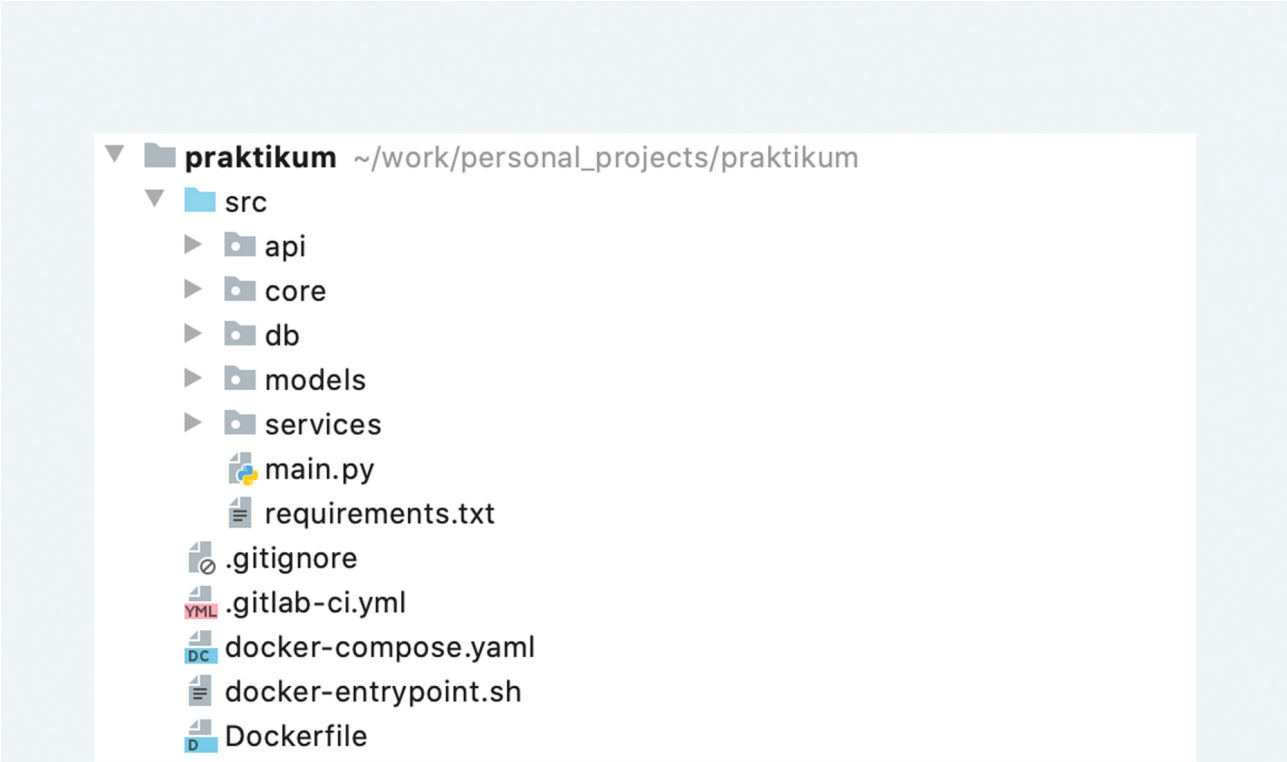
FastAPI

Работая с Django, вы привыкли к структуре проекта, которую предоставляет фреймворк. Но у FastAPI, как и у любых других микрофреймворков, отсутствует заранее предусмотренная структура проекта.

Чтобы проект не превратился в бесформенную лужу, у него должна быть структура. Поэтому создадим каркас для проекта так, как это обычно делают в компаниях.



- Корень проекта — в нём находятся базовые вещи, например, Dockerfile, ci и gitignore.
- src — содержит исходный код приложения.
- main.py — входная точка приложения.
- api — модуль, в котором реализуется API. Другими словами, это модуль для предоставления http-интерфейса клиентским приложениям. Внутри модуля отсутствует какая-либо бизнес-логика, так как она не должна быть завязана на HTTP.
- core — содержит разные конфигурационные файлы.

