Тема-6. Backend

Что тут есть?
Основная информация: архитектура веб-приложения; AJAX; backend; API; модель MVC; WSGI и ASGI.
FastAPI: что это; отличия от Django и Flask; встроенная документация; эндпоинты и обработчики; Pydantic; валидация данных;

Архитектуры веб-приложений

Есть две работы веб-приложения:

- классическая модель;

- модель с использованием AJAX.

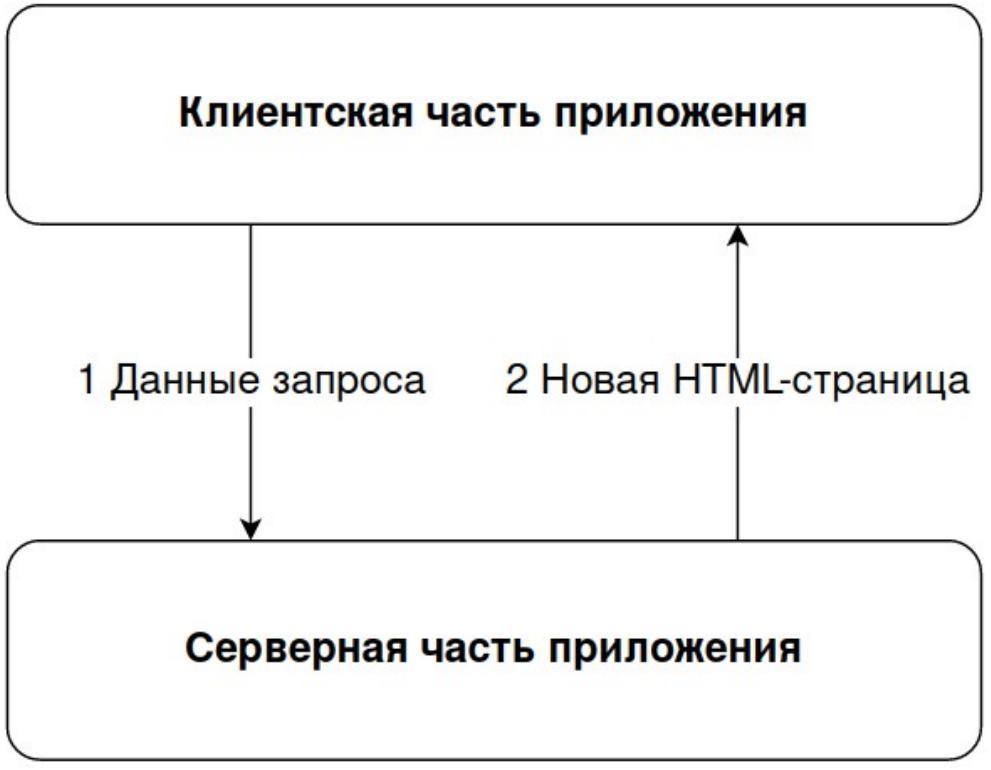
Архитектуры веб-приложений

Приложение, построенное по классической модели работает следующим образом:

1) пользователь делает запрос на сервер;

2) сервер формирует новую страницу и отдает её клиенту;

3) при последующих запросах также всякий раз создается новая страница для пользователя.



