

Лекция 9

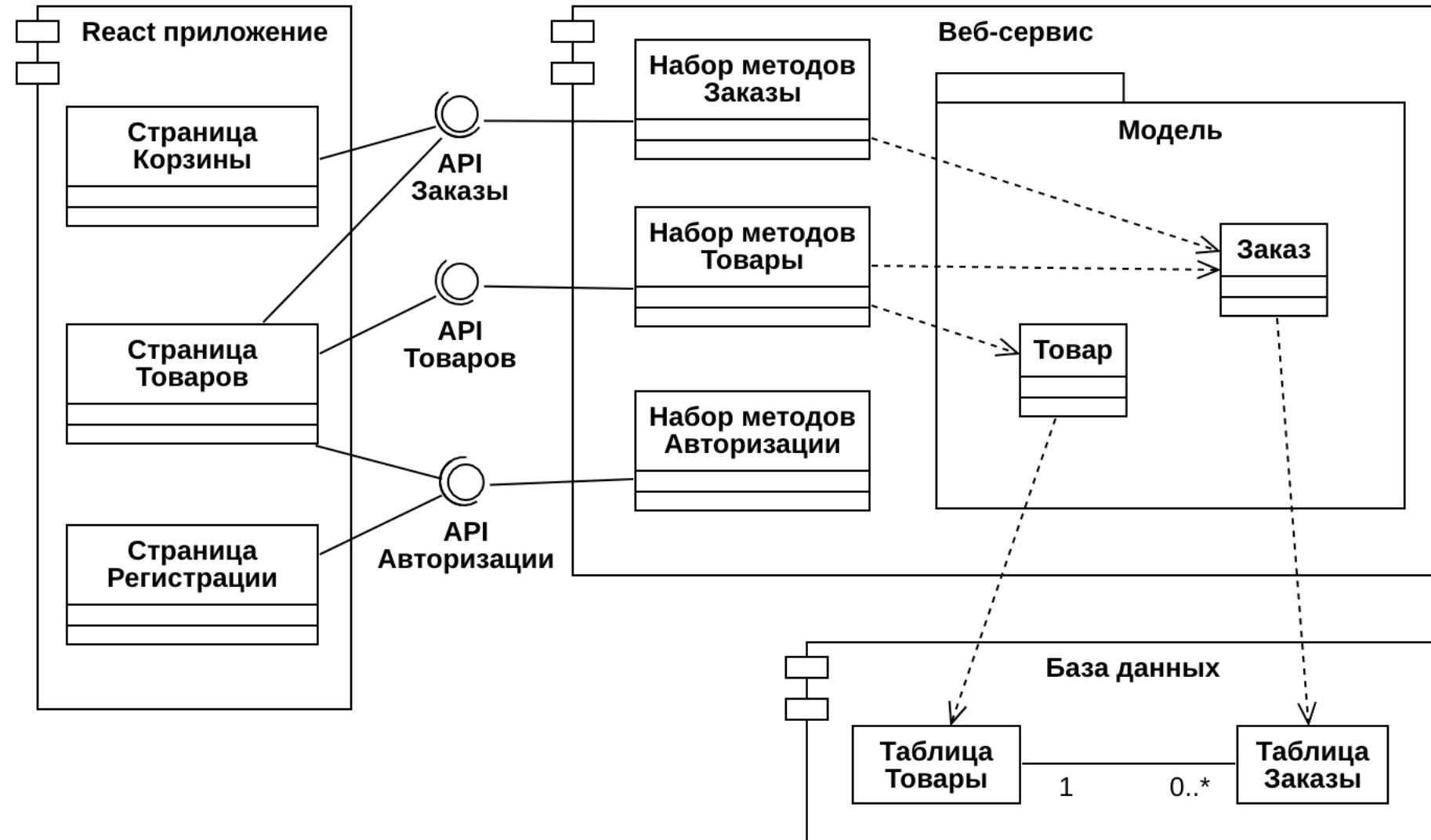
Архитектура фронтенда

Разработка интернет приложений

Канев Антон Игоревич

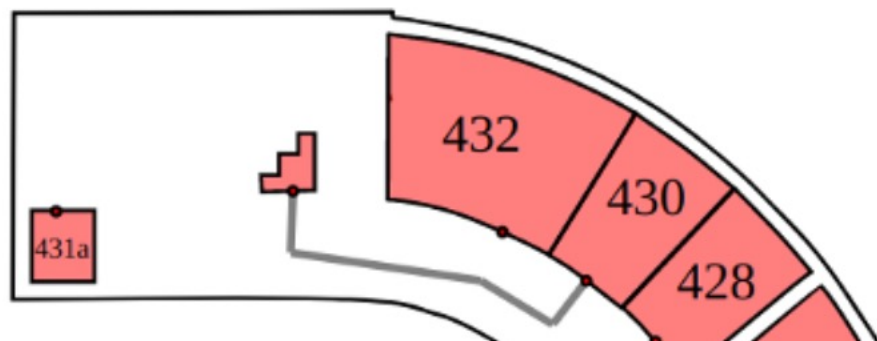
Классы фронта

- Мы хотим объединить бэкенд и фронтенд
- Но на 1 диаграмме все не поместиться
- Поэтому для 5 лабораторной оставляем ТОЛЬКО страницы React и домены веб-сервиса



