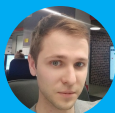




# PROMISES, ASYNC/AWAIT, TIMERS & EVENT LOOP



АЛЕКСАНДР РУСАНОВ



# АЛЕКСАНДР РУСАКОВ

Веб-разработчик



[a\\_s\\_rusakov@mail.ru](mailto:a_s_rusakov@mail.ru)



[@a\\_s\\_rusakov](https://t.me/as_rusakov)



# ПЛАН ЗАНЯТИЯ

1. [Длительные операции](#)
2. [Event Loop](#)
3. [setTimeout\(callback, 0\)](#)
4. [Web Workers](#)
5. [Callback Hell](#)
6. [Promises](#)
7. [async/await](#)
8. [Тестирование асинхронного кода](#)



# ДЛИТЕЛЬНЫЕ ОПЕРАЦИИ

# ДЛИТЕЛЬНЫЕ ОПЕРАЦИИ

Давайте рассмотрим пример обработки "тяжёлых" операций в JS:

```
1  let sum = 0;  
2  for (let i = 0; i < 1e11; i++) {  
3      sum += i;  
4  }
```

Если загрузить в браузер страницу с таким скриптом, то мы увидим, что страница не реагирует на нажатия кнопок. Наступит так называемый freeze (замораживание экрана).

Давайте разбираться, в чём причина.



# МНОГОПОТОЧНОСТЬ

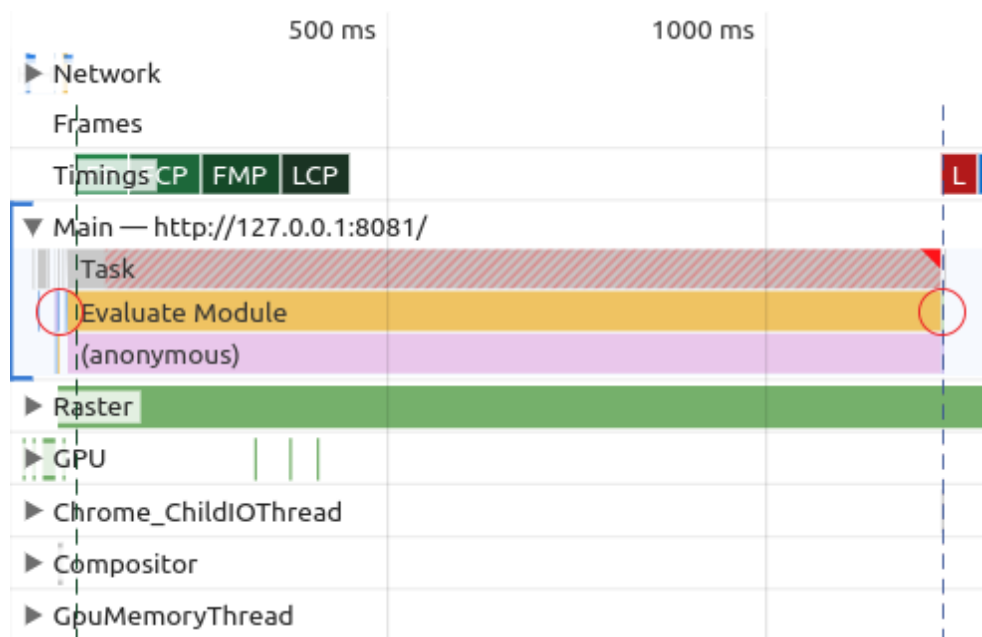
Многие языки программирования предлагают инструменты для создания и управления несколькими потоками выполнения.

Традиционно, этот раздел считается одним из самых сложных и подверженных ошибкам.

JS исполняет приложения в одном потоке\* (упрощённый взгляд), т.е. может выполнять одну операцию в единицу времени.