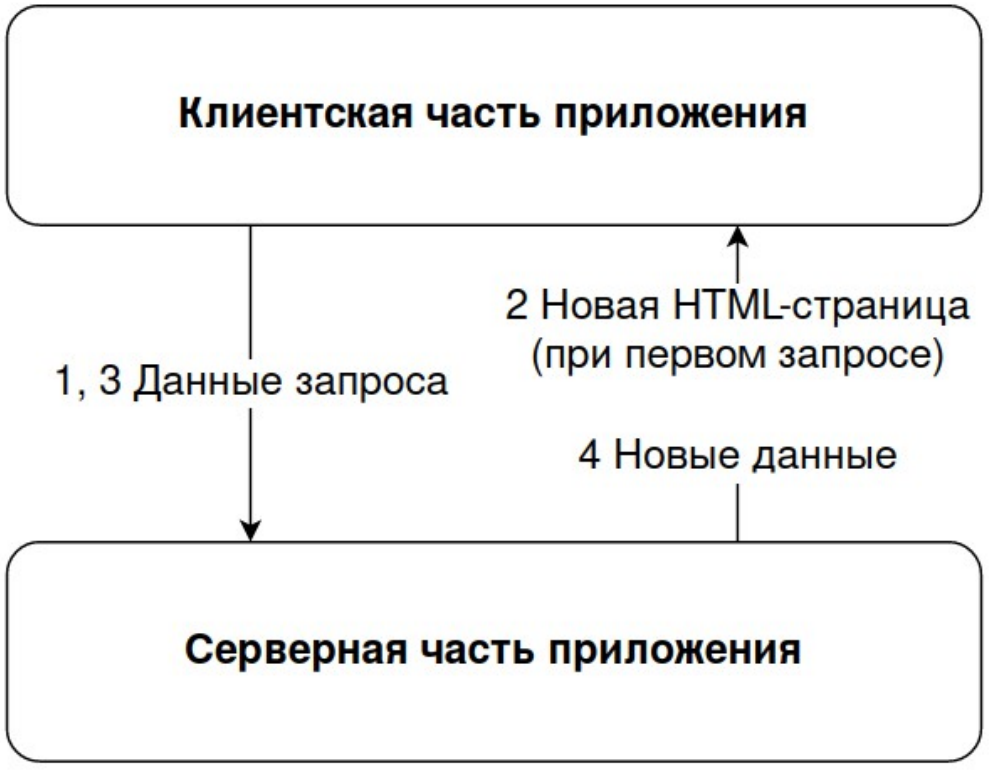
Архитектуры веб-приложений

Приложение, использующее AJAX работает следующим образом:

1) пользователь делает запрос на сервер;

2) сервер формирует новую страницу и отдает её клиенту;

3) при последующих запросах сервер отдает минимуму данных, необходимых для изменения страницы (4).



Об AJAX

AJAX (Asynchronous Javascript And Xml) – подход к построению веб-приложений, при котором возможен фоновый обмен данными между клиентом и сервером.

Фактически, формат XML, который присутствует в аббревиатуре, устарел, ему на смену пришел формат JSON – более простой и удобный.

Поэтому под AJAX будет иметься в виду обмен данными любого формата.

Плюсы AJAX

- Сервер не создает на каждый запрос пользователя новую страницу. Как следствие сайт работает быстрее, трафика расходуется меньше.

- Возможна интерактивная обработка данных. Например, пользователь что-то вводит и ему сразу дается подсказка подходящих вариантов.

- Если на странице запущено видео или музыка, то их можно не прерывать при каждом запросе.

Минусы AJAX

Контент, который генерируется с помощью JS не виден в поисковиках, поскольку страницы не содержат всю информацию изначально.

Нужно дважды проверять данные форм. Первый раз, чтобы сразу подсказать пользователю, если что-то не так при вводе, второй раз чтобы не допустить записи неверных данных (ведь запрос можно сэмулировать с терминала и обойти валидацию фронтенда).

Как AJAX используется

Чтобы использовать AJAX в веб-приложении, необходимо, чтобы:

- сервер предоставлял API для отправки запросов;

- на фронтенде использовался скрипт, который способен правильно обработать ответ сервера и перестроить страницу (обычно используется XMLHttpRequest (он работает и с JSON) или специальные библиотеки, например, axios).

Что такое API

Когда говорят о взаимодействии программ друг с другом и клиента с сервером в частности, говорят об API.

API (Application Programming Interface) – это описание схемы взаимодействия программ, или иначе, правила, по которым можно работать с определенной программой, например, с сервером приложения.