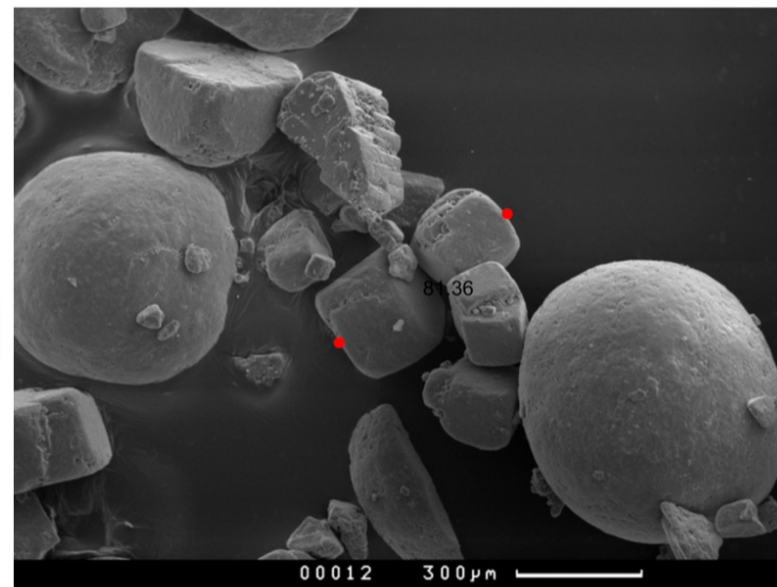
Фронтенд ИУ5

906.2 (аудитория) 430 (аудитория) Построить



<https://github.com/iu5git/CampusMap>

<https://github.com/iu5git/PhotoPointApp>

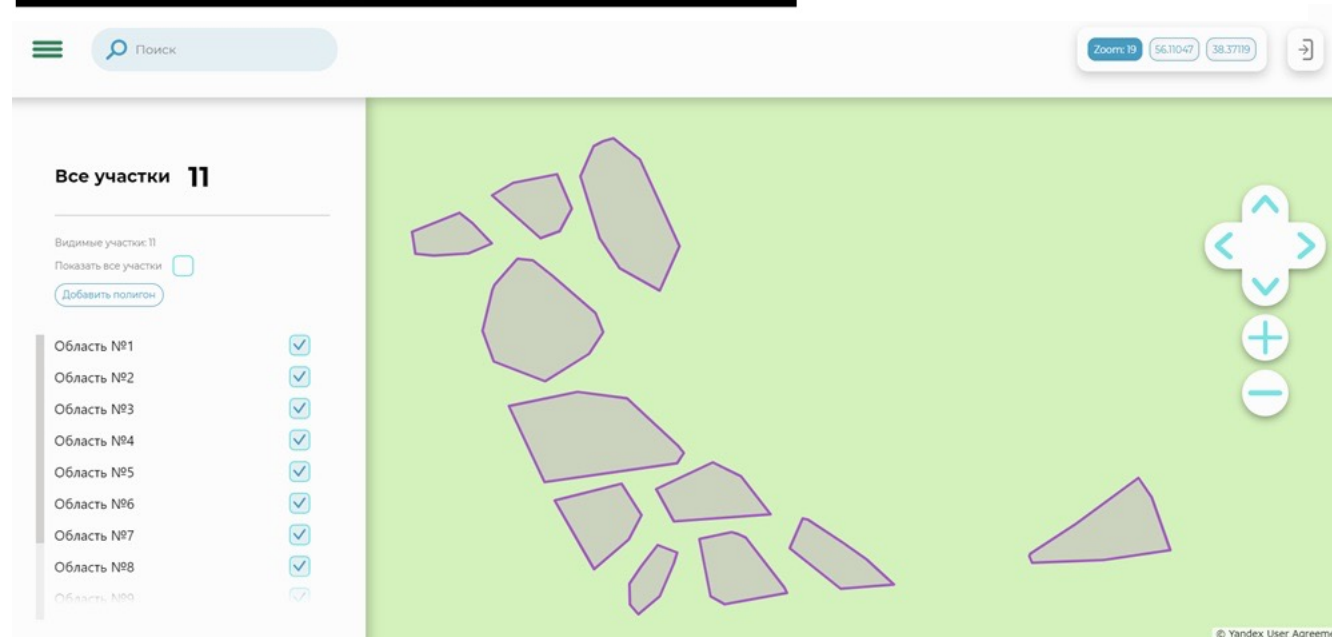


Загрузите изображение