- db — предоставляет объекты баз данных (Redis, Elasticsearch) и провайдеры для внедрения зависимостей. Redis будет использоваться для кеширования, чтобы не нагружать лишний раз Elasticsearch.
- models — содержит классы, описывающие бизнес-сущности, например, фильмы, жанры, актёров.
- services — главное в сервисе. В этом модуле находится реализация всей бизнес-логики. Таким образом она отделена от транспорта. Благодаря такому разделению, вам будет легче добавлять новые типы транспортов в сервис. Например, легко добавить RPC протокол поверх AMQP или Websockets.

Начнём проект с конфигов. Создадим файл core/config.py.

Скопировать код
PYTHON
```python
import os
from logging import config as logging_config

from core.logger import LOGGING

# Применяем настройки логирования
logging_config.dictConfig(LOGGING)

# Название проекта. Используется в Swagger-документации
PROJECT_NAME = os.getenv('PROJECT_NAME', 'movies')

# Настройки Redis
REDIS_HOST = os.getenv('REDIS_HOST', '127.0.0.1')
REDIS_PORT = int(os.getenv('REDIS_PORT', 6379))

# Настройки Elasticsearch
ELASTIC_HOST = os.getenv('ELASTIC_HOST', '127.0.0.1')
ELASTIC_PORT = int(os.getenv('ELASTIC_PORT', 9200))

# Корень проекта
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

Создадим файл с конфигурацией логов core/logger.py.

Скопировать код
PYTHON
```python
LOG_FORMAT = '%(asctime)s - %(name)s - %(levelname)s - %
(message)s'
```

```
LOG_DEFAULT_HANDLERS = ['console', ]

# В логгере настраивается логгирование uvicorn-сервера.
# Про логирование в Python можно прочитать в документации
# https://docs.python.org/3/howto/logging.html
# https://docs.python.org/3/howto/logging-cookbook.html

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': LOG_FORMAT
        },
        'default': {
            '()': 'uvicorn.logging.DefaultFormatter',
            'fmt': '%(levelprefix)s %(message)s',
            'use_colors': None,
        },
        'access': {
            '()': 'uvicorn.logging.AccessFormatter',
            'fmt': "%(levelprefix)s %(client_addr)s - '%
(request_line)s' %(status_code)s",
        },
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'verbose',
        },
        'default': {
            'formatter': 'default',
            'class': 'logging.StreamHandler',
            'stream': 'ext://sys.stdout',
        },
        'access': {
            'formatter': 'access',
            'class': 'logging.StreamHandler',
            'stream': 'ext://sys.stdout',
        },
    },
    'loggers': {
        '': {
            'handlers': LOG_DEFAULT_HANDLERS,
            'level': 'INFO',
        },
        'uvicorn.error': {
            'level': 'INFO',
        },
        'uvicorn.access': {
            'handlers': ['access'],
```

```
            'level': 'INFO',
            'propagate': False,
        },
    },
    'root': {
        'level': 'INFO',
        'formatter': 'verbose',
        'handlers': LOG_DEFAULT_HANDLERS,
    },
}
```

Создайте main.py.

Скопировать код
PYTHON

```python
import logging

import uvicorn as uvicorn
from fastapi import FastAPI
from fastapi.responses import ORJSONResponse

from core import config
from core.logger import LOGGING

app = FastAPI(
    # Конфигурируем название проекта. Оно будет отображаться в документации
    title=config.PROJECT_NAME,
    # Адрес документации в красивом интерфейсе
    docs_url='/api/openapi',
    # Адрес документации в формате OpenAPI
    openapi_url='/api/openapi.json',
    # Можно сразу сделать небольшую оптимизацию сервиса
        # и заменить стандартный JSON-сереализатор на более шуструю версию, написанную на Rust
    default_response_class=ORJSONResponse,
)

if __name__ == '__main__':
    # Приложение должно запускаться с помощью команды
    # `uvicorn main:app --host 0.0.0.0 --port 8000`
    # Но таким способом проблематично запускать сервис в дебагере,
        # поэтому сервер приложения для отладки запускаем здесь
    uvicorn.run(
        'main:app',
        host='0.0.0.0',
        port=8000,
        log_config=LOGGING,
        log_level=logging.DEBUG,
```

```
    )
```

Запустите сервис. После запуска должна открываться страница с документацией `http://0.0.0.0:8000/api/openapi`.

Теперь объявим в модуле `db` соединения с Elasticsearch и Redis. Создайте следующие файлы:

- db/elastic.py