# МНОГОПОТОЧНОСТЬ

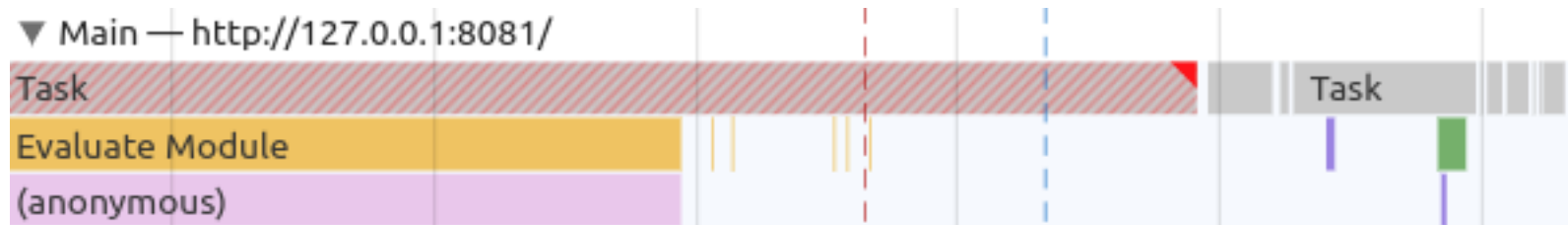
Загрузим нашу страницу в DevTools: вкладка Performance -> Start profiling and reload page (желательно это делать в режиме инкогнито с отключенными расширениями):



Оранжевым цветом обозначено исполнение нашего скрипта. А красным выделены области, которые можно приблизить (клавиши WASD) и посмотреть, что в них происходит.

# МНОГОПОТОЧНОСТЬ

Приблизим ту часть, что справа:



То, что идёт после "Evaluate Module":

- оранжевые полосы - события `load`, `DOMContentLoaded` и т.д.
- фиолетовая и зелёные полосы - отрисовка UI

И вот здесь самое интересное: получается, что пока браузер выполняет какой-то скрипт, то отрисовка и обработка событий - не возможны.





# EVENT LOOP



# EVENT LOOP

Event Loop - цикл обработки событий. В целом, его можно представлять себе так:



## EVENT LOOP

Что тут происходит: когда браузер исполняет какой-то скрипт, он исполняет его целиком (т.е. выполняет все вызовы функций), не отвлекаясь на необходимость отрисовки UI или обработку событий.

Как только обработка скрипта завершена, браузеру "нечего" делать, поэтому он начинает выбирать задачи из очереди (назовём её `macrotasks`).

В этой очереди стоят очереди на отрисовку экрана и вызов `callback`'ов.

---

# CALLBACKS

Callbacks - подход, при котором вместо ожидания какого-либо события (например, завершения операции) либо обработки какого-то элемента, мы передаём функцию (callback), которую нужно выполнить после наступления этого события, либо для обработки этого элемента.

Например, делая HTTP-запрос или навешивая Listener на элемент, мы передаём функцию-callback, которая будет вызвана тогда, когда произойдёт событие.

Это позволяет нам писать асинхронный код в однопоточном режиме: всё попадает в очередь задач и "вытаскивается" оттуда после того, как браузер завершил исполнять текущую задачу.

# CALLBACKS

Важно: понятие callback используется не только в контексте асинхронности. Callback является функцией, передаваемой в качестве аргумента другой функции, для вызова внутри этой функции. Для Built-in объектов это выполнение каких-либо операций (например, для Array - поиск, сравнение и т.д.).

Среда, в которой будет исполняться ваш JS-код (будь это браузер или Node.js) сама берёт на себя заботу по вызову вашего callback'а в нужный момент времени.

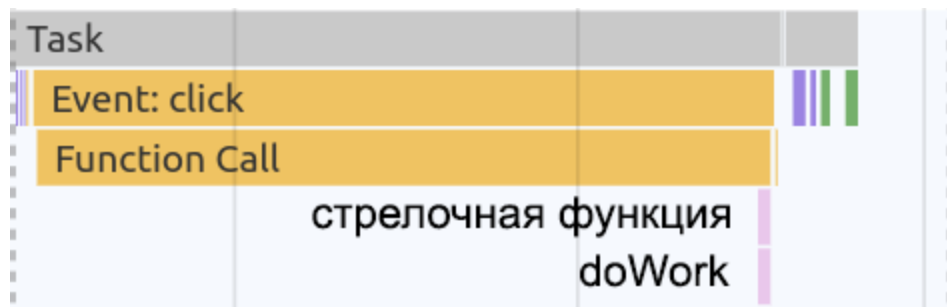
# EVENTLISTENER

Давайте посмотрим на примере Event Listener'a:

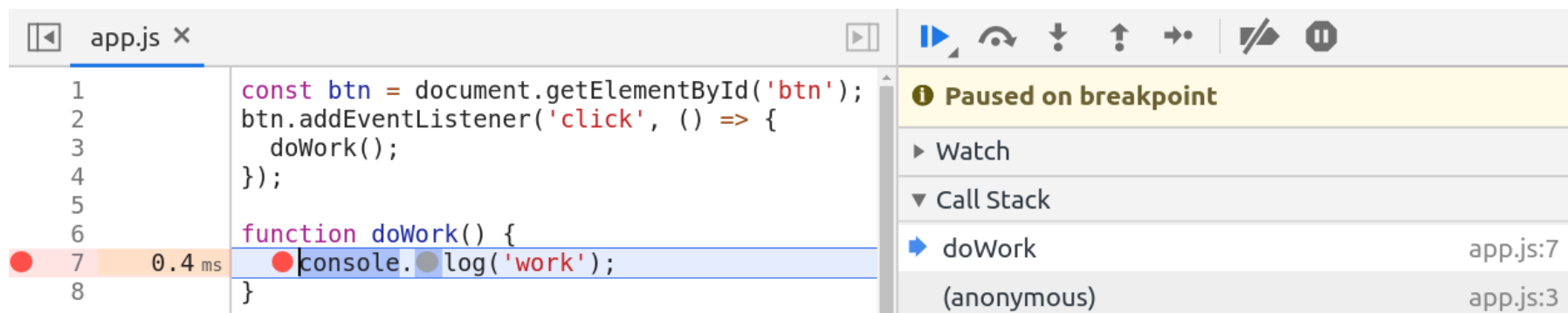
```
1  const btn = document.getElementById('btn');
2  btn.addEventListener('click', () => {
3    doWork();
4  });
5
6  function doWork() {
7    console.log('work');
8  }
```

И поставим на профайлинг, только не при загрузке страницы, а с уже загруженной страницей.

# EVENTLISTENER



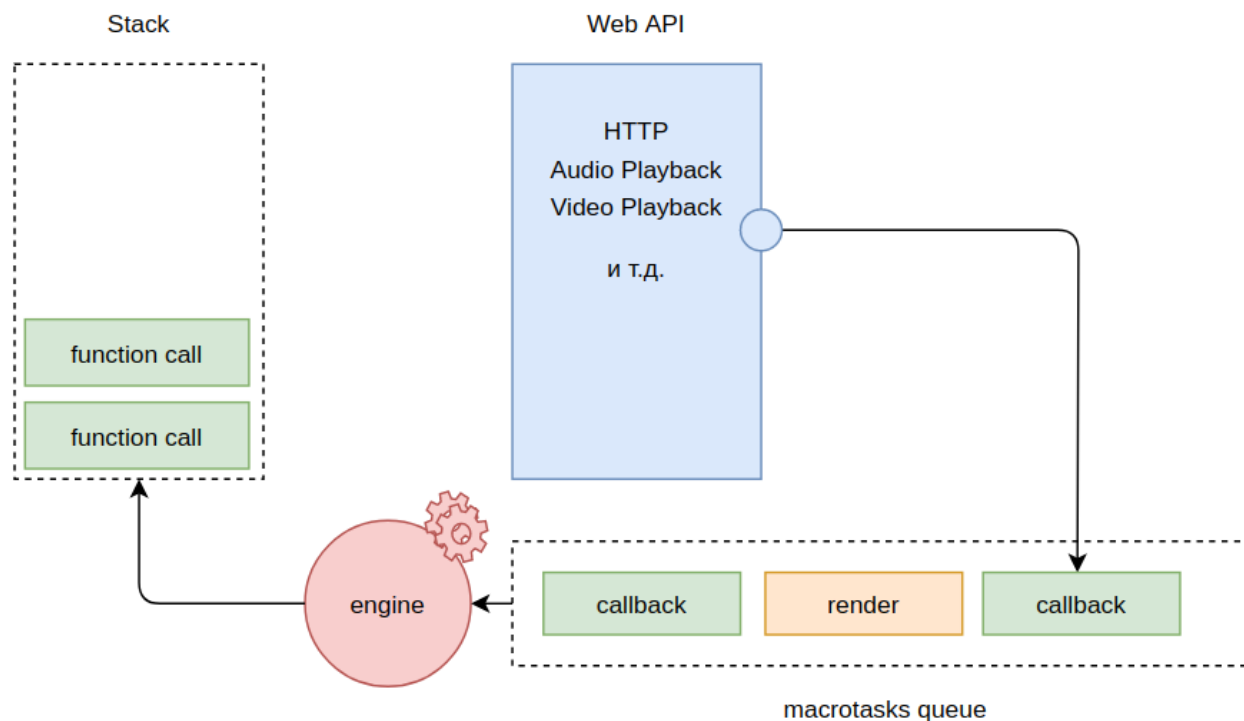
Когда функция вызывает одна другую - то они организуются в структуру, которая называется стек вызовов:



Соответственно, при вызове функция кладётся в стек, а после завершения работы, снимается со стека. Если функция вызывает другую функцию - то другая функция кладётся в стек и выполняется до завершения.

# EVENT LOOP

Таким образом, Event Loop на самом деле, выглядит следующим образом:

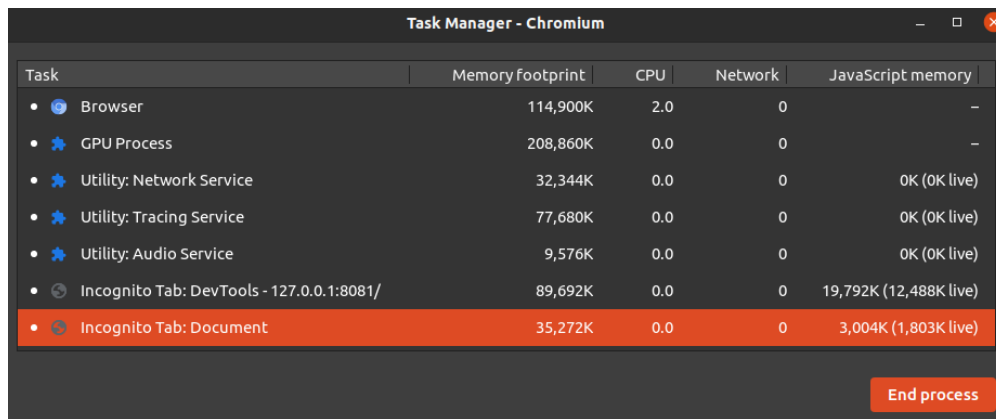


Выполняет задачу, пока стек не опустеет  
После чего берёт следующую задачу из очереди



# WEB API

Причём реализации Web API, например, сетевые запросы, работают в отдельном потоке или даже процессе:



The screenshot shows the 'Task Manager - Chromium' window. It contains a table with the following columns: Task, Memory footprint, CPU, Network, and JavaScript memory. The 'Incognito Tab: Document' process is highlighted in red.

Task	Memory footprint	CPU	Network	JavaScript memory
• Browser	114,900K	2.0	0	–
• GPU Process	208,860K	0.0	0	–
• Utility: Network Service	32,344K	0.0	0	0K (0K live)
• Utility: Tracing Service	77,680K	0.0	0	0K (0K live)
• Utility: Audio Service	9,576K	0.0	0	0K (0K live)
• Incognito Tab: DevTools - 127.0.0.1:8081/	89,692K	0.0	0	19,792K (12,488K live)
• Incognito Tab: Document	35,272K	0.0	0	3,004K (1,803K live)

End process

Поэтому вызовы Web API и не "подвешивают" наше приложение. Яркий пример: внедрите на страницу тег `audio` и поставьте на воспроизведение. После чего в callback'е запустите тяжёлый скрипт: UI зависнет, но воспроизведение аудио не прекратится.



# AUDIO

Демонстрация примера с Audio.