1510061884164590658.jpg

Ширина изображения: 300 Единицы измерения: МКМ

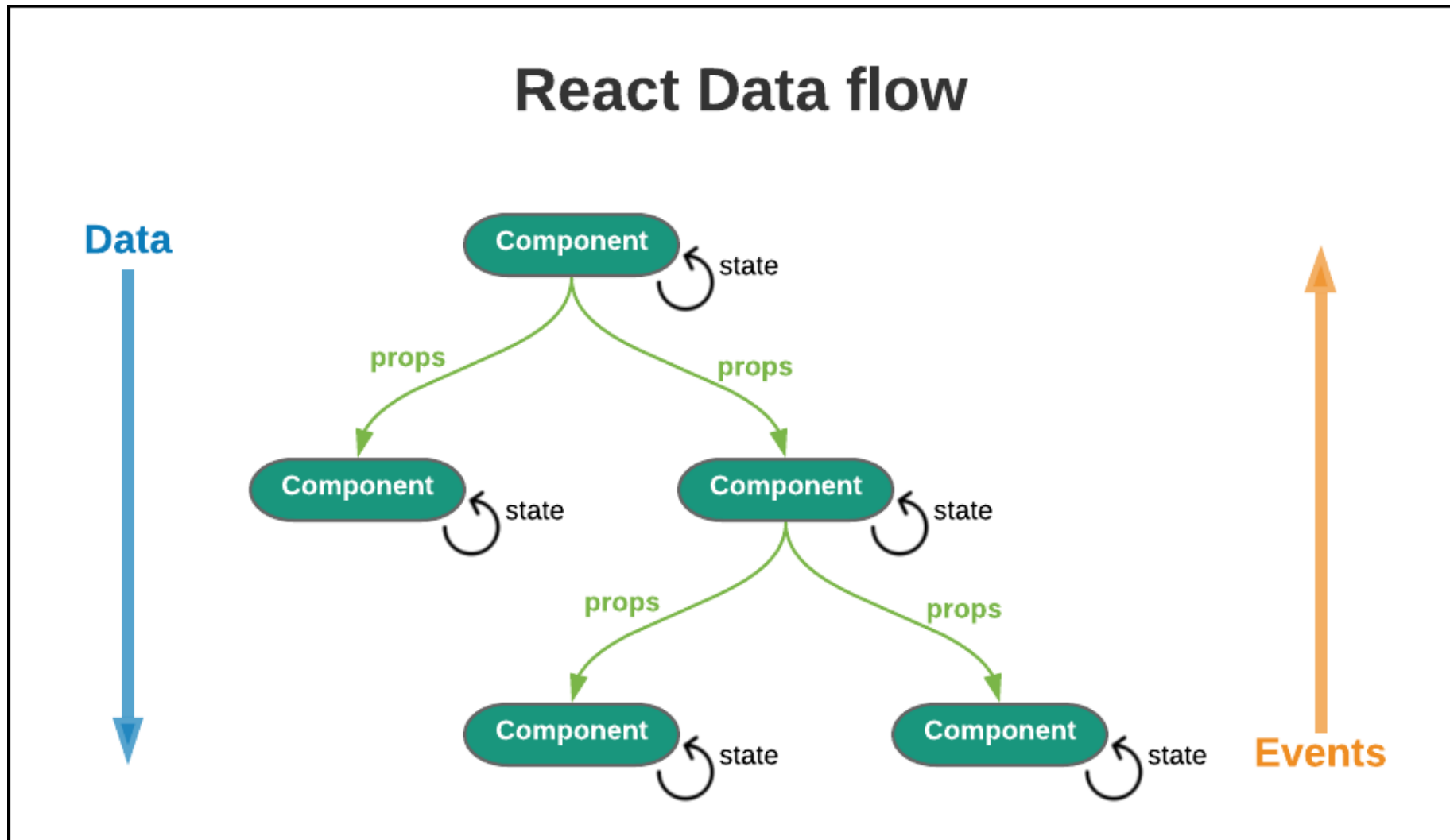
Текущее измерение: 81.36мкм Добавить

Начальная точка (x, y)	Конечная точка (x, y)	Расстояние
189.6, 138.0	125.0, 187.5	81.36 мкм