```
Скопировать код
PYTHON
  from elasticsearch import AsyncElasticsearch

        es: AsyncElasticsearch = None

        # Функция понадобится при внедрении зависимостей
        async def get_elastic() -> AsyncElasticsearch:
            return es
```

- db/redis.py

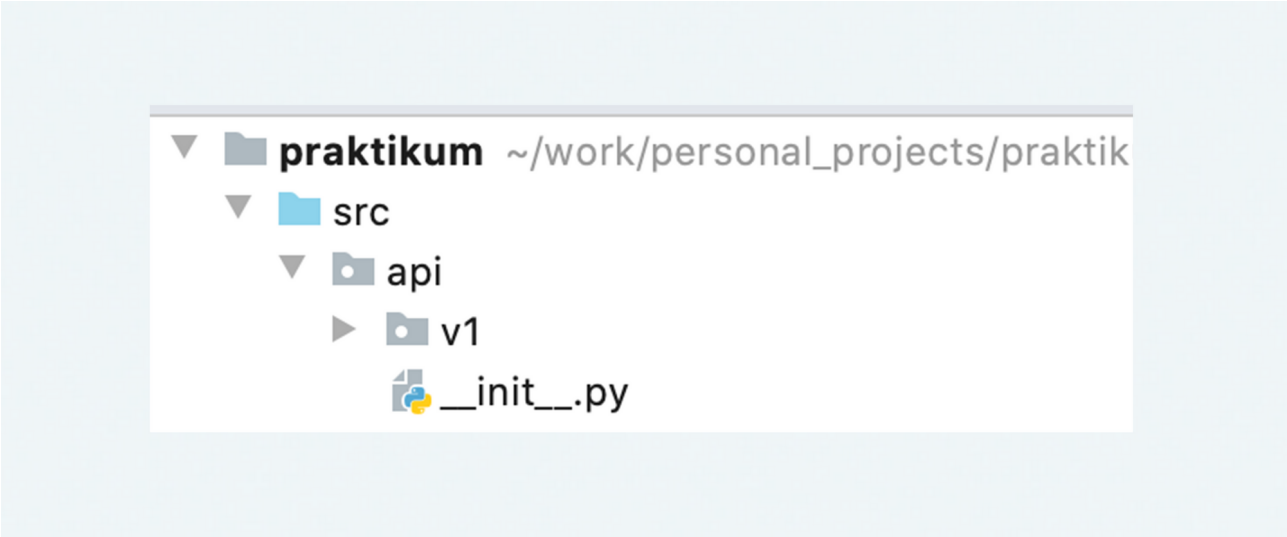
```
Скопировать код
PYTHON
  from aioredis import Redis

        redis: Redis = None

        # Функция понадобится при внедрении зависимостей
        async def get_redis() -> Redis:
            return redis
```

Изменим main.py, чтобы подключить соединения к базам.

```
Скопировать код
PYTHON
import logging

import aioredis
import uvicorn as uvicorn
```

```
from elasticsearch import AsyncElasticsearch
from fastapi import FastAPI
from fastapi.responses import ORJSONResponse

from core import config
from core.logger import LOGGING
from db import elastic
from db import redis

app = FastAPI(
    title=config.PROJECT_NAME,
    docs_url='/api/openapi',
    openapi_url='/api/openapi.json',
    default_response_class=ORJSONResponse,
)


@app.on_event('startup')
async def startup():
    # Подключаемся к базам при старте сервера
    # Подключиться можем при работающем event-loop
    # Поэтому логика подключения происходит в асинхронной функции
    redis.redis = await
aioredis.create_redis_pool((config.REDIS_HOST, config.REDIS_PORT),
minsize=10, maxsize=20)
    elastic.es =
AsyncElasticsearch(hosts=[f'{config.ELASTIC_HOST}:
{config.ELASTIC_PORT}'])


@app.on_event('shutdown')
async def shutdown():
    # Отключаемся от баз при выключении сервера
    await redis.redis.close()
    await elastic.es.close()


if __name__ == '__main__':
    uvicorn.run(
        'main:app',
        host='0.0.0.0',
        port=8000,
        log_config=LOGGING,
        log_level=logging.DEBUG,
    )
```

Создадим первый обработчик HTTP-запросов. Создадим модуль `api`, а внутри него — модуль `v1`. У вас должна получиться такая структура:



Вы создали модуль `v1`, чтобы удобно версионировать методы API. Версионирование позволяет менять API без ущерба для конечных пользователей.

Внутри `v1` создайте файл `film.py`.

Скопировать код
PYTHON

