

Секция Backend Python

Pydantic и API

Виктор Швайковский

Ведущий инженер машинного обучения

Обо мне

- С 2020 года в разработке
- Устроился в компанию через ШИФТ
- Вырос из разработчика в тимлиды
- Играю в шахматы, решаю Leetcode



2

ШИФТ



План лекции

1. Типизация в Python – зачем и как
2. OpenAPI – стандарт описания API
3. Pydantic – валидация и сериализация данных
4. Dependency Injection – управление зависимостями
5. Практика – всё вместе в вашем проекте



Проблема – API без контракта

```
# Backend разработчик написал:  
@app.post("/users")  
def create_user(data: dict):  
    return {"id": 1, "name": data["name"]}  
  
# Frontend разработчик спрашивает:  
# - Какие поля нужно передать?  
# - Какой формат даты?  
# - Что вернётся при ошибке?  
# - Где документация?
```

Python – динамически типизированный язык

```
def process_user(user):
    return user["name"].upper()

process_user({"name": "Alice"})          # OK
process_user({"username": "Bob"})        # KeyError: 'name'
process_user("Alice")                   # TypeError: string indices must be integers
process_user(None)                     # TypeError: 'NoneType' is not subscriptable
process_user(42)                       # TypeError: 'int' is not subscriptable

# Узнаём об ошибке только в runtime
```

Type Hints – подсказки типов

```
# Без аннотаций – непонятно, что принимает и возвращает
def greet(name):
    return f"Hello, {name}!"

# С аннотациями – всё ясно
def greet(name: str) -> str:
    return f"Hello, {name}!"

# Сложные типы
from typing import Optional

def find_user(user_id: int) -> Optional[dict]:
    """Возвращает пользователя или None, если не найден"""
    ...

def process_items(items: list[str], limit: int = 10) -> list[str]:
    """Обрабатывает список строк"""
    return items[:limit]
```

Зачем нужны аннотации типов?

- 1. Документация в коде**
 - Типы говорят, что функция ожидает, без чтения docstring
- 2. Автодополнение в IDE**
 - VS Code / PyCharm знают, какие методы доступны
- 3. Раннее обнаружение ошибок**
 - MyPy, Pyright находят баги до запуска кода
- 4. Безопасный рефакторинг**
 - Переименовали класс – IDE покажет все места использования
- 5. Pydantic & FastAPI**
 - Типы используются в валидации и при генерации документации

От типов к валидации – проблема

```
def create_user(name: str, age: int) -> dict:  
    return {"name": name, "age": age}  
  
# Python не проверяет типы при вызове  
create_user("Alice", 25)          # ✅ Ожидаемо работает  
create_user("Alice", "молодой") # ⚠️ Тоже работает  
create_user(None, None)         # ⚠️ И это работает  
  
# Нам нужен инструмент, который:  
# 1. Читает аннотации типов  
# 2. Проверяет данные в runtime  
# 3. Выдаёт понятные ошибки
```

Pydantic – типы становятся реальностью

- Читает аннотации типов
- Валидирует данные в runtime
- Конвертирует совместимые типы
- Выдаёт понятные ошибки
- Генерирует JSON Schema для OpenAPI

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int

# Теперь типы проверяются:
user = User(name="Alice", age=25)      # ✅ OK
user = User(name="Alice", age="25")      # ✅ OK! Pydantic
# конвертирует "25" → 25
user = User(name="Alice", age="молодой") # ❌ ValidationError!
```

Решение – OpenAPI

- OpenAPI – это стандарт описания REST API, независимый от языка программирования
- Ключевые преимущества:
 - Единый контракт между frontend и backend
 - Автоматическая генерация документации
 - Генерация клиентского кода
 - Валидация запросов и ответов
 - Формальное описание структуры JSON – какие поля, типы и форматы ожидаем на входе и возвращаем на выходе



Структура OpenAPI документа

```
openapi: 3.1.0
info:
  title: User Service API
  version: 1.0.0
paths:
  /users:
    post:
      summary: Создать пользователя
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/UserCreate'
      responses:
        '201':
          description: Пользователь создан
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
```

FastAPI + OpenAPI

```
from fastapi import FastAPI

app = FastAPI(
    title="User Service",
    description="API для управления пользователями",
    version="1.0.0"
)