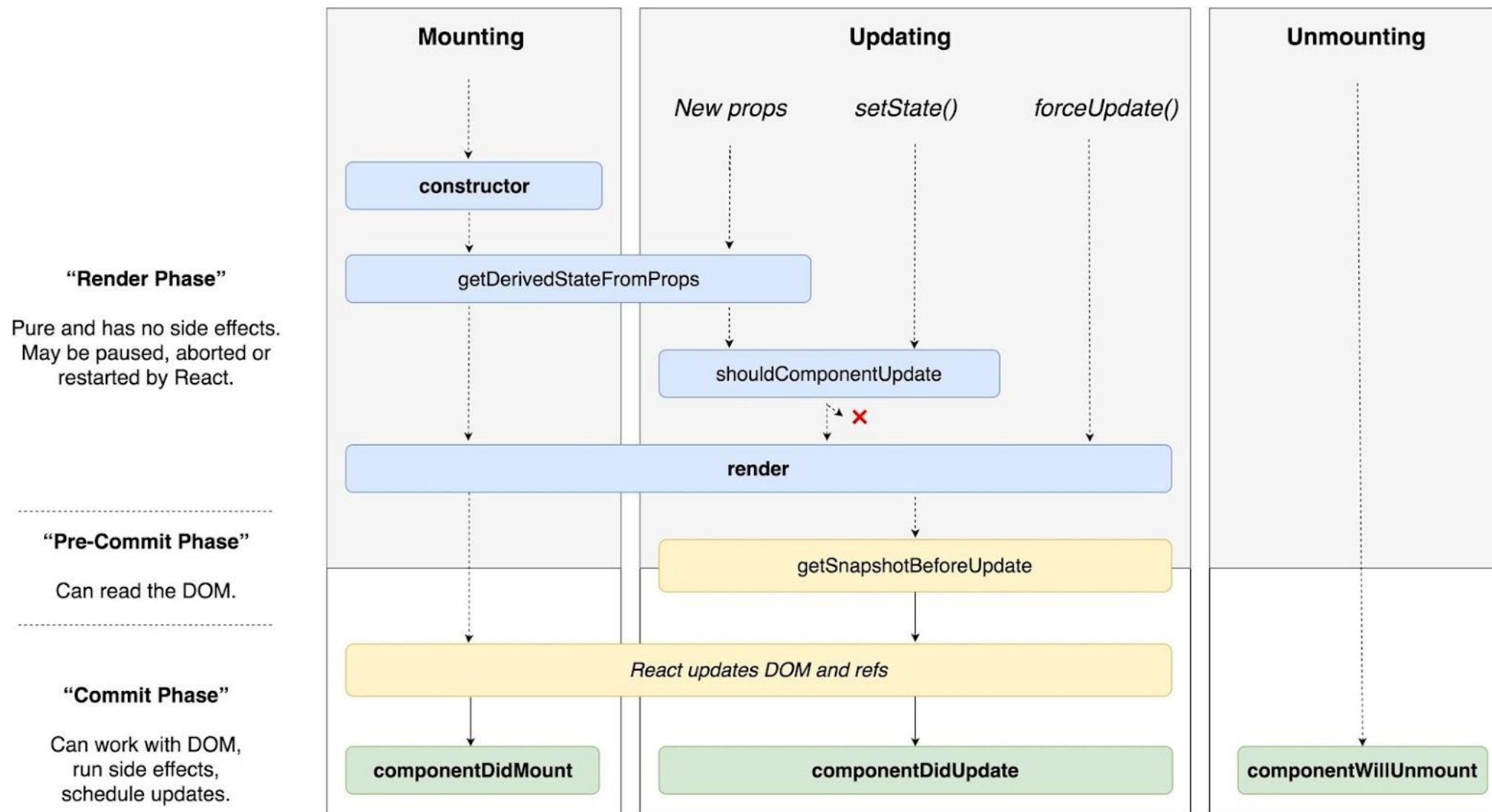


<https://github.com/iu5git/MapViewApp>

Поток данных и сообщений



Методы жизненного цикла компонента



Функциональные компоненты

- Описание компонентов с помощью чистых функций создает меньше кода, а значит его легче поддерживать.
- Чистые функции намного проще тестировать. Вы просто передаете props на вход и ожидаете какую то разметку.
- В будущем чистые функции будут выигрывать по скорости работы в сравнении с классами из-за отсутствия методов жизненного цикла
- Все это стало возможным благодаря хукам <https://react.dev/reference/react>

Хуки

```
import React, { useEffect, useState } from 'react'
import Axios from 'axios'
```

```
export default function Hello() {
```

```
  const [Name, setName] = useState('')
```

```
  useEffect(() => {
    Axios.get('/api/user/name')
      .then(response => {
        setName(response.data.name)
      })
  }, [])
```

```
  return (
    <div>
      My name is {Name}
    </div>
  )
}
```

```
import React, { Component } from 'react'
import Axios from 'axios'
```

```
export default class Hello extends Component {
```

```
  constructor(props) {
    super(props);
    this.state = { name: '' };
  }
```

```
  componentDidMount() {
    Axios.get('/api/user/name')
      .then(response => {
        this.setState({ name: response.data.name })
      })
  }
```

```
  render() {
    return (
      <div>
        My name is {this.state.name}
      </div>
    )
  }
```

Другие хуки

- **useContext**: позволяет работать с контекстом — с механизмом для организации совместного доступа к данным без передачи свойств.
- **useReducer**: усложняет useState добавляя разделение логики в зависимости от action. Вместе с useContext дают аналог Redux.

```
const ThemeContext = createContext(null);

export default function MyApp() {
  return (
    <ThemeContext.Provider value="dark">
      <Form />
    </ThemeContext.Provider>
  )
}
```

```
function Button({ children }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;
  return (
    <button className={className}>
      {children}
    </button>
  );
}
```


Другие хуки

- **useMemo**: используется для возврата мемоизированного значения. Может применяться, чтобы функция возвратила кешированное значение.
- Можно сохранить результаты вычислений между вызовами render

```
import { useMemo } from 'react';
```

```
function TodoList({ todos, tab, theme }) {  
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);  
  // ...  
}
```