```python
from fastapi import APIRouter
from pydantic import BaseModel

# Объект router, в котором регистрируем обработчики
router = APIRouter()

# FastAPI в качестве моделей использует библиотеку pydantic
# https://pydantic-docs.helpmanual.io
# У неё есть встроенные механизмы валидации, сериализации и
десериализации
# Также она основана на дата-классах

# Модель ответа API
class Film(BaseModel):
    id: str
    title: str

# С помощью декоратора регистрируем обработчик film_details
# На обработку запросов по адресу <some_prefix>/some_id
# Позже подключим роутер к корневому роутеру
# И адрес запроса будет выглядеть так — /api/v1/film/some_id
# В сигнатуре функции указываем тип данных, получаемый из адреса
запроса (film_id: str)
# И указываем тип возвращаемого объекта — Film
@router.get('/{film_id}', response_model=Film)
```

```
async def film_details(film_id: str) -> Film:
    return Film(id='some_id', title='some_title')
```

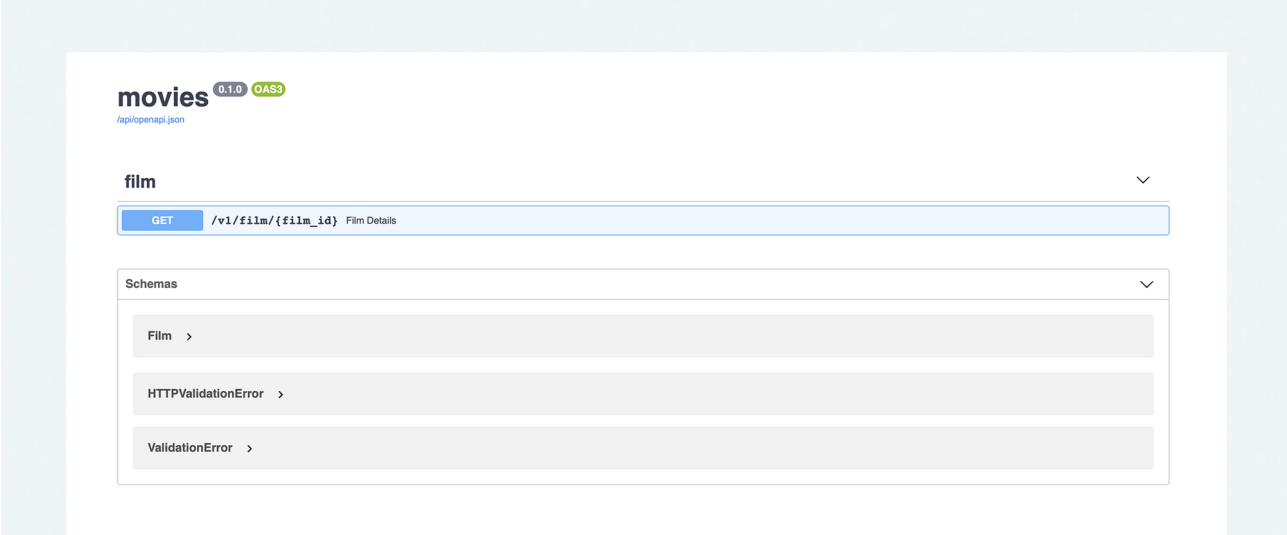
Подключим роутер фильмов к приложению. Отредактируем main.py.

Скопировать код
PYTHON

```
import logging

import aioredis
import uvicorn as uvicorn
from elasticsearch import AsyncElasticsearch
from fastapi import FastAPI
from fastapi.responses import ORJSONResponse

from api.v1 import film
from core import config
from core.logger import LOGGING
from db import elastic, redis

app = FastAPI(
    title=config.PROJECT_NAME,
    docs_url='/api/openapi',
    openapi_url='/api/openapi.json',
    default_response_class=ORJSONResponse,
)


@app.on_event('startup')
async def startup():
    redis.redis = await
aioredis.create_redis_pool((config.REDIS_HOST, config.REDIS_PORT),
minsize=10, maxsize=20)
    elastic.es =
AsyncElasticsearch(hosts=[f'{config.ELASTIC_HOST}:
{config.ELASTIC_PORT}'])


