Сириус
IT-Колледж

FastAPI

Параметры пути. Параметры запроса. Тело запроса. Базы данных SQL.

Сириус IT-Колледж

Параметры пути

Для определения "параметров" или "переменных" используется синтаксис форматирования строк Python:

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
async def read_item(item_id):
    return {"item_id": item_id}
```

Значение параметра пути `item_id` будет передано в функцию `read_item` как аргумент `item_id`.

Сириус
IT-Колледж

Параметры пути. Типы

Можно определить типы для параметров пути, используя синтаксис типов Python:

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

В этом случае, параметр `item_id` должен принимать только целые числа.

Сириус IT-Колледж

Параметры пути. Pydantic

Валидация данных выполняется с использованием Pydantic.

Pydantic — это библиотека Python, созданная Сэмюэлем Колвином, которая упрощает процесс проверки данных. Это универсальный инструмент, который можно использовать в различных сферах, таких как создание API, работа с базами данных и обработка данных в проектах. Библиотека имеет простой и интуитивно понятный синтаксис, позволяющий легко определять и проверять модели данных. Она включает в себя возможность указывать типы, значения по умолчанию и ограничения непосредственно в коде, что делает его понятным и удобным в сопровождении.

Несмотря на свою простоту, Pydantic также очень гибка и мощна, предоставляя встроенные средства проверки и ограничения, а также возможность создавать собственные средства проверки для дальнейшего определения и проверки данных. Данная библиотека доступна как для опытных разработчиков Python, так и для новичков, предлагая что-то для всех.

Pydantic позволяет использовать стандартные определения типов как str, float, bool и многие другие.

Сириус
IT-Колледж

Параметры пути. Порядок операций

Во время создания операций пути может возникнуть ситуация, когда у вас есть фиксированный путь. Например, вы можете определить путь `/users/me`, чтобы получить данные о текущем пользователе. А затем вы можете определить путь `/users/{user_id}`, чтобы получить данные о конкретном пользователе.

Так как операции выполняются в том порядке, в котором они определены в коде, вы должны определить пути сначала для `/users/me`, затем для `/users/{user_id}`.

Сириус IT-Колледж

Параметры пути. Порядок операций

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/users/me")
async def read_user_me():
    return {"user_id": "the current user"}


@app.get("/users/{user_id}")
async def read_user(user_id: str):
    return {"user_id": user_id}
```

В противном случае, путь `/users/{user_id}` будет так же выполняться, как путь `/users/me`, с параметром `user_id` со значением `me`.

Сириус IT-Колледж

Параметры пути. Порядок операций

Аналогично, нельзя переопределить пути:

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/users")
async def read_users():
    return ["Rick", "Morty"]


@app.get("/users")
async def read_users2():
    return ["Bean", "Elfo"]
```

Выполнятся всегда будет первая операция.

Сириус IT-Колледж

Параметры пути. Значения по умолчанию

Если у вас есть операция пути с параметром, но вы хотите иметь возможное корректное значение по умолчанию, вы можете использовать Python Enum.

```
from enum import Enum
from fastapi import FastAPI

class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"
    lenet = "lenet"

app = FastAPI()

@app.get("/models/{model_name}")
async def get_model(model_name: ModelName):
    if model_name is ModelName.alexnet:
        return {"model_name": model_name, "message": "Deep Learning FTW!"}
    if model_name.value == "lenet":
        return {"model_name": model_name, "message": "LeCNN all the images"}
    return {"model_name": model_name, "message": "Have some residuals"}
```

Сириус IT-Колледж

Параметры пути. Параметры содержащие пути

Допустим, если вы хотите определить путь, содержащий путь внутри `/files/{file_path}`. Но параметр `file_path` сам должен содержать путь, например `home/johndoe/myfile.txt`. Таким образом URL для этого пути будет выглядеть так: `/files/home/johndoe/myfile.txt`.

Напрямую такие операции не поддерживаются. Однако можно использовать следующий вариант: `/files/{file_path:path}`. В таком случае, имя параметра `file_path` и часть `:path` будут использованы для указания того, что параметр должен соответствовать любому пути.

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/files/{file_path:path}")
async def read_file(file_path: str):
    return {"file_path": file_path}
```

Сириус IT-Колледж

Параметры запроса

Когда вы определяете аргументы функции, которые не являются параметрами пути, они автоматически интерпретируются как параметры запроса.

```
from fastapi import FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]

@app.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
    return fake_items_db[skip : skip + limit]
```

Запрос - это набор ключ-значение параметров, которые передаются в URL после знака `?` и разделённых `&`.