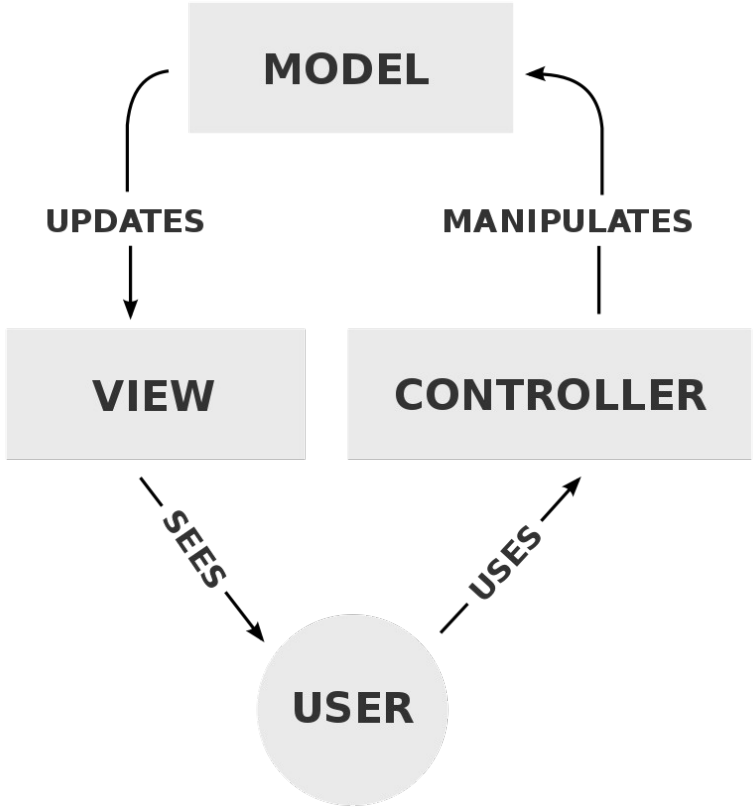
Что такое бекенд

Бекенд (Backend) – это часть веб-приложения, которая работает на сервере.

Бекенд отвечает за взаимодействие веб-приложения с другими сервисами: базой данных, другими серверами и т.д.

Стандартная модель бекенда

Как правило, бекенд состоит из трех частей:
- модель (model) – код, отвечающий за хранение данных;
- представление (view) – код, который отвечает за представление данных для фронтенда;
- контроллер (controller) – код, отвечающий за логику бекенда.



Модель бекенда

Модель – это описание формы хранения данных. Если используется реляционная база данных, то модель – описание таблиц БД и их связей.

Модель часто описывается не напрямую на SQL, а средствами языка, с помощью ORM. Например:

```
class Survey(Base, Timestamp):
    __tablename__ = 'survey'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("user.id", ondelete="CASCADE"))
    user = relationship("User", lazy="selectin")
    scheme_id = Column(Integer, ForeignKey("survey_scheme.id", ondelete="CASCADE"))
    scheme = relationship("SurveyScheme", lazy="selectin")
    data = Column(JSON, nullable=False)
```

Представление и контроллер

Представление – это часть бекенда, которая отвечает за передачу данных на фронтенд.

Контроллер – это часть бекенда, которая отвечает за изменение модели.

Все запросы (независимо от того, читают ли они данные или записывают) маршрутизируются на сервере и ведут на разные функции (обработчики), которые обрабатывают эти запросы. Маршруты, по которым получаются или записываются данные, называются эндпоинтами (конечная точка). Эндпоинты описываются строками, подобными URL.

API бекенда

Пример эндпоинта и обработчика:

```
@router.put("/pipeline/run")
def run_pipeline(
        data_files: Optional[List[UploadFile]] = None,
        data_sources: str = Form(...),
        node_tree: str = Form(...),
        session: Session = Depends(get_db),
        current_user: User = Depends(get_current_user)
):
    check_if_user_analyst(current_user, session)
    json_node_tree = json.loads(node_tree)
    data_sources_ = json.loads(data_sources)
    return run_pipeline(data_files, data_sources_, json_node_tree, session)
```

API бекенда

Эндпоинт – это иерархически организованный адрес. По нему можно понять, для какой части приложения он предназначен и что благодаря ему можно менять или запрашивать.