# Документация доступна автоматически:
# http://localhost:8000/docs      – Swagger UI
# http://localhost:8000/redoc      – ReDoc
# http://localhost:8000/openapi.json – Raw JSON
```

Swagger UI

User Service API 1.0.0 OAS 3.1

/openapi.json

← Название, версия

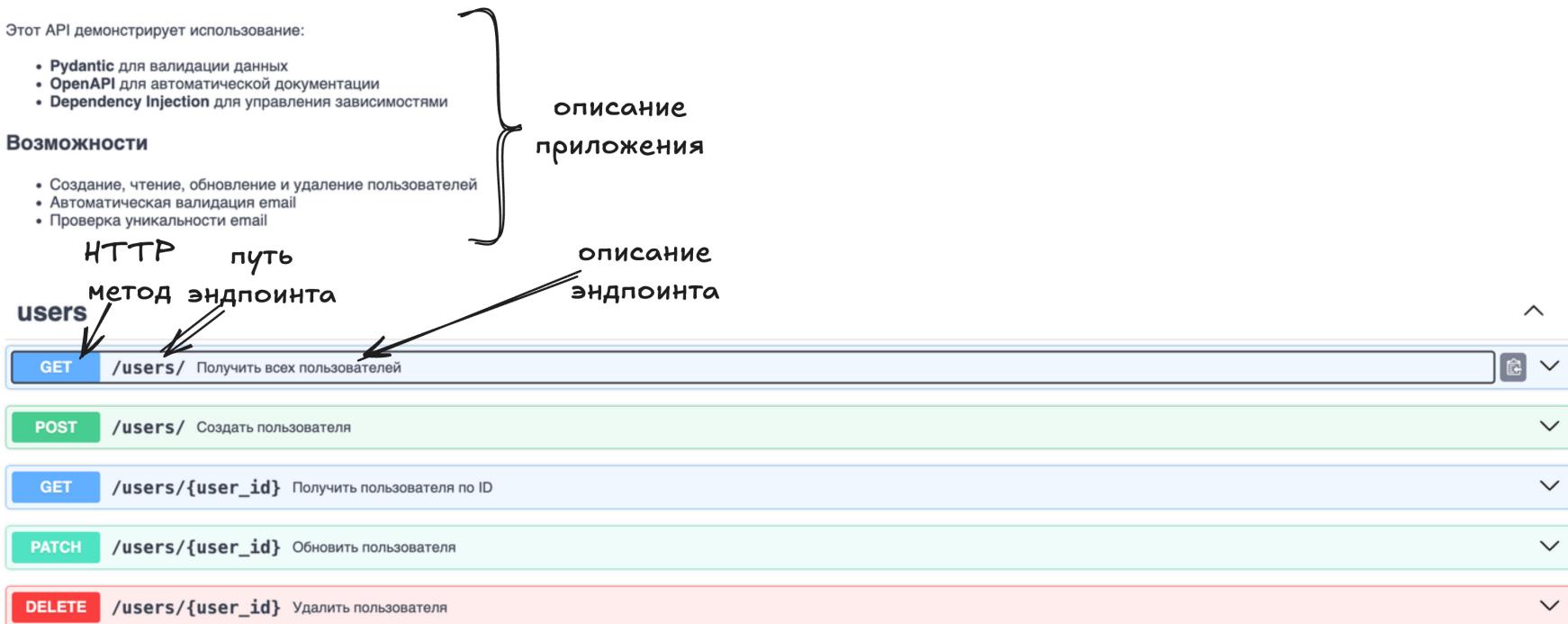
Демо-приложение

Этот API демонстрирует использование:

- Pydantic для валидации данных
- OpenAPI для автоматической документации
- Dependency Injection для управления зависимостями

Возможности

- Создание, чтение, обновление и удаление пользователей
- Автоматическая валидация email
- Проверка уникальности email



13

ШИФР



Deep dive в Pydantic

```
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name: str
    email: str
    is_active: bool = True # значение по умолчанию

# Создание объекта
user = User(id=1, name="Alice", email="alice@example.com")

# Автоматическая валидация и преобразование типов
user = User(id="1", name="Alice", email="alice@example.com")
```

Валидация в действии

```
from pydantic import BaseModel, ValidationError

class User(BaseModel):
    id: int
    email: str

    try:
        user = User(id="not_a_number", email="invalid")
    except ValidationError as e:
        print(e.json())
```

Output:

```
[{"loc": ["id"], "msg": "Input should be a valid integer", "type": "int_parsing"}]
```

Типы полей в Pydantic

```
from datetime import datetime
from typing import Optional
from pydantic import BaseModel, EmailStr, HttpUrl

class UserProfile(BaseModel):
    # Базовые типы
    username: str
    age: int
    balance: float

    # Optional – может быть None
    bio: Optional[str] = None

    # Специальные типы Pydantic
    email: EmailStr          # валидирует email
    website: HttpUrl         # валидирует URL

