Лекция 7

Асинхронный JavaScript

Так в чем же заключается асинхронность?

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js');
alert('Ура, скрипт загрузился! (на самом деле нет)');

```

Строки кода в JS **не ожидают** окончания выполнения предыдущей

Строки порождают **задачи**, а результат ее выполнения может **занять много времени**

Но другие строки кода **в это время** порождают свои задачи

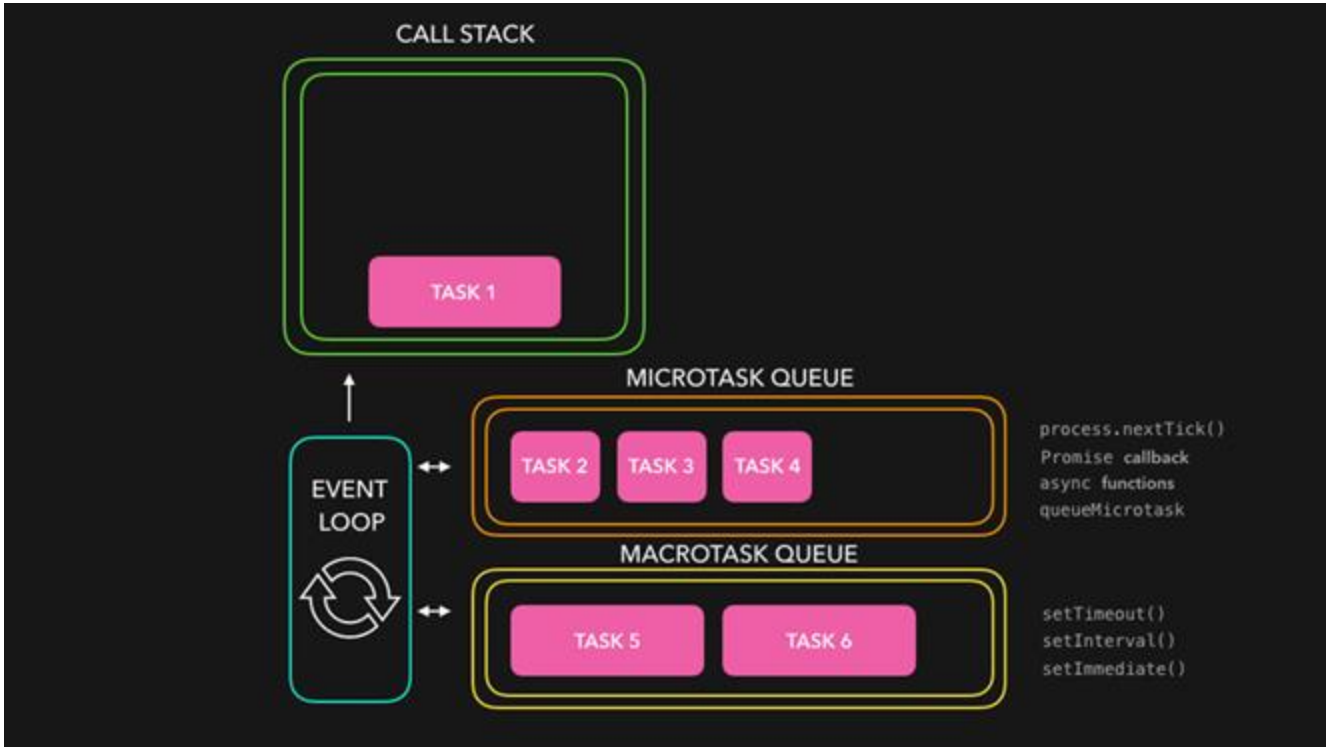
Более того, в результате выполнения каждой задачи могут **порождаться** еще задачи (внутри функции еще **будет** alert)

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
  alert(`Здорово, скрипт ${script.src} загрузился`);
  alert( _ ); // функция, объявленная в загруженном скрипте
});

```

Event Loop

JavaScript однопоточный и асинхронный

- **макротаски** - любые функции отложенного вызова (callback);
- **микротаски** - промисы и MutationObserver
- **рендеринг** - Paint, Layout (см устройство браузера), requestAnimationFrame, и тд.



Задачи могут попасть в **Call Stack** и выполниться только через **Event Loop**
Event Loop - это такой менеджер, определяет какой задаче когда выполняться.