**SETTIMEOUT(CALLBACK, 0)**

# ДЛИТЕЛЬНЫЕ ОПЕРАЦИИ

Это всё здорово, но никак не решает нашу проблему с зависанием UI при тяжёлых операциях.

Но есть трюк, который позволит этого избежать:

```
setTimeout(callback, 0).
```

Почему это работает? Потому что вызов `setTimeout` ставит задачу в очередь, позволяя выполниться отрисовке UI (даже несмотря на то, что последний аргумент равен 0).

```
1  const btn = document.getElementById('btn');
2  btn.addEventListener('click', () => {
3    setTimeout(() => {
4      sum = 0;
5      current = 0;
6      doWork();
7    }, 0);
8  });
9
10 let sum = 0;
11 const start = 0;
12 const finish = 1e11;
13 let current = 0;
14
15 function doWork() {
16   let sum = current;
17   for (let i = 0; i < 1e6 && current < finish; i++) {
18     sum += i;
19     current++;
20   }
21   if (current === finish) {
22     console.log(current);
23     return;
24   }
25   setTimeout(doWork, 0);
26 }
```

# SETTIMEOUT

Такой подход усложняет код, но зато оставляет ваш интерфейс  
ОТЗЫВЧИВЫМ:





# WEB WORKERS



# WEB WORKERS

Альтернативный подход заключается в использовании [Web Workers](#).

[Web Workers](#) - API, позволяющие исполнять скрипты в фоновом режиме независимо от скриптов, работающих с пользовательским интерфейсом.



# ОГРАНИЧЕНИЯ

Web Worker'ы не имеют доступа к `window`, `document` и DOM, но при этом имеют доступ к:

- таймаутам и интервалам;
- XHR/fetch;
- возможности создавать другие Worker'ы.

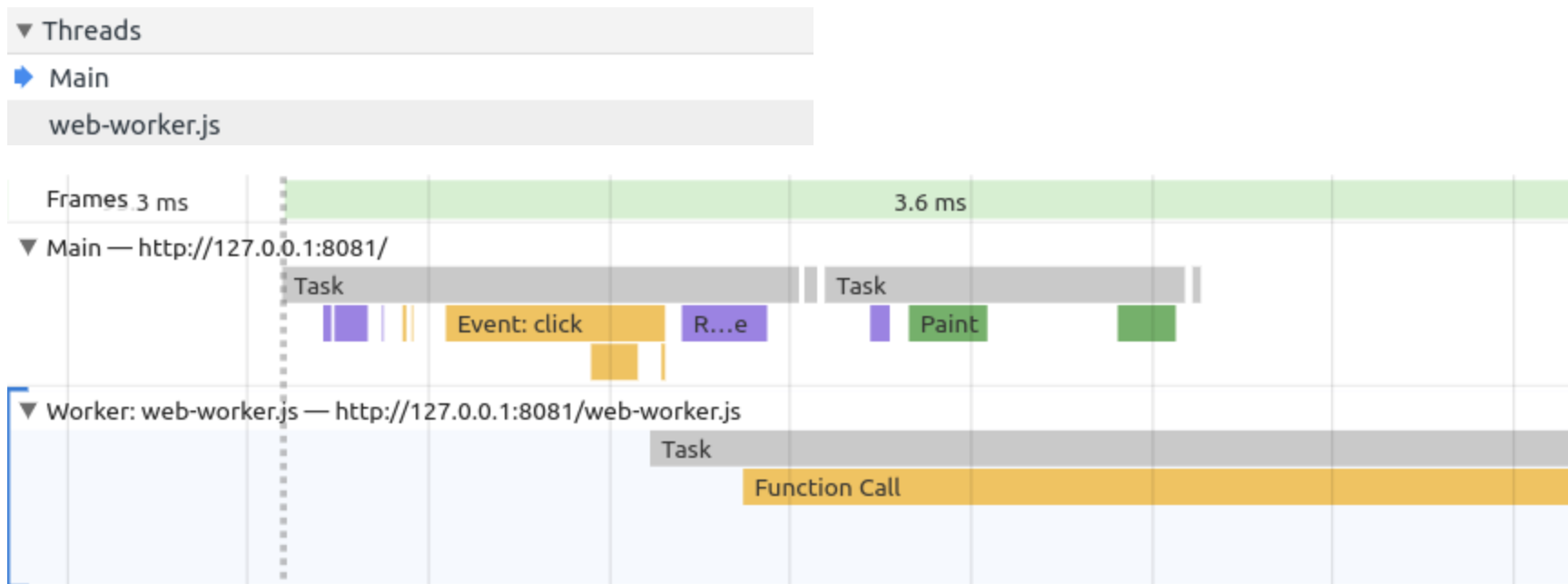
app.js:

```
1  const worker = new Worker('web-worker.js');
2  worker.addEventListener('message', evt => {
3    console.log(evt);
4  });
5  worker.addEventListener('error', evt => {
6    console.error(evt);
7  });
8  btn.addEventListener('click', () => {
9    worker.postMessage(1e11); // конечно же, мы можем отправить более сложный объект
10 });
```