    # Datetime
    created_at: datetime
```

Вложенные модели

```
from pydantic import BaseModel

class Address(BaseModel):
    city: str
    street: str
    building: int

class Company(BaseModel):
    name: str
    address: Address # вложенная модель

class User(BaseModel):
    name: str
    company: Company # ещё одна вложенная модель

# Использование:
user = User(
    name="Alice",
    company={
        "name": "TechCorp",
        "address": {"city": "Новосибирск", "street": "Ленина", "building": 1}
    }
)
```

Списки и множества

```
from pydantic import BaseModel

class Order(BaseModel):
    id: int
    items: list[str]          # список строк
    quantities: list[int]      # список чисел
    tags: set[str]             # множество (уникальные)

class User(BaseModel):
    name: str
    orders: list[Order]        # список вложенных моделей

# Использование
user = User(
    name="Alice",
    orders=[
        {"id": 1, "items": ["apple", "banana"], "quantities": [2, 3], "tags": {"fruit"}},
        {"id": 2, "items": ["bread"], "quantities": [1], "tags": {"bakery"}}
    ]
)
```

Кастомная валидация с field_validator

```
from pydantic import BaseModel, field_validator

class User(BaseModel):
    username: str
    age: int

    @field_validator('username')
    @classmethod
    def username_must_be_alphanumeric(cls, v: str) -> str:
        if not v.isalnum():
            raise ValueError('username должен содержать только буквы и цифры')
        return v.lower() # можно преобразовать значение

    @field_validator('age')
    @classmethod
    def age_must_be_positive(cls, v: int) -> int:
        if v < 0 or v > 150:
            raise ValueError('age должен быть от 0 до 150')
        return v
```

model_validator для сложной логики

```
from pydantic import BaseModel, model_validator

class DateRange(BaseModel):
    start_date: datetime
    end_date: datetime

    @model_validator(mode='after')
    def check_dates(self) -> 'DateRange':
        if self.start_date >= self.end_date:
            raise ValueError('start_date должен быть раньше end_date')
        return self

class PasswordChange(BaseModel):
    password: str
    password_confirm: str

    @model_validator(mode='after')
    def passwords_match(self) -> 'PasswordChange':
        if self.password != self.password_confirm:
            raise ValueError('Пароли не совпадают')
        return self
```

Паттерн Request/Response моделей

```
# Что приходит от клиента при создании
class UserCreate(BaseModel):
    username: str
    email: EmailStr
    password: str

# Что возвращаем клиенту
class UserResponse(BaseModel):
    id: int
    username: str
    email: EmailStr
    created_at: datetime
    # password НЕ возвращаем!

# Для обновления – все поля опциональны
class UserUpdate(BaseModel):
    username: Optional[str] = None
    email: Optional[EmailStr] = None
```

Pydantic в FastAPI эндпоинтах

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserCreate(BaseModel):
    username: str
    email: EmailStr

class UserResponse(BaseModel):
    id: int
    username: str
    email: EmailStr

@app.post("/users", response_model=UserResponse)
def create_user(user: UserCreate) -> UserResponse:
    return UserResponse(
        id=1,
        username=user.username,
        email=user.email
    )
```

Краткое резюме

- Типизация Python – аннотации как документация
- Pydantic превращает типы в валидацию
- OpenAPI – стандарт документации API
- FastAPI генерирует OpenAPI автоматически
- BaseModel – основа для создания схем
- Встроенные типы: EmailStr, HttpUrl, datetime...
- Кастомные валидаторы: field_validator, model_validator
- Паттерн: UserCreate, UserResponse, UserUpdate



Проблема – повторяющийся код

```
@app.get("/users/{user_id}")
def get_user(user_id):
    db = get_database_connection()    # повторяется
    user = db.get_user(user_id)
    if not user:
        raise HTTPException(404)
    return user

@app.delete("/users/{user_id}")
def delete_user(user_id):
    db = get_database_connection()    # повторяется
    user = db.get_user(user_id)        # повторяется
    if not user:                      # повторяется
        raise HTTPException(404)      # повторяется
    db.delete(user)
    return {"status": "deleted"}
```

Что такое Dependency Injection

✗ Без DI

```
get_users()
```

```
    db = Database()
```

- Жёсткая связь
- Сложно тестировать
- Нельзя подменить

✓ С DI

FastAPI (DI Container)

```
Database → get_users
```

inject

```
def get_users(  
    db: Database = Depends(get_db)  
):
```

- Слабая связь
- Легко тестировать (mock)
- Можно подменить реализацию

В тестах:
`app.dependency_overrides[
 get_db] = mock_db`

DI в FastAPI – функция Depends

```
from fastapi import FastAPI, Depends

app = FastAPI()

# Зависимость – обычная функция
def get_database():
    db = Database()
    try:
        yield db
    finally:
        db.close()

# Используем через Depends
@app.get("/users")
def get_users(db = Depends(get_database)):
    return db.get_all_users()

@app.get("/orders")
def get_orders(db = Depends(get_database)):
    return db.get_all_orders()
```