React Router Hooks

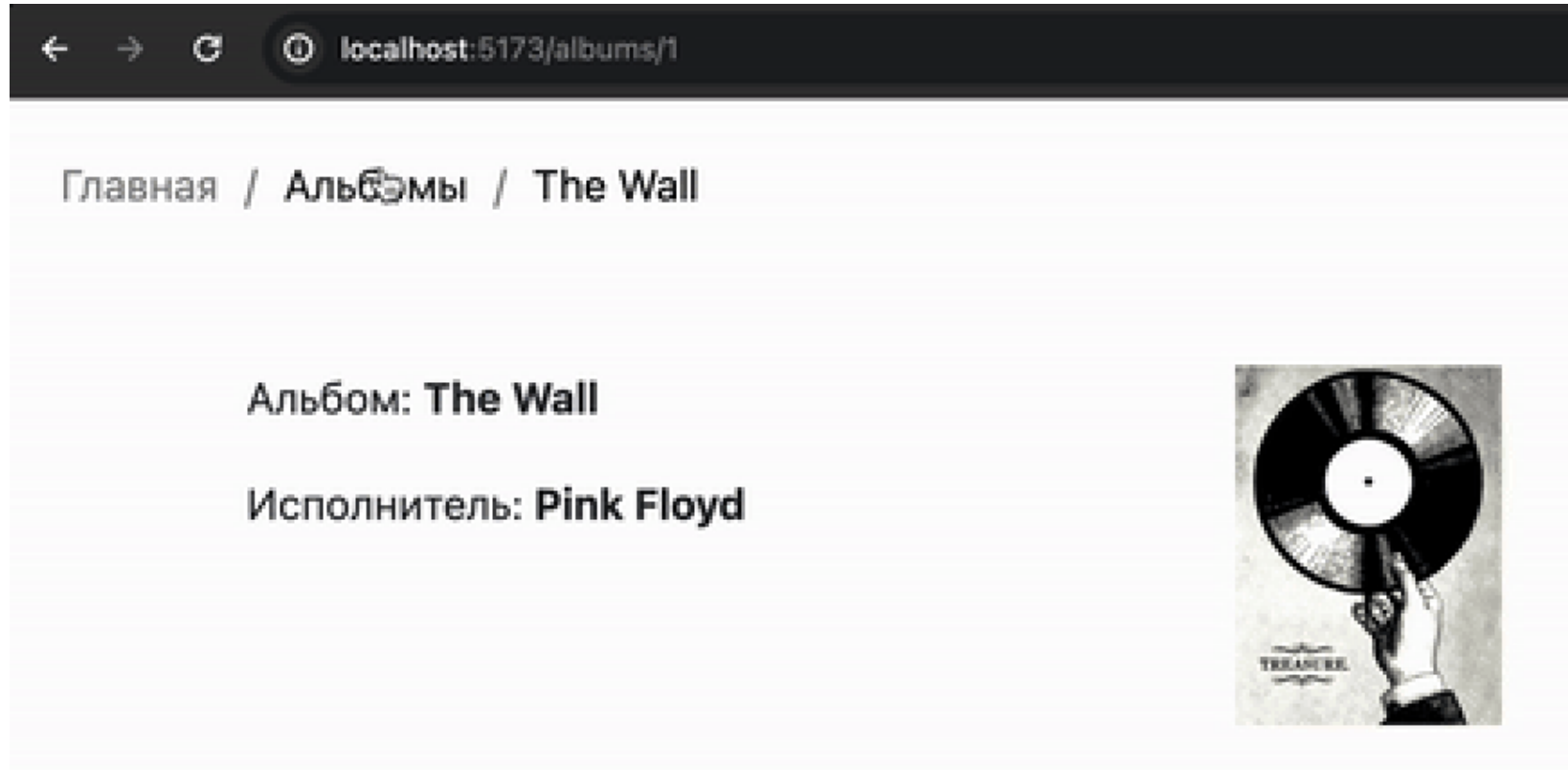
- **useLocation**: все данные о текущем пути url
- **useNavigate**: объект истории браузера
- **useParams**: параметры из url
- <https://reactrouter.com/>

```
1  import * as React from 'react';
2  import { Routes, Route, useParams } from 'react-router-dom';
3
4  function ProfilePage() {
5    // Get the userId param from the URL.
6    let { userId } = useParams();
7    // ...
8  }
9
10 function App() {
11   return (
12     <Routes>
13       <Route path="users">
14         <Route path=":userId" element={<ProfilePage />} />
15         <Route path="me" element={...} />
16       </Route>
17     </Routes>
18   );
19 }
```

Breadcrumbs

Для навигации пользователя по страницам нашего приложения будем использовать

- Самописные Хлебные крошки
- Навигационную панель (меню) из react-bootstrap



Breadcrumbs, поиск

- Создаем наш собственный компонент для реализации хлебных крошек
- В поиске ждем событие `onSubmit`, в котором используем состояние `searchValue`

```
return (  
  <div className="container">  
    <BreadCrumbs crumbs={[{ label: ROUTE_LABELS.ALBUMS }] } />  
  
    <InputField  
      value={searchValue}  
      setValue={(value) => setSearchValue(value)}  
      loading={loading}  
      onSubmit={handleSearch}  
    />  
  </div>  
)
```

```
export const BreadCrumbs: FC<BreadCrumbsProps> = (props) => {  
  const { crumbs } = props;  
  
  return (  
    <ul className="breadcrumbs">  
      <li>  
        <Link to={ROUTES.HOME}>Главная</Link>  
      </li>  
      {!!crumbs.length &&  
        crumbs.map((crumb, index) => (  
          <React.Fragment key={index}>  
            <li className="slash">/</li>  
            {index === crumbs.length - 1 ? (  
              <li>{crumb.label}</li>  
            ) : (  
              <li>  
                <Link to={crumb.path || ""}>{crumb.label}</Link>  
              </li>  
            )}  
          </React.Fragment>  
        ))}  
    </ul>  
  )
```