Подробнее тут https://learn.javascript.ru/callbacks

Тут есть маленькая проблема – Callback hell

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...и так далее, пока все скрипты не будут заг
    });

  })

});
```

Код становится нечитаемым, слишком много отступов и высокий шанс ошибиться

Можно сделать рекурсивные вызовы, но это все равно неудобно и не решает всех проблем

```
function onLoadScript1(script) {
    // логика...
    loadScript(onLoadScript2);
}

function onLoadScript2(script) {
    // логика...
    loadScript(onLoadScript3);
}

function onLoadScript3(script) {
    // логика...
}

loadScript('/my/script.js', onLoadScript1);

```

Следующая проблема – обработка ошибок

В каком месте кода произошла ошибка?

И как об этом узнать в месте, где мы вызывали нашу функцию?

```
1 function loadScript(src, callback) {
2   let script = document.createElement('script');
3   script.src = src;
4   script.onload = () => callback(script);
5   document.head.append(script);
6 }
7
8 function onLoadScript(script) {
9   if (!script) {
10    throw new Error('Ай-ай-ай!');
11  }
12 }
13
14 try {
15  loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', onLoadScript);
16 } catch(err) {
17  // не попадем...
18 }
19
```

Что такое Promise?

Promise (обещание) - это обертка, которая позволяет нам использовать переменные, значения которых нам неизвестны на момент создания обещания.

По-сути Promise позволяет асинхронный код организовывать так, будто он синхронный.

- Java (`java.util.concurrent.Future`)
- C++ (`std::future`)
- C# (`System.Threading.Tasks`)
- …
- **JavaScript (`Promise`)**

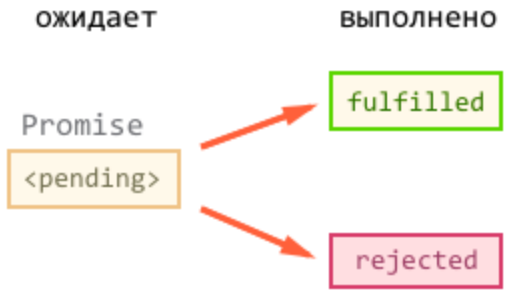
Как это выглядит в JavaScript. Callback Hell?

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...и так далее, пока все скрипты не будут загружены
    });

  })

});
```

Вариант с **promise** больше похож на привычный нам код в других языках

```
loadScript('/my/script.js')
  .then(() => loadScript('/my/script2.js'))
  .then(() => loadScript('/my/script3.js'))
  .then(() => {});

```

Promise в JavaScript



Promises (промисы) — это специальные объекты, которые могут находиться в одном из трёх состояний:

- вначале **pending** («ожидание»)
- затем либо **fulfilled** («выполнено успешно»)
- либо **rejected** («выполнено с ошибкой»)

К промис можно привязать два **коллбека**:

- onFulfilled — срабатывают, когда promise «выполнен успешно»
- onRejected — срабатывают, когда promise «выполнен с ошибкой»

```
1 const promise = new Promise(function(resolve, reject) {
2     // do smth
3     resolve('success'); // or
4     // reject(new Error('failure'));
5 });
6
7 promise
8     .then(res => console.log(res))
9     .catch(err => console.error(err));
10
```

Варианты статусов и обработчиков

```
const promise = new Promise(function(resolve, reject) {
    // Здесь можно выполнять любые действия

    // вызов resolve(result) переведёт промис в состояние fulfilled
    // вызов reject(error) переведёт промис в состояние rejected
});

// Можно создать сразу "готовый" промис
const fulfilled = Promise.resolve(result);
// const fulfilled = new Promise((resolve, _) => resolve(result));
const rejected = Promise.reject(error);
// const rejected = new Promise((_, reject) => reject(error));

```

Мы будем постоянно создавать промисы, для нас важны **resolve** и **reject**, которые меняют состояние промиса

```
const promise = new Promise( ... );

// Можно навесить их одновременно
promise.then(onFulfilled, onRejected);

// Можно по отдельности
// Только обработчик onFulfilled
promise.then(onFulfilled);
// Только обработчик onRejected
promise.then(null, onRejected);
promise.catch(onRejected); // Или так

```

onFulfilled, onRejected – это тот код, который выполнится в соответствующем варианте промиса

Мы их указываем в блоках then, catch (**не путать** с try)

Обработка асинхронных ошибок в цепочке

В начале происходит ошибка, поэтому мы ищем далее обработчик onRejected (внутри catch), а все then пропускаем

```
1  // 'value 1', 'Error!', 'Error catched!'
2  const promise = Promise.resolve('value 1');
3
4  promise
5      .then(res => { console.log(res); throw 'Error!'; })              // 1
6      .then(res => { console.log('foo'); })
7      .then(res => { console.log('bar'); })
8      .then(res => { console.log('baz'); })
9      .catch(err => { console.error(err); return 'Error catched!'; }) // 2
10     .then(res => { console.log(res); });                             // 3
11
```

Переделаем наш пример на промисы

```
function loadScript(src, callback) {
  return new Promise((resolve) => {
    const script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(callback(script));
    script.onerror = () => reject(new Error('Ай-яй-яй!'));

    document.head.append(script);
  });
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js')
  .then((script) => {alert(`Здорово, скрипт ${script.src} загрузился`)})
  .catch((error) => {alert(`Плохо дело, ошибка ${error}`)});

```

Какие еще варианты с Promise есть?

- Promise.all
- Promise.race (самый быстрый)
- Promise.any (вернет первый fulfilled)
- Promise.allSettled

```
1 // Делаем что-нибудь асинхронное и важное параллельно
2 Promise.all([
3       PromiseGet('/user/1'),
4       PromiseGet('/user/2'),
5 ]).then(function(users) {
6       // Результатом станет массив из значений всех промисов
7       users.forEach(function(user, i) {
8           console.log(`User #${i}: ${value}`);
9       });
10 });
11
```

Async/Await

Async функции всегда возвращают promise

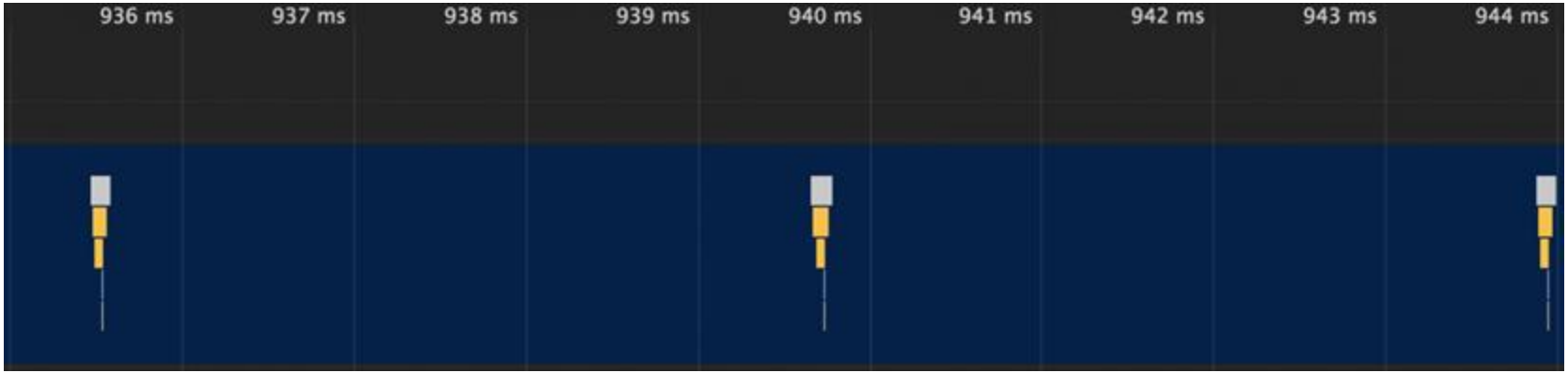
```
async function f1() {
    return 1;
}
async function f2() {
    return Promise.resolve(1);
}
f1().then(console.log) // 1
f2().then(console.log) // 1

```

```
async function f() {
  let p = new Promise((resolve)=> setTimeout(()=>resolve('done'), 1000))
  let result = await p; // будет ждать 1сек
    console.log(result)
}
// await нельзя использовать в обычных функциях

```

Что произойдет?

Макротаска в 4мс

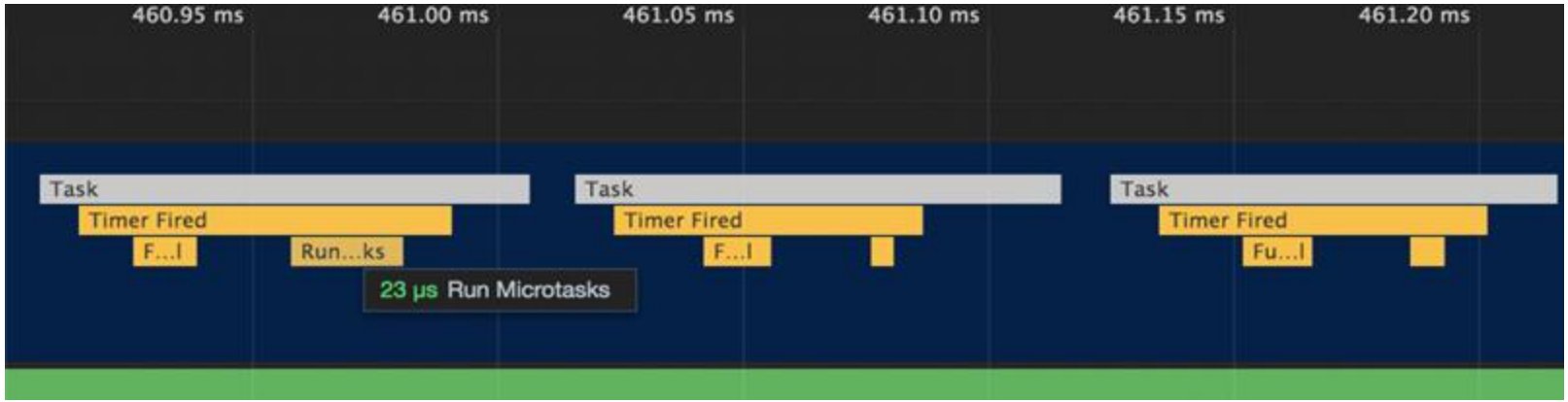
```
1 function foo() {
2   setTimeout(foo, 0);
3 }
4
5 foo();
6
```



Что произойдет?

```
1 function foo() {
2   Promise.resolve().then(foo);
3 }
4
5 foo();
6
```

Микротаски друг за другом



Задача с собеседований

Сначала выполняются строки кода

Потом промисы из микрозадач

В конце коллбэки из макрозадач

```
this is the start
this is just a message
this is the end
Promise.resolve1
Promise.resolve2
Promise.resolve3
setTimeout1
setTimeout2

```

```
(function() {

  console.log('this is the start');

  setTimeout(function cb() {
    console.log('setTimeout1');
  });

  console.log('this is just a message');

  Promise.resolve().then(() => {
    console.log('Promise.resolve1');
  })

  setTimeout(function cb1() {
    console.log('setTimeout2');
  }, 0);

  Promise.resolve()
    .then(() => {
    console.log('Promise.resolve2');
  }).then(() => {
    console.log('Promise.resolve3');
  })

  console.log('this is the end');
})();
```

Fetch API

Метод `fetch` — это **XMLHttpRequest** нового поколения

Улучшенный интерфейс для осуществления запросов к серверу: как по части возможностей и контроля над происходящим, так и по синтаксису, так как построен на **промисах**

```
const getDataFromServer = async () => {
    try {
        //Делаем GET запрос на указанный урл
        const result = await fetch('yandex.ru');

        // возвращаем результат в случае успеха
        return result;
    } catch (e) {
        console.log(e);
    }
```

Fetch API options

```
1 // Синтаксис метода fetch:
2 const fetchPromise = fetch(url[, options]);
3
```

- `method` — метод запроса
- `headers` — заголовки запроса (объект)
- `body` — тело запроса: FormData, Blob, строка и т.п.

```
1 fetch('/courses', {
2     method: 'POST',
3     mode: 'cors',
4     credentials: 'include',
5     body: JSON.stringify({
6         title: 'ПСП',
7         authors: ['Толпаров Натан', 'Алехин Сергей']
8     })
9 });
10
```

Браузерная поддержка

JavaScript был создан в 1996 году

В 1997 году Ecma International предложила стандартизировать JavaScript, и в результате появился ECMAScript

ECMAScript содержит правила, сведения и рекомендации, которые должны соблюдаться скриптовым языком, чтобы он считался совместимым с ECMAScript

JavaScript - скриптовый язык общего назначения, соответствующий спецификации ECMAScript

Подробнее тут https://habr.com/ru/companies/nix/articles/342904/

Подробнее тут https://frontend.tech-mail.ru/slides/s8/

JavaScript-движок

Программа или интерпретатор, способный понимать и выполнять JavaScript-код

JavaScript-движки обычно используются в веб-браузерах, включая V8 в Chrome, SpiderMonkey в Firefox и Chakra в Edge. Каждый движок подобен языковому модулю, который позволяет приложению поддерживать определенное подмножество языка JavaScript.

Среда выполнения – в ней JavaScript-код выполняется и интерпретируется JavaScript-движком:

- На клиентской стороне средой выполнения JavaScript будет веб-браузер, в котором становятся доступными для манипуляций такие хост-объекты, как окна и HTML-документы.

- На серверной стороне среда выполнения JavaScript — это Node.js.

Процесс TC39

TC39 (технический комитет 39) — занимается развитием **JavaScript**. Его членами являются компании (помимо прочих, все основные производители браузеров)

Процесс TC39 — алгоритм внесения изменений в спецификацию ECMAScript. Каждое предложение по добавлению новой возможности в ECMAScript в процессе созревания проходит ряд этапов

0 этап: идея (strawman)
1 этап: предложение (proposal)
2 этап: черновик (draft)
3 этап: кандидат (candidate)
4 этап: финал (finished)

ECMAScript (как выглядит)

- Декабрь 1999 — ECMAScript 3
- ECMAScript 4 (**abandoned**) — заброшенная версия
- Декабрь 2009 — ECMAScript 5
- Июнь 2011 — ECMAScript 5.1
- **Июль 2015 — ECMAScript 2015 (ECMAScript 6th edition)**
- Июль 2016 — ECMAScript 2016 (ECMAScript 7th edition)
- Июнь 2017 — ECMAScript 2017 (ECMAScript 8th edition)
- *Лето 2018* — ECMAScript 2018 (и так далее)

ES.Next — так временно называют совокупность новых возможностей языка, которые могут войти в следующую версию спецификации.

Фичи из ES.Next правильнее называть “предложения” (proposals) , потому что они всё ещё находятся на стадии обсуждения

Возможности браузера

Как нам использовать последний функционал JavaScript и поддерживать все браузеры?
Совместимы ли все браузеры с версией ECMAScript на котором мы пишем наш код?

Лучше перевести наш современный код в **старую версию!**

http://caniuse.com/

Транспайлинг и Babel

Транспайлинг — это конвертация кода программы, написанной на одном языке программирования в другой язык программирования

Тут показан пример перевода кода из ES6 (ES2015) в старый ES5

Babel — многофункциональный транспайлер, позволяет транспиллировать ES5, ES6, ES2016, ES2017, ES2018, ES.Next, JSX и Flow

```
// before
const f = num => `${num} в квадрате это ${num ** 2}`;

// after
var f = function (num) {
    return num + ' в квадрате это ' + Math.pow(num, 2);
};
```

```
# устанавливаем модуль
$ npm install -D @babel/core @babel/cli @babel/preset-env

# файл с конфигурацией
$ nano .babelrc
{
    "presets": [[
        "@babel/env",
        { targets: {edge: "15"}}
    ]]
}

# запускаем
$ babel modern.js --watch --out-file compatible.js
```

Bundler

Bundler — программа, которая упаковывает сложный проект со многими файлами и внешними зависимостями в один (иногда несколько) файл, который будет отправлен браузеру.

Подробнее тут https://habr.com/ru/companies/vk/articles/340922/

```
<head>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.4/moment.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/1.4.0/axios.min.js"></script>
  <script src="index.js"></script>
  <script src="auth.js"></script>
  <script src="reg.js"></script>
</head>
```

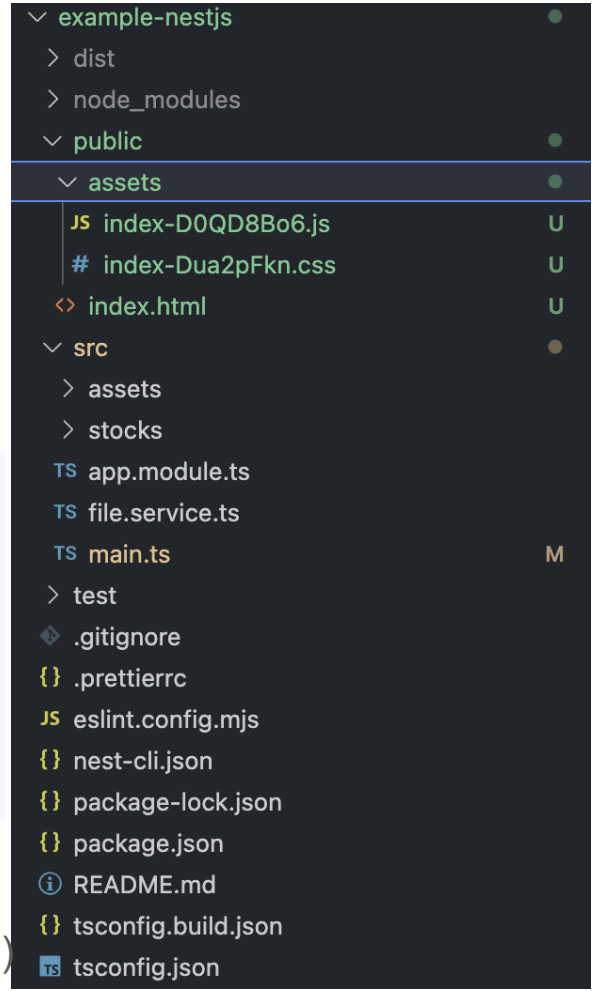
```
<head>
  <script src="bundle.js"></script>
</head>
```

Vite

- В файле vite.config.js указываем папку, в которую соберем наш bundle (итоговые файлы js и css)
- Выполняем npm run dev

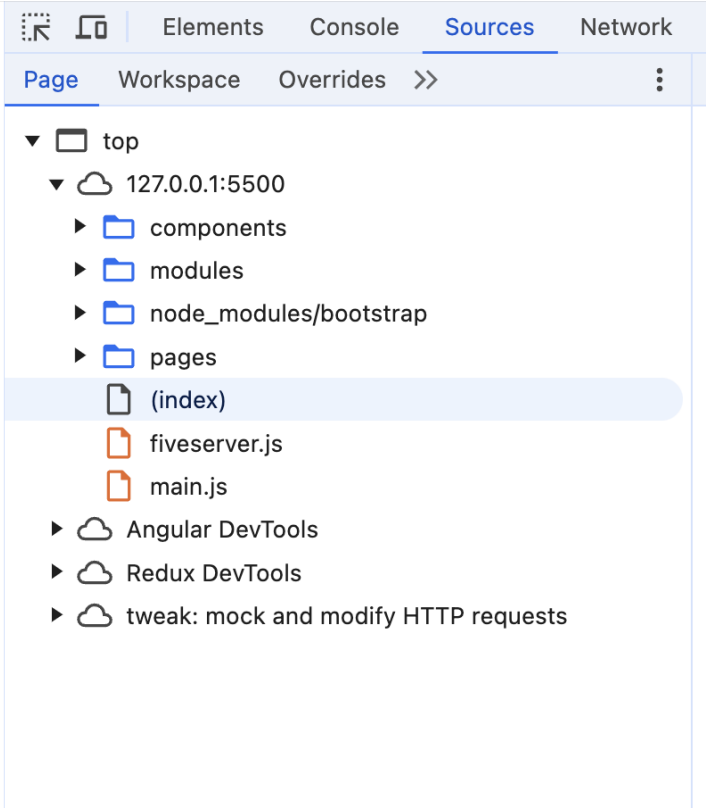
Эту операцию делаем в репозитории нашего фронтенда из 6 лабораторной

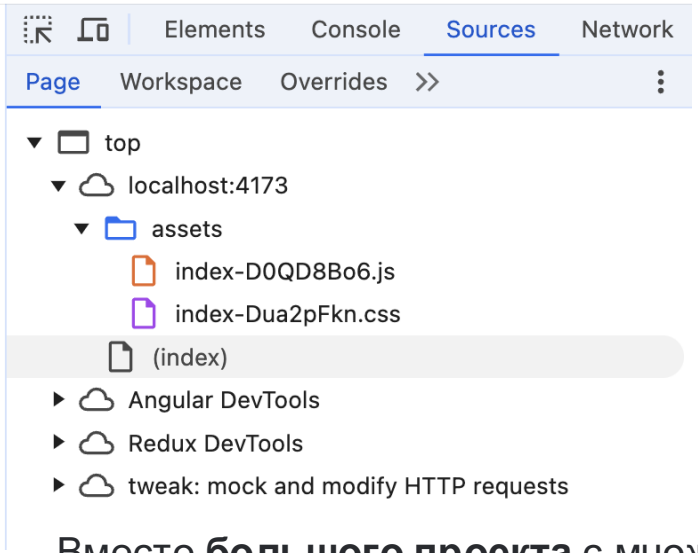
```
// vite.config.js
export default {
    build: {
        outDir: './public',
        emptyOutDir: true,
    },
};
```



- После этого папку public с нашим bundle мы копируем в проект нашего бэкенда (файлы исходного кода нам не нужны)

Применение сборки bundle через Vite





Вместо **большого проекта** с множеством файлов мы получили **один файл** js и один css, который и будет выполняться в браузере

Более того, теперь у нас нет проблемы CORS, так как страницу из bundle мы получаем от сервера бэкенда