@app.on_event('shutdown')
async def shutdown():
    await redis.redis.close()
    await elastic.es.close()


# Подключаем роутер к серверу, указав префикс /v1/film
# Теги указываем для удобства навигации по документации
app.include_router(film.router, prefix='/v1/film', tags=['film'])

if __name__ == '__main__':
```

```
    uvicorn.run(
        'main:app',
        host='0.0.0.0',
        port=8000,
        log_config=LOGGING,
        log_level=logging.DEBUG,
    )
```

Запустите приложение. В документации появится описание добавленного метода http://localhost:8000/api/openapi.



Теперь настало время реализовать бизнес-логику. FastAPI пропагандирует использование подхода внедрения зависимостей — dependency injection).

DI позволяет развязать структуру кода. При таком подходе бизнес-логика перестаёт быть зависимой от работы с конкретной БД или фреймворком, что позволяет быстро менять их при необходимости. Второе преимущество — упрощение тестирования приложений, так как не придётся делать множество monkey patch в коде тестов. Это упростит читаемость тестов для других разработчиков и сделает код чище.

В FastAPI зависимости указывается с помощью функции Depends, в которую передаётся функция-провайдер. Она возвращает необходимую зависимость. Функция-провайдер также предоставляет свои зависимости.

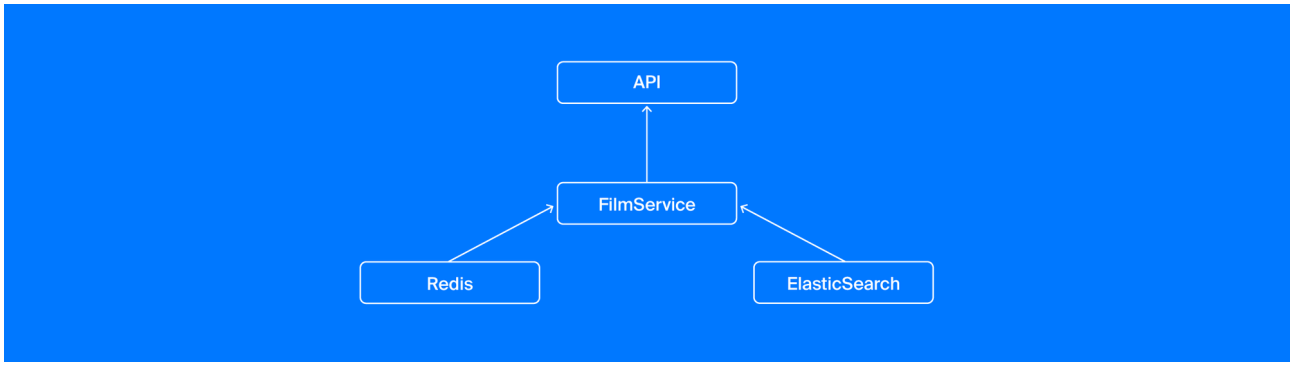
Перейдём к практике, чтобы сразу ощутить магию.

В модуле `services` создайте файл `film.py`.

Скопировать код
PYTHON

```python
from functools import lru_cache

from aioredis import Redis
from elasticsearch import AsyncElasticsearch
from fastapi import Depends

from db.elastic import get_elastic
from db.redis import get_redis


# FilmService содержит бизнес-логику по работе с фильмами.
# Никакой магии тут нет. Обычный класс с обычными методами.
# Этот класс ничего не знает про DI — максимально сильный и независимый.
class FilmService:
    def __init__(self, redis: Redis, elastic: AsyncElasticsearch):
        self.redis = redis
        self.elastic = elastic


# get_film_service — это провайдер FilmService.
# С помощью Depends он сообщает, что ему необходимы Redis и Elasticsearch
# Для их получения вы ранее создали функции-провайдеры в модуле db
# Используем lru_cache-декоратор, чтобы создать объект сервиса в едином экземпляре (синглтона)
@lru_cache()
def get_film_service(
        redis: Redis = Depends(get_redis),
        elastic: AsyncElasticsearch = Depends(get_elastic),
) -> FilmService:
    return FilmService(redis, elastic)
```

Граф зависимостей будет выглядеть следующим образом.



Создадим модель фильма в `models/film.py`. Ранее вы уже создавали модель `Film` в `api/v1/film.py`, но она используется исключительно для представления данных по HTTP. Внутренние модели, одну из которых вы создаёте сейчас, используется только в рамках бизнес-логики.

Скопировать код
PYTHON