Типичные зависимости

```
# 1. Подключение к базе данных
def get_db() -> Generator[Session, None, None]:
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# 2. Текущий пользователь (аутентификация)
def get_current_user(token: str = Header()) -> User:
    user = decode_token(token)
    if not user:
        raise HTTPException(401, "Invalid token")
    return user

# 3. Параметры пагинации
def get_pagination(skip: int = 0, limit: int = 100) -> dict:
    return {"skip": skip, "limit": min(limit, 100)}
```

Зависимости с параметрами

```
from fastapi import Depends, HTTPException

# Зависимость с параметрами из path
def get_item_or_404(item_id: int, db = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if not item:
        raise HTTPException(404, f"Item {item_id} not found")
    return item

@app.get("/items/{item_id}")
def read_item(item: Item = Depends(get_item_or_404)):
    return item

@app.put("/items/{item_id}")
def update_item(item: Item = Depends(get_item_or_404), data: ItemUpdate = Body()):
    # item уже найден и проверен
    item.update(data)
    return item
```

Цепочки зависимостей

```
def get_db():
    ...

def get_current_user(db = Depends(get_db)):
    # Нужна база, чтобы найти пользователя
    ...

def get_admin_user(user = Depends(get_current_user)):
    # Нужен текущий пользователь
    if not user.is_admin:
        raise HTTPException(403, "Admin required")
    return user

@app.delete("/users/{user_id}")
def delete_user(
    user_id: int,
    admin = Depends(get_admin_user), # Проверит, что это админ
    db = Depends(get_db)
):
    # Сюда попадём только если админ
    ...
```

Классы как зависимости

```
class Pagination:
    def __init__(self, skip: int = 0, limit: int = 100):
        self.skip = skip
        self.limit = min(limit, 100) # Защита от слишком больших limit

class ItemFilter:
    def __init__(
        self,
        category: Optional[str] = None,
        min_price: Optional[float] = None,
        max_price: Optional[float] = None
    ):
        self.category = category
        self.min_price = min_price
        self.max_price = max_price

@app.get("/items")
def get_items(
    pagination: Pagination = Depends(),
    filters: ItemFilter = Depends()
):
    ...
    ...
```

DI и тестирование

```
# app/main.py
def get_db():
    return RealDatabase()

@app.get("/users")
def get_users(db = Depends(get_db)):
    return db.get_all()

# tests/test_main.py
def get_test_db():
    return FakeDatabase() # Мок

# Подменяем зависимость для тестов
app.dependency_overrides[get_db] = get_test_db

client = TestClient(app)
response = client.get("/users")
# Теперь используется FakeDatabase
```

Практический пример – полный эндпоинт

```
from fastapi import FastAPI, Depends, HTTPException
from pydantic import BaseModel, EmailStr

app = FastAPI(title="User Service")

class UserCreate(BaseModel):
    email: EmailStr
    name: str

class UserResponse(BaseModel):
    id: int
    email: str
    name: str

def get_db():
    db = Database()
    try:
        yield db
    finally:
        db.close()

@app.post("/users", response_model=UserResponse, status_code=201)
def create_user(user: UserCreate, db = Depends(get_db)) -> UserResponse:
    if db.user_exists(user.email):
        raise HTTPException(400, "Email already registered")
    new_user = db.create_user(user.email, user.name)
    return UserResponse(id=new_user.id, email=new_user.email, name=new_user.name)
```

Best Practices

- Один файл – одна ответственность
- Модели в отдельной папке schemas/
- Зависимости в dependencies/
- Группируйте эндпоинты в routers

Структура проекта:

```
app/
  ├── main.py          # FastAPI app
  └── schemas/
      └── user.py      # Pydantic модели
  ├── dependencies/
      └── database.py  # Зависимости
  └── routers/
      └── users.py     # Эндпоинты
```

Полезные ресурсы

- Официальная документация:
 - FastAPI – <https://fastapi.tiangolo.com/>
 - Pydantic – <https://docs.pydantic.dev/>
- По теме лекции:
 - Request Body – <https://fastapi.tiangolo.com/tutorial/body/>
 - Pydantic Field Types – <https://docs.pydantic.dev/latest/concepts/fields/>
 - Dependencies – <https://fastapi.tiangolo.com/tutorial/dependencies/>
 - Python Type Hints – <https://docs.python.org/3/library/typing.html>