Mock

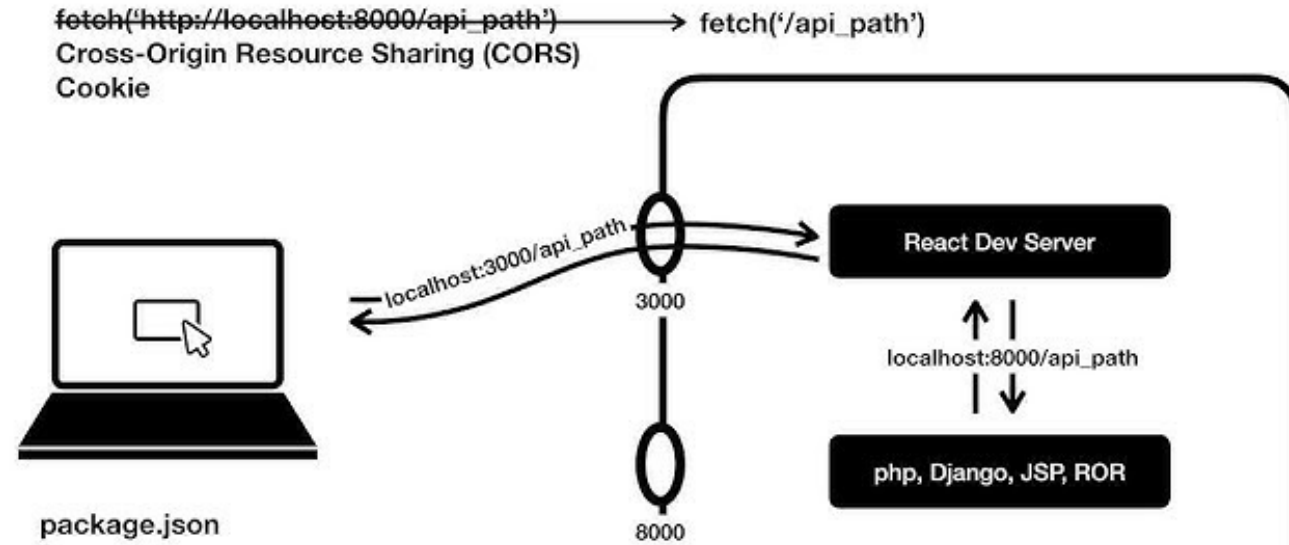
- В реальной жизни бывает очень сложно связать фронтенд и бэкенд вместе: что-то не готово, проблемы с развертыванием или Cors
- Поэтому бэкенд проверяем через Swagger/Postman, а для фронтенда готовим Mock

```
import { I iTunesResult } from "../getMusicByName";

export const SONGS MOCK: I iTunesResult = {
  resultCount: 3,
  results: [
    {
      wrapperType: "track",
      artistName: "Pink Floyd",
      collectionCensoredName: "The Wall",
      trackViewUrl: "",
      artworkUrl100: "",
    },
    {
      wrapperType: "track",
      artistName: "Queen",
      collectionCensoredName: "A Night At The Opera",
      trackViewUrl: "",
      artworkUrl100: "",
    },
    {
      wrapperType: "track",
      artistName: "AC/DC",
      collectionCensoredName: "Made in Heaven",
      trackViewUrl: "",
      artworkUrl100: "",
    },
  ],
};
```

Обратный прокси-сервер для CORS

- Одно из решений - отправляем запросы на напрямую в веб-сервис, а проксируем через наш сервер фронтенда
- Похоже на prod решение при проксировании через Nginx

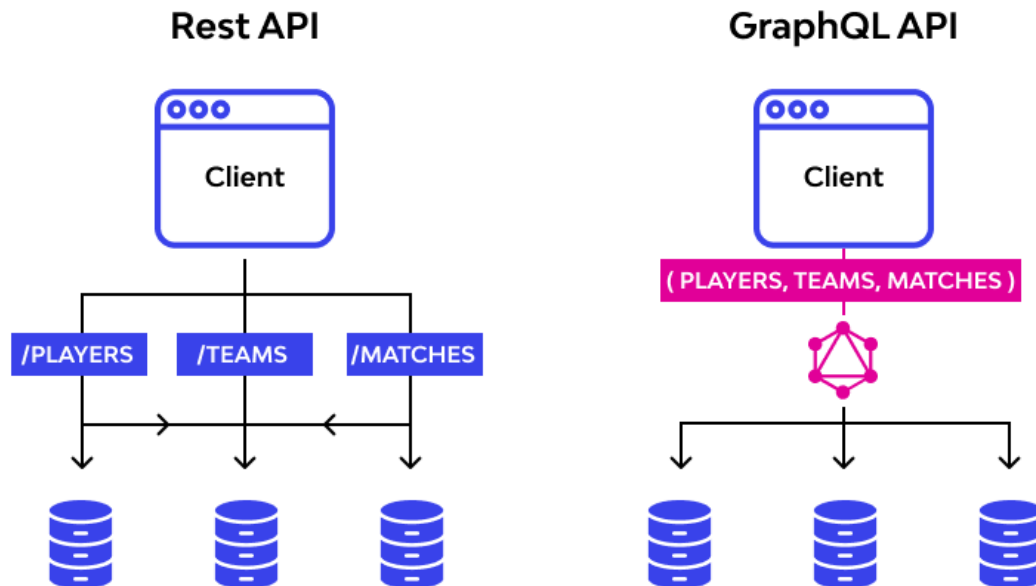
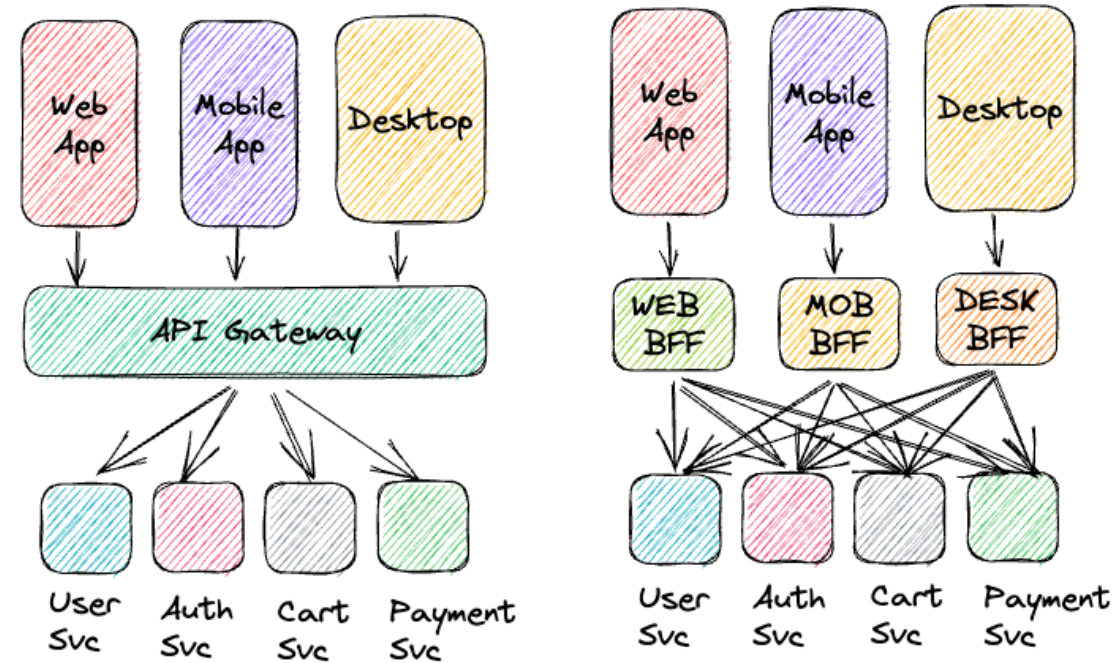


```
package.json
{
  ...
  proxy: 'http://localhost:8000',
  ...
}
```

```
export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      "/api": {
        target: "http://localhost:8080",
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, "/"),
      },
    },
  },
});
```