```python
import orjson

# Используем pydantic для упрощения работы при перегонке данных из json в объекты
from pydantic import BaseModel


class Film(BaseModel):
    id: str
    title: str
    description: str

    class Config:
        # Заменяем стандартную работу с json на более быструю
        json_loads = orjson.loads
        json_dumps = orjson.dumps
```

Реализуем бизнес-логику для получения фильмов по id.

Скопировать код
PYTHON

```python
from functools import lru_cache
from typing import Optional

from aioredis import Redis
from elasticsearch import AsyncElasticsearch
from fastapi import Depends

from db.elastic import get_elastic
from db.redis import get_redis
from models.film import Film

FILM_CACHE_EXPIRE_IN_SECONDS = 60 * 5  # 5 минут


class FilmService:
    def __init__(self, redis: Redis, elastic: AsyncElasticsearch):
        self.redis = redis
        self.elastic = elastic

    # get_by_id возвращает объект фильма. Он опционален, так как фильм может отсутствовать в базе
    async def get_by_id(self, film_id: str) -> Optional[Film]:
        # Пытаемся получить данные из кеша, потому что оно работает быстрее
        film = await self._film_from_cache(film_id)
        if not film:
            # Если фильма нет в кеше, то ищем его в Elasticsearch
            film = await self._get_film_from_elastic(film_id)
            if not film:
                # Если он отсутствует в Elasticsearch, значит, фильма вообще нет в базе
                return None
            # Сохраняем фильм  в кеш
            await self._put_film_to_cache(film)

        return film

    async def _get_film_from_elastic(self, film_id: str) -> Optional[Film]:
        doc = await self.elastic.get('movies', film_id)
        return Film(**doc['_source'])

    async def _film_from_cache(self, film_id: str) -> Optional[Film]:
        # Пытаемся получить данные о фильме из кеша, используя команду get
```

```
        # https://redis.io/commands/get
        data = await self.redis.get(film_id)
        if not data:
            return None

        # pydantic предоставляет удобное API для создания объекта
моделей из json
        film = Film.parse_raw(data)
        return film

    async def _put_film_to_cache(self, film: Film):
        # Сохраняем данные о фильме, используя команду set
        # Выставляем время жизни кеша — 5 минут
        # https://redis.io/commands/set
        # pydantic позволяет сериализовать модель в json
        await self.redis.set(film.id, film.json(),
expire=FILM_CACHE_EXPIRE_IN_SECONDS)


@lru_cache()
def get_film_service(
        redis: Redis = Depends(get_redis),
        elastic: AsyncElasticsearch = Depends(get_elastic),
) -> FilmService:
    return FilmService(redis, elastic)
```

Подключите сервис к API.

```
Скопировать код
PYTHON
from http import HTTPStatus

from fastapi import APIRouter, Depends, HTTPException
from pydantic import BaseModel

from services.film import FilmService, get_film_service

router = APIRouter()


class Film(BaseModel):
    id: str
    title: str


# Внедряем FilmService с помощью Depends(get_film_service)
@router.get('/{film_id}', response_model=Film)
async def film_details(film_id: str, film_service: FilmService =
Depends(get_film_service)) -> Film:
```

```
    film = await film_service.get_by_id(film_id)
    if not film:
        # Если фильм не найден, отдаём 404 статус
        # Желательно пользоваться уже определёнными HTTP-
статусами, которые содержат enum
                # Такой код будет более поддерживаемым
        raise HTTPException(status_code=HTTPStatus.NOT_FOUND,
detail='film not found')

    # Перекладываем данные из models.Film в Film
    # Обратите внимание, что у модели бизнес-логики есть поле
description
        # Которое отсутствует в модели ответа API.
        # Если бы использовалась общая модель для бизнес-логики и
формирования ответов API
        # вы бы предоставляли клиентам данные, которые им не нужны
        # и, возможно, данные, которые опасно возвращать
    return Film(id=film.id, title=film.title)
```

Итог

Вы познакомились с FastAPI и научились настраивать проект с нуля. Также вы познакомились с подходом внедрения зависимостей, научились работать с базовыми командами Redis get и set. А ещё поняли, зачем разделять модели бизнес-логики и ответов API.