlalchemy.url', str(db.engine.url))**: Устанавливается основная опция конфигурации **sqlalchemy.url**, которая содержит URL-адрес базы данных. В данном случае, URL-адрес получается из инстанса SQLAlchemy (**db.engine.url**).
9. **target\_metadata = db.metadata**: Создается переменная **target\_metadata**, которая получает объект метаданных SQLAlchemy, связанный с инстансом SQLAlchemy (**db.metadata**).
10. **db\_url = config.get\_main\_option('sqlalchemy.url')**: Получается значение основной опции конфигурации **sqlalchemy.url** и сохраняется в переменную **db\_url**. Это значение представляет URL-адрес базы данных.
11. **engine = create\_engine(db\_url)**: Создается экземпляр SQLAlchemy движка (**engine**) с использованием URL-адреса базы данных (**db\_url**).
12. **db.bind = engine**: Устанавливается связь между инстансом SQLAlchemy (**db**) и созданным движком (**engine**). Это позволяет SQLAlchemy использовать этот движок для работы с базой данных.
13. **logger = logging.getLogger('alembicLog')**: Создается экземпляр логгера с именем **'alembicLog'**.
14. **logger.setLevel(logging.DEBUG)**: Устанавливается уровень логирования логгера на **DEBUG**.
15. **file\_handler = logging.FileHandler('alembicLog.log')**: Создается обработчик для записи логов в файл с именем **'alembicLog.log'**.
16. **file\_handler.setLevel(logging.DEBUG)**: Устанавливается уровень логирования обработчика на **DEBUG**.
17. **formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')**: Создается форматтер для указания формата записей лога.
18. **file\_handler.setFormatter(formatter)**: Устанавливается форматтер для обработчика лога.
19. **logger.addHandler(file\_handler)**: Добавляется обработчик лога к логгеру.

Все эти строки кода настраивают окружение миграций Alembic, включая связь с базой данных, настройку логирования и определение метаданных SQLAlchemy для использования при создании миграций.

для получения дополнительной отладочной информации:

alembic upgrade 6e52590e930b –debug

Используйте оператор перенаправления **>** для записи вывода в файл.

alembic upgrade 6e52590e930b > migration\_script.sql

alembic revision --autogenerate -m "Initial migration"