BFF и GraphQL

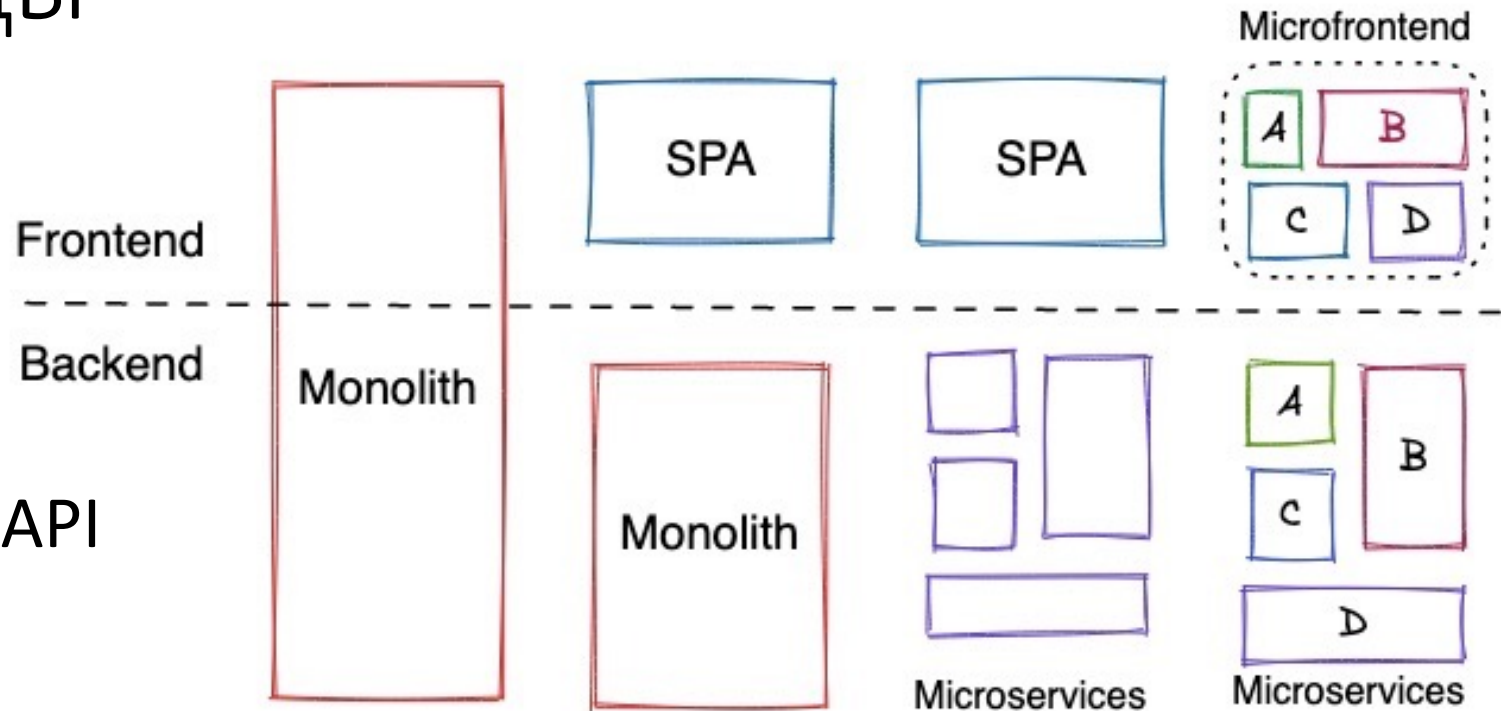
- GraphQL – язык запросов и сервер, который с открытым исходным кодом