Сириус IT-Колледж

Параметры запроса

Например, в URL:

```
http://127.0.0.1:8000/items/?skip=0&limit=10
```

следующие параметры:

- `skip`: со значением `0`
- `limit`: со значением `10`

Так как параметры запроса являются частью URL, они являются строками.

Но если вы определяете их с типами Python, то они будут преобразованы в соответствующие типы.

Параметры запроса. Значения по умолчанию

Так как параметры запроса не являются фиксированными частью URL, они могут быть опциональными и иметь значения по умолчанию. В примере ниже значения по умолчанию будут `skip=0` и `limit=10`.

```
http://127.0.0.1:8000/items/
```

Этот запрос аналогичен следующему:

```
http://127.0.0.1:8000/items/?skip=0&limit=10
```

Но, например, в запросе

```
http://127.0.0.1:8000/items/?skip=20
```

параметрами переданными в функцию будут:

- `skip=20`: так как он задан в URL
- `limit=10`: так как используется значение по умолчанию

Сириус IT-Колледж

Параметры запроса. Опциональные параметры

Опциональные параметры можно определить установив их значение по умолчанию в None.

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
async def read_item(item_id: str, q: str | None = None):
    if q:
        return {"item_id": item_id, "q": q}
    return {"item_id": item_id}
```

В таком случае, параметр `q` будет опционален и равен None по умолчанию.

Сириус IT-Колледж

Параметры запроса. Преобразование типов

Так же можно определить параметр типа `bool` и он будет преобразован

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
async def read_item(item_id: str, q: str | None = None, short: bool = False):
    item = {"item_id": item_id}
    if q:
        item.update({"q": q})
    if not short:
        item.update(
            {"description": "This is an amazing item that has a long description"}
        )
    return item
```

Сириус IT-Колледж

Параметры запроса. Преобразование типов

В таком случае, если перейти по URL:

```
http://127.0.0.1:8000/items/foo?short=1

http://127.0.0.1:8000/items/foo?short=True

http://127.0.0.1:8000/items/foo?short=true

http://127.0.0.1:8000/items/foo?short=on

http://127.0.0.1:8000/items/foo?short=yes
```

или любым другим вариациям (все в верхнем регистре, первая буква в верхнем регистре и т.д.), то параметр `short` будет преобразован в `True`.

Сириус IT-Колледж

Параметры запроса. Использование вместе с параметрами пути

Вместе с параметрами пути можно использовать параметры запроса, FastAPI сам определит что есть что по имени параметра.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users/{user_id}/items/{item_id}")
async def read_user_item(
    user_id: int, item_id: str, q: str | None = None, short: bool = False
):
    item = {"item_id": item_id, "owner_id": user_id}
    if q:
        item.update({"q": q})
    if not short:
        item.update(
            {"description": "This is an amazing item that has a long description"}
        )
    return item
```

Сириус IT-Колледж

Параметры запроса. Обязательные параметры

Когда вы задаете значение по умолчанию для параметра запроса, он необязателен.

Если вы не хотите задавать конкретное значение, просто установите значение по умолчанию в None.

Но если вы хотите, чтобы параметр запроса был обязательный, то вы можете просто не задавать значение по умолчанию.

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
async def read_user_item(item_id: str, needy: str):
    item = {"item_id": item_id, "needy": needy}
    return item
```

Здесь параметр `needy` является обязательным параметром типа `str`.

Сириус IT-Колледж

Параметры запроса

Возможно использование всех типов параметров запроса:

```
from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
async def read_user_item(
    item_id: str, needy: str, skip: int = 0, limit: int | None = None
):
    item = {"item_id": item_id, "needy": needy, "skip": skip, "limit": limit}
    return item
```

В данном случае, будет 3 параметра:

- `needy`, обязательный типа `str`.
- `skip`, `int` со значением по умолчанию `0`.
- `limit`, опциональный типа `int`.

Сириус
IT-Колледж

Тело запроса

Когда вам необходимо отправить данные из клиента (допустим, браузера) в ваш API, вы отправляете их как тело запроса.

Тело запроса - это данные, отправляемые клиентом в ваш API. **Тело ответа** - это данные, которые ваш API отправляет клиенту.

Ваш API почти всегда отправляет тело ответа. Но клиентам не обязательно всегда отправлять тело запроса.

Чтобы объявить тело запроса, необходимо использовать модели Pydantic, со всей их мощью и преимуществами.

Сириус IT-Колледж

Тело запроса

Первое, что вам необходимо сделать, это импортировать `BaseModel` из пакета `pydantic`. После этого вы описываете вашу модель данных как класс, наследующий от `BaseModel`.