web-worker.js:

```
1  self.addEventListener('message', evt => {
2    let sum = 0;
3    for (let i = 0; i < evt.data; i++) {
4      sum += i;
5    }
6    self.postMessage(sum);
7  });
```

# ПРИМЕР WORKER'A





# **CALLBACK HELL**

# ЗАДАЧА

Перед нами стоит следующая задача: загрузить аналитические данные с сервера и произвести обработку данных на стороне пользователя, выдав ему аналитический отчёт.

Почему на стороне пользователя, а не на сервере?

Не всегда у нас есть возможность получить доступ к серверу. Возможно, мы используем API Vk для получения этих данных. А разработчики Vk врядли вам дадут написать на их серверах свою аналитическую функцию ☹️. Хотя на самом деле, некоторые подобные возможности они предоставляют.

# CALLBACKS

```
1  function getResponse(args, callback) {  
2    // где-то внутри функции getResponse  
3    const response = ...;  
4  
5    callback(response);  
6  }  
7  
8  getResponse(..., (response) => { // наш callback  
9    ...  
10  });
```

# CALLBACK HELL

```
1  function getResponse(args, callback) {  
2      // где-то внутри функции getResponse  
3      const response = ...;  
4  
5      callback(response);  
6  }  
7  
8  function processResponse(args, callback) {  
9      // где-то внутри функции processResponse  
10     const data = ...;  
11  
12     callback(data);  
13 }
```

# CALLBACK HELL

```
1 // наш первый callback
2 getResponse(..., (response) => {
3     // наш второй callback
4     processResponse(..., (data) => {
5
6     });
7 });
```





# CALLBACK HELL

Нетрудно себе представить, что будет если вызовов у нас будет не 2, а хотя бы 10.

Структура кода превращается в большое количество вложенных вызовов.

Для этого даже придумали отдельный термин - [Callback Hell](#).



# PROMISES

# PROMISES

Использование Promise (обещания) - механизм, позволяющий упростить написание асинхронного кода и решить ряд проблем callback'ов.

```
1  function getResponse(args) {  
2    // Do something  
3    return new Promise((resolve, reject) => {  
4      ...  
5    });  
6  }
```

Теперь функции не принимают callback для вызова, а возвращают объект класса `Promise`, который и будет играть ключевую роль.

# ИДЕЯ PROMISE

Ключевая идея Promise - это объект, который может находиться всего в трёх состояниях:

- `pending`
- `fulfilled`
- `rejected`

И единственное, что может произойти с `Promise` - это переход из состояния `pending` в состояние `fulfilled` или `rejected`.

Произойти этот переход может только один раз.



## ИДЕЯ PROMISE

Поскольку функция, выполняющая асинхронную операцию не может вернуть значение этой операции, она возвращает Promise, который и “заворачивает” результат выполнения этой операции.

# СОЗДАНИЕ PROMISE

```
1  function getResponse(args) {  
2    // Do something  
3    return new Promise((resolve, reject) => {  
4      setTimeout(() => {  
5        resolve('value');  
6      }, 500);  
7    });  
8  }  
9  
10 const responsePromise = getResponse(args);
```

# СОЗДАНИЕ PROMISE

`resolve`, `reject` - функции, вызываемые по завершении операции и переводящие `Promise` в состояние `fulfilled` или `rejected`, соответственно.