- Backend for frontend – шлюз к нашим API, адаптированный под каждого из потребителей: веб-приложение, мобильное, десктоп

Микрофронтенды

- В одном интерфейсе несколько фронтенд фреймворков: разные команды фронтенд разработчиков, разные API
- Одни из способов уйти от зависимости от одной технологии и постепенно внедрять новую



Next.js

- SPA сильный инструмент, но как индексировать в поисковиках? На странице нет никаких данных
- Для этого используем серверные компоненты Next.js
- Хуков нет: данные передаем через props
- SSG (Static Site Generation) – HTML генерируется при **сборке** приложения

```
import type { InferGetStaticPropsType, GetStaticProps } from 'next'

type Repo = {
  name: string
  stargazers_count: number
}

export const getStaticProps = (async (context) => {
  const res = await fetch('https://api.github.com/repos/vercel/next.js')
  const repo = await res.json()
  return { props: { repo } }
}) satisfies GetStaticProps<{
  repo: Repo
}>

export default function Page({
  repo,
}: InferGetStaticPropsType<typeof getStaticProps>) {
  return repo.stargazers_count
}
```

FSD

Рассмотрим приложение социальной сети.

- app/ содержит настройку роутера, глобальные хранилища и стили.
- pages/ содержит компоненты роутов на каждую страницу в приложении, преимущественно композирующие, по возможности, без собственной логики.

В рамках этого приложения рассмотрим карточку поста в ленте новостей.

- widgets/ содержит "собранную" карточку поста, с содержимым и интерактивными кнопками, в которые вшиты запросы к бэкенду.
- features/ содержит всю интерактивность карточки (например, кнопку лайка) и логику обработки этой интерактивности.
- entities/ содержит скелет карточки со слотами под интерактивные элементы. Компонент, демонстрирующий автора поста, также находится в этой папке, но в другом слайсе.



Layers

Slices

Segments

- <https://feature-sliced.design/ru/docs/get-started/overview>

Архитектура FSD

Слои стандартизированы во всех проектах и расположены вертикально. Модули на одном слое могут взаимодействовать лишь с модулями, находящимися на слоях строго ниже. На данный момент слоев семь (снизу вверх):

1. shared — переиспользуемый код, не имеющий отношения к специфике приложения/бизнеса (например, UIKit, libs, API)
2. entities (сущности) — бизнес-сущности (например, User, Product, Order)
3. features (фичи) — взаимодействия с пользователем, действия, которые несут бизнес-ценность для пользователя (например, SendComment, AddToCart, UsersSearch)
4. widgets (виджеты) — композиционный слой для соединения сущностей и фич в самостоятельные блоки (например, IssuesList, UserProfile).
5. pages (страницы) — композиционный слой для сборки полноценных страниц из сущностей, фич и виджетов.
6. processes (процессы, устаревший слой) — сложные сценарии, покрывающие несколько страниц (например, авторизация)
7. app — настройки, стили и провайдеры для всего приложения.

Архитектура FSD

- Затем есть **слайсы**, разделяющие код по предметной области. Они группируют логически связанные модули, что облегчает навигацию по кодовой базе. Слайсы не могут использовать другие слайсы на том же слое, что обеспечивает высокий уровень *связности* (cohesion) при низком уровне *зацепления* (coupling).
- В свою очередь, каждый слайс состоит из **сегментов**. Это маленькие модули, главная задача которых — разделить код внутри слайса по техническому назначению. Самые распространенные сегменты — ui, model (store, actions), api и lib(utils/hooks), но в вашем слайсе может не быть каких-то сегментов, могут быть другие, по вашему усмотрению.

Преимущества FSD

Единообразие

- Код распределяется согласно области влияния (слой), предметной области (слайс) и техническому назначению (сегмент).
Благодаря этому архитектура стандартизируется и становится более простой для ознакомления.

Контролируемое переиспользование логики

- Каждый компонент архитектуры имеет свое назначение и предсказуемый список зависимостей.
Благодаря этому сохраняется баланс между соблюдением принципа **DRY** и возможностью адаптировать модуль под разные цели.

Устойчивость к изменениям и рефакторингу

- Один модуль не может использовать другой модуль, расположенный на том же слое или на слоях выше. Благодаря этому приложение можно изолированно модифицировать под новые требования без непредвиденных последствий.

Ориентированность на потребности бизнеса и пользователей



- Разбиение приложения по бизнес-доменам помогает глубже понимать, структурировать и находить фичи проекта.

Пример FSD

- В нашем GitLab доступен простой пример по FSD - рекомендуется для дипломной работы.
- В нем каждый слой состоит из слайсов, например, **Header**, **LoginPage** и так далее.

main ▾

react / src / shared / ui / Loader / Loader.tsx

 **Loader.tsx**  367 B

```
1 import { Box, CircularProgress } from '@mui/material';
2
3 import type { FC } from 'react';
4
5 export const Loader: FC = () => (
6   <Box
7     alignItems='center'
8     display='flex'
9     height='100vh'
10    justifyContent='center'
11    left='0'
12    position='fixed'
13    top='0'
14    width='100%'
15  >
16    <CircularProgress />
17  </Box>
18 );
```

Name

..

api

app

entities/user


features/Login


layouts/AuthorizedLayout


pages

shared

widgets/Header

 index.css

 index.tsx

 vite-env.d.ts

- <https://projects.iu5.bmstu.ru/iu5/infrastructure/departments-services/templates/react>