Например, по эндпоинту /api/v1/auth/user/1 можно понять, что он используется для работы с конкретным пользователем.

v1 в примере выше обозначает версию API. С развитием веб-приложение меняется и его API может устареть. Чтобы не ломать совместимость с другими сервисами, использующими API, принято создавать новую версию API, не удаляя старую.

API бекенда

Устройство эндпоинта обычно такое:
/api/версия/раздел/подраздел?/сущность?/номер-сущности?

Раздел – это какая-то отдельная часть приложения. Часто выделяется раздел auth (авторизации, регистрации и т.д.), store, если в приложении есть магазин, какой-то специфический раздел с основной функциональностью приложения (например: логика игры, API для запроса контента и т.д.)

Подразделы необязательны, но могут быть, если раздел большой.

API бекенда

Сущность часто ассоциируется с таблицей в БД, но может быть и не связана с ней. Но она характеризует какой-то конкретный объект, с которым можно работать.

Номер сущности указывается, если обработчик по этому эндпоинту работает с конкретным экземпляром, а не со всеми сразу.

Если запрос на чтение (GET), то в эндпоинте могут присутствовать параметры, как в URL: /api/v1/book/ru/df43kjnt?page=20

Фреймворк FastAPI

FastAPI – это фреймворк для написания логики бекенда. Он основан на фреймворке Starlette (который может использоваться и отдельно) и Pydantic (для сериализации данных).

Фреймворк FastAPI

Если сравнивать в Django и Flask, то FastAPI имеет следующие отличия:

- позволяет писать асинхронный код;

- работает по модели ASGI (об этом далее);

- FastAPI быстрее, чем Django и Flask.

Фреймворк FastAPI

Минимальный код приложения на FastAPI:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def data():
    return {"data": "some data"}
```

Фреймворк FastAPI

Пример команд для запуска бекенда на FastAPI (с помощью uvicorn):

С терминала:

```
uvicorn server:app --host 0.0.0.0 --port 5000
 --log-level info --reload --debug --reload-dir ./app
```

Из кода Python:

```
uvicorn.run("server:app", host="0.0.0.0", port=5000, log_level="info", reload=True, debug=True, reload_dirs=["app"])
```

Фреймворк FastAPI

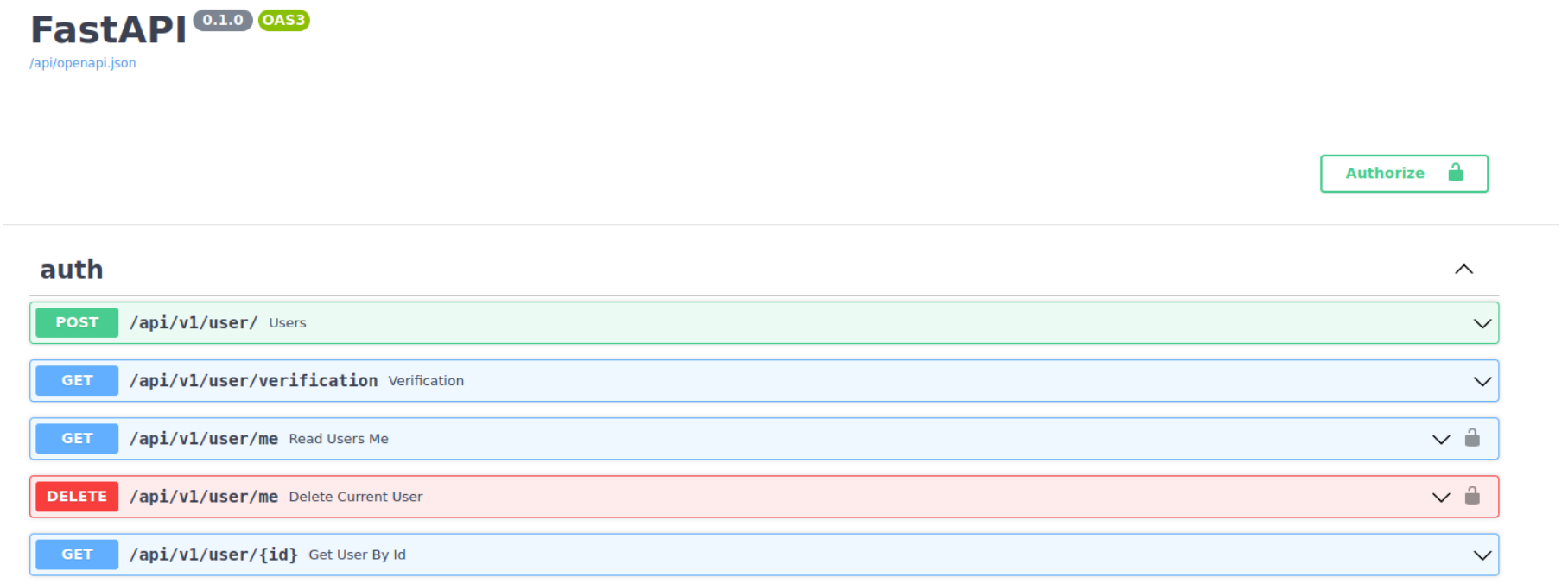
FastAPI автоматически генерирует документацию по API с помощью Swagger.

Посмотреть её можно обычно по адресу <server-address>/docs (или /api/docs)

Этот путь можно настроить при создании приложения:
app = FastAPI(docs_url="/api/docs")

Фреймворк FastAPI

Пример автоматически сгенерированной документации:



Фреймворк FastAPI

В FastAPI в эндпоинте можно указывать переменные значения:

```
@app.get("/product/{id}")
def get_product(id: int):
    product = get_product(id)
    return product
```

Фреймворк FastAPI

Если некоторый аргумент указан в обработчике, но его нет в пути эндпоинта (и он не передается из каких-либо кастомных декораторов), то он будет искаться в параметрах эндпоинта. Например, эндпоинт, по которому отправляется запрос: "/book/2?page=12"

Как получить параметры в обработчике:

```
@app.get("/book/{id}")
def get_book(id: int, page: int):
    # id = 2, page = 12
```

Фреймворк FastAPI

Для параметров добавлять несложные проверки с помощью функций Query (параметр после ?), Path (параметр в пути) и Body (параметр в теле запроса):

```
@app.get("/book/{id}")
def get_book(
    id: int = Path(..., gt=1),
    page: int = Query(None, gt=0, le=10000)
):
    # id = 2, page = 12
```

Фреймворк FastAPI

Параметры обработчика могут также зависеть от какой-либо функции. Это называется dependency injection.

```
from fastapi import Depends

@app.get("/book/{id}")
def get_book(
    id: int,
    page: int,
    user: User = Depends(get_current_user)
):
    pass
```

Фреймворк FastAPI

Сама функция при этом может принимать какие-то параметры, получая их из запроса и возвращать что-то новое в обработчик.

```
def get_current_user(token: str):
    pass  # get user entry by token
    return user
```

Фреймворк FastAPI

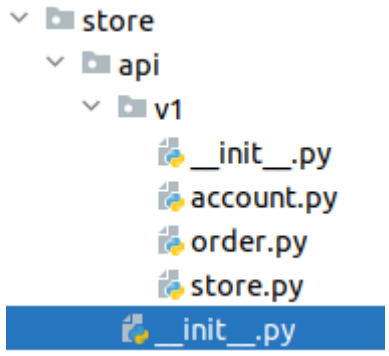
Эндпоинт не обязательно всегда писать полностью для обработчика. В FastAPI есть APIRouter, который позволяет иерархически организовать эндпоинты и соответствующие обработчики.

Предположим, у нас есть код:

```
from fastapi import FastAPI
from api import router

app = FastAPI()

app.include_router(api.router, prefix="/api")
```

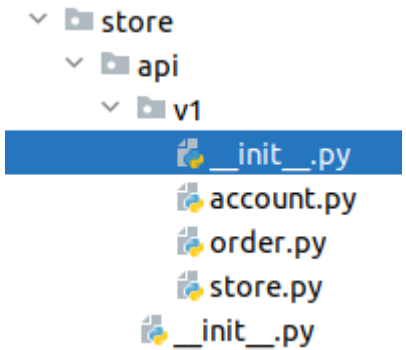
Фреймворк FastAPI

```
from fastapi import APIRouter
from . import v1

router = APIRouter()

router.include_router(v1.router, prefix="/v1", tags=["store"])
```



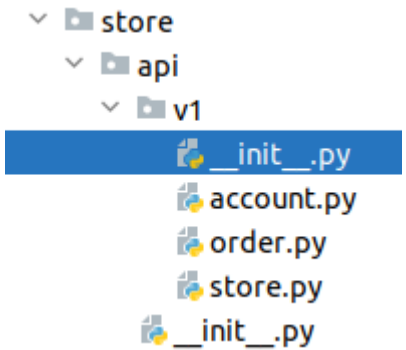
Фреймворк FastAPI

```
from fastapi import APIRouter
from .store import router as store_router
from .account import router as account_router
from .order import router as order_router


router = APIRouter()

router.include_router(store_router, prefix="/store")
router.include_router(account_router, prefix="/store/account")
router.include_router(order_router, prefix="/store/order")
```



Фреймворк FastAPI

```
from fastapi import APIRouter

router = APIRouter()

@router.get("/tag-list/")
def get_tag_list(user_input: str, lang: str = ""):
    pass

# Полный путь этого эндпоинта: /api/v1/store/tag-list.
```



Библиотека Pydantic

Иногда нужно пересылать данные более сложной структуры, чем просто строки или числа. Для этого в FastAPI используется бибилиотека Pydantic.

Она позволяет писать схемы данных (или модели), точно определяя структуру и типы данных.

Библиотека Pydantic

Пример создания схемы данных с помощью Pydantic:

```
from pydantic import Json, BaseModel
from datetime import date
from typing import List

class Config(BaseModel):
    config: Json
    name: str = "config.json"
    id: int
    users: List[str]
    date: date
```

Библиотека Pydantic

Схемы данных можно сразу валидировать с помощью Field (работает так же, как и Query):

```
from pydantic import BaseModel, Field

class Item(BaseModel):
    id: int = Field(..., gt=0)
```

Библиотека Pydantic

Можно делать валидацию и так:

```
from pydantic import BaseModel, validator

class Item(BaseModel):
    id: int

    @validator("id")
    def check_id(cls, v):
        if v < 1:
            raise ValueError("id must be a natural number")
        return v
```

Библиотека Pydantic

Использовать Pydantic модели можно для получения данных (например, чтобы получить информацию из POST-запроса):

```
@app.post("/config")
def set_config(config: Config):
    pass
```

Библиотека Pydantic

Использовать Pydantic модели можно и для передачи данных на фронтенд.

```
@app.get("/config", response_model=List[Config])
def get_configs():
    pass
```

WSGI & ASGI

WSGI (Web Server Gateway Interface) – это программа, которая запускает бекенд и обрабатывает входящие запросы с клиента.

По умолчанию эта программа работает синхронно. То есть пока обрабатывается один запрос, остальные ожидают.

С FastAPI используется модифицированная версия WSGI – ASGI (A – Asynchronous). То есть запросы обрабатываются асинхронно.

Полезные ссылки

- **https://fastapi.tiangolo.com/ – сайт FastAPI;**
- **https://pydantic-docs.helpmanual.io/ – сайт Pydantic.**

Следующая тема: SQLAlchemy & Alembic