# THEN

Метод, принимающий callback, который должен вызваться в случае перехода `Promise` в состояние `fulfilled`:

```
1  const responsePromise = getResponse(args);
2  responsePromise.then((response) => {
3    ...
4  });
```



# ОБРАБОТКА ОШИБОК

При переходе `Promise` в состояние `rejected` вызывается callback, указанный вторым параметром в методе `then`:

```
1  const responsePromise = getResponse(args);
2  responsePromise.then((response) => {
3    ...
4  }, (error) => { // callback for rejected
5    ...
6  });
```

# CATCH

Метод, принимающий callback, который должен вызваться в случае перехода `Promise` в состояние `rejected` или выбрасывания исключения (если оно произошло в коде `then`):

```
1  const responsePromise = getResponse(args);  
2  responsePromise.catch((error) => {  
3    ...  
4  });
```

## THEN + CATCH

```
1  const responsePromise = getResponse(args);
2  responsePromise.then((response) => {
3    ...
4  }).catch((error) => {
5    // callback for `rejected` и обработчик ошибок в `then`
6
7  });
```

# FINALLY

Метод, принимающий callback, который должен вызваться в случае перехода `Promise` в состояние `fulfilled` или `rejected` (вне зависимости от того, в какое состояние перешёл `Promise`).

# FINALLY

```
1  const responsePromise = getResponse(args);
2  responsePromise.then((response) => {
3    ...
4  }).catch((error) => {
5    ...
6  }).finally(() => {
7    // final actions
8  });
```

Используется для исключения дублирования кода в `then` и `catch`



# PROMISIFICATION

Использование Promise потребовало переписывания старого кода.

Переписывание старого кода (без Promise ) с использованием Promise обозначают термином **Promisification**

# ЦЕПОЧКИ PROMISE

`Promise` можно объединять в цепочки, если `then` возвращает тоже `Promise`:

```
1  function getResponse(args) {  
2    // Do something  
3    return new Promise((resolve, reject) => { ... });  
4  }  
5  
6  function processResponse(response) {  
7    // Do something  
8    return new Promise((resolve, reject) => { ... });  
9  }
```

## ЦЕПОЧКИ PROMISE

```
1  getResponse(args).then((response) => {  
2    return processResponse(response);  
3  }).then((data) => {  
4    // do something  
5  }).catch((error) => {  
6    // handle error  
7  }).finally(() => {  
8    // final handlings  
9  })
```



# ИТОГИ ПО PROMISE

Зачем нужны `Promise`, почему не делать всё на callback'ах?

1. Использование `Promise` упрощает работу с асинхронным кодом, помогая избежать Callback Hell
2. Современное API написано с использованием `Promise`, поэтому важно уметь использовать этот инструмент
3. `Promise` не отменяют callback'и - их всё равно придётся использовать
4. Переход из состояния `pending` в состояние `fulfilled` или `rejected` может произойти **только один раз**
5. `Promise` нельзя отменить



**В КАКОМ ПОРЯДКЕ ВЫЗЫВАЕТСЯ `then`,  
`catch` ?**

| В том, в котором записаны

# В КАКОМ ПОРЯДКЕ ВЫЗЫВАЕТСЯ `then`, `catch` ?

```
1  const promise = getResponse();
2  promise.then((data) => {
3    // выброс ошибки, сработает следующий по блоку `catch`
4    throw new Error();
5  }).catch((error) => {
6    console.log('first error happened:');
7  })
8  .then((data) => {
9    console.log(data); // <- сработает `then`
10 }).catch((error) => {
11   console.log('second error happened:'); // <- не сработает
12 }); // undefined
```

Почему так?

## THEN И CATCH

Методы `then` и `catch` тоже возвращают `Promise`, благодаря чему возможно построение цепочки `Promise`.

# THEN И CATCH

Особенности `then`:

- `then` возвращает `Promise`
- если из `then` возвращается значение, то оно автоматически заворачивается в `Promise`, который переходит в состояние `fulfilled`
- соответственно, если из `then` ничего не возвращается