Используйте аннотации типов Python для всех атрибутов:

```python
from typing import Union
from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: Union[float, None] = None
```

Сириус
IT-Колледж

Тело запроса

Также как и при описании параметров запроса, когда атрибут модели имеет значение по умолчанию, он является необязательным. Иначе он обязателен. Используйте `None`, чтобы сделать его необязательным без использования конкретных значений по умолчанию.

Например, модель выше описывает вот такой JSON "объект" (или словарь Python):

```
{
    "name": "Foo",
    "description": "An optional description",
    "price": 45.2,
    "tax": 3.5
}
```

Сириус IT-Колледж

Тело запроса

Чтобы добавить параметр к вашему обработчику, объявите его также, как вы объявляли параметры пути или параметры запроса:

```python
from typing import Union
from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Union[str, None] = None
    price: float
    tax: Union[float, None] = None

app = FastAPI()

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

...и укажите созданную модель в качестве типа параметра, `Item`.

Сириус IT-Колледж

Тело запроса

Внутри функции вам доступны все атрибуты объекта модели напрямую:

```
@app.post("/items/")
async def create_item(item: Item):
    item_dict = item.dict()
    if item.tax:
        price_with_tax = item.price + item.tax
        item_dict.update({"price_with_tax": price_with_tax})
    return item_dict
```

Сириус
IT-Колледж

Тело запроса + параметры пути

Вы можете одновременно объявлять параметры пути и тело запроса.

FastAPI распознает, какие параметры функции соответствуют параметрам пути и должны быть получены из пути, а какие параметры функции, объявленные как модели Pydantic, должны быть получены из тела запроса.

```python
@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item):
    return {"item_id": item_id, **item.dict()}
```

Сириус IT-Колледж

Тело запроса + параметры пути + параметры запроса

Вы также можете одновременно объявить параметры для пути, запроса и тела запроса.

```
@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item, q: Union[str, None] = None):
    result = {"item_id": item_id, **item.dict()}
    if q:
        result.update({"q": q})
    return result
```

Параметры функции распознаются следующим образом:

- Если параметр также указан в пути, то он будет использоваться как параметр пути.
- Если аннотация типа параметра содержит примитивный тип (int, float, str, bool и т.п.), он будет интерпретирован как параметр запроса.
- Если аннотация типа параметра представляет собой модель Pydantic, он будет интерпретирован как параметр тела запроса.

Сириус IT-Колледж

Базы данных

FastAPI не обязывает использовать базы данных SQL (реляционные базы данных).

Далее будет рассмотрен пример с использованием SQLAlchemy и SQLite.

Данный может быть адаптирован под любые базы данных, поддерживаемые фреймворком SQLAlchemy.

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Microsoft SQL Server, etc.

Для примера выбрана SQLite, так как он использует один файл и имеет интегрированный подход к Python.

Сириус IT-Колледж

Базы данных. ORM

Типичный подход - использование **ORM (Object-Relational Mapping, объектно-реляционное отображение, или преобразование)**.

При таком подходе, вы создаете класс который представляет собой таблицу в базе данных, каждый атрибут представляет столбец с именем и типом.

Например, класс `Pet` представляет собой таблицу `pets` в базе данных. Каждый объект этого класса представляет собой строку в таблице `pets`.

Например, объект `orion_cat` (это экземпляр `Pet`) может иметь атрибут `orion_cat.type`, для представления столбца `type`. Значение этого атрибута может быть, например, `"cat"`.

Модели ORM имеют инструменты для создания связей и отношений между таблицами или сущностями.

Таким образом, может быть атрибут `orion_cat.owner` и он может содержать информацию о владельце этого питомца из таблицы `owners`.

Базы данных. ORM

Так, `orion_cat.owner.name` может быть имя (из столбца `name` в таблице `owners`) владельца кота. Оно может иметь значение `"Arquilian"`.

ORM выполняет все этапы для взаимодействия с базой данных для получения информации о владельце кота, когда вы пытаетесь получить его из объекта `pet`.

Наиболее распространенные ORM: Django-ORM (часть фреймворка Django), SQLAlchemy ORM (часть фреймворка SQLAlchemy) and Peewee (отдельный фреймворк).

Далее будет использоваться SQLAlchemy ORM.

Сириус
IT-Колледж

Базы данных. Структура проекта

Пусть есть каталог `my_super_project` и в нем есть каталог `sql_app` со следующим содержимым:

```
.
└── sql_app
    ├── __init__.py
    ├── crud.py
    ├── database.py
    ├── main.py
    ├── models.py
    └── schemas.py
```

Файл `__init__.py` представляет собой пустой файл, но он указывает Python, что `sql_app` с вложенными файлами является пакетом.

Сириус IT-Колледж

Базы данных. SQLAlchemy

Рассмотрим файл `database.py` с содержимым:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./sql_app.db"
# SQLALCHEMY_DATABASE_URL = "postgresql://user:password@postgresserver/db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Базы данных. SQLAlchemy. Создание моделей

Рассмотрим файл `models.py` с содержимым:

```python
from sqlalchemy import Boolean, Column, ForeignKey, Integer, String
from sqlalchemy.orm import relationship
from .database import Base

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)
    items = relationship("Item", back_populates="owner")

class Item(Base):
    __tablename__ = "items"
    id = Column(Integer, primary_key=True)
    title = Column(String, index=True)
    description = Column(String, index=True)
    owner_id = Column(Integer, ForeignKey("users.id"))
    owner = relationship("User", back_populates="items")
```

Сириус
IT-Колледж

Базы данных. Pydantic модели/схемы

Необходимо создать модели Pydantic `ItemBase` и `UserBase` с такими же атрибутами для создания или чтения данных.

А так же `ItemCreate` и `UserCreate`, которые наследуются от них (т.е. имеют одни и те же атрибуты) плюс дополнительные атрибуты, необходимые для создания.

Так, если создать пользователя, то он должен иметь пароль.

Но для безопасности пароль не должен быть в других моделях Pydantic, например, он не будет отправлен из API при чтении пользователя.

Сириус IT-Колледж

Базы данных. Pydantic модели/схемы

Рассмотрим файл `schemas.py` с содержимым:

```python
from pydantic import BaseModel

class ItemBase(BaseModel):
    title: str
    description: str | None = None

class ItemCreate(ItemBase):
    pass

class Item(ItemBase):
    id: int
    owner_id: int
    class Config:
        orm_mode = True
```

```python
class UserBase(BaseModel):
    email: str

class UserCreate(UserBase):
    password: str

class User(UserBase):
    id: int
    is_active: bool
    items: list[Item] = []
    class Config:
        orm_mode = True
```

Сириус IT-Колледж

Базы данных. CRUD

CRUD - **C**reate, **R**ead, **U**pdate, and **D**elete

Рассмотрим файл `sql_app/crud.py`

```
from sqlalchemy.orm import Session

from . import models, schemas


def get_user(db: Session, user_id: int):
    return db.query(models.User).filter(models.User.id == user_id).first()


def get_user_by_email(db: Session, email: str):
    return db.query(models.User).filter(models.User.email == email).first()


def get_users(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.User).offset(skip).limit(limit).all()
```

Базы данных. CRUD

```
def create_user(db: Session, user: schemas.UserCreate):
    fake_hashed_password = user.password + "notreallyhashed"
    db_user = models.User(email=user.email, hashed_password=fake_hashed_password)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user


def get_items(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Item).offset(skip).limit(limit).all()


def create_user_item(db: Session, item: schemas.ItemCreate, user_id: int):
    db_item = models.Item(**item.dict(), owner_id=user_id)
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item
```

Сириус IT-Колледж

Основное приложение FastAPI

В файле `sql_app/main.py` создадим основное приложение FastAPI и объединим части вместе

```python
from fastapi import Depends, FastAPI, HTTPException
from sqlalchemy.orm import Session
from . import crud, models, schemas
from .database import SessionLocal, engine

models.Base.metadata.create_all(bind=engine)

app = FastAPI()

# Dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Сириус IT-Колледж

Основное приложение FastAPI

```python
@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = crud.get_user_by_email(db, email=user.email)
    if db_user:
        raise HTTPException(status_code=400, detail="Email already registered")
    return crud.create_user(db=db, user=user)

@app.get("/users/", response_model=list[schemas.User])
def read_users(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    users = crud.get_users(db, skip=skip, limit=limit)
    return users

@app.get("/users/{user_id}", response_model=schemas.User)
def read_user(user_id: int, db: Session = Depends(get_db)):
    db_user = crud.get_user(db, user_id=user_id)
    if db_user is None:
        raise HTTPException(status_code=404, detail="User not found")
    return db_user
```

Сириус IT-Колледж

Основное приложение FastAPI

```
@app.post("/users/{user_id}/items/",
          response_model=schemas.Item)
def create_item_for_user(user_id: int, item: schemas.ItemCreate,
                         db: Session = Depends(get_db)):
    return crud.create_user_item(db=db, item=item, user_id=user_id)


@app.get("/items/", response_model=list[schemas.Item])
def read_items(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    items = crud.get_items(db, skip=skip, limit=limit)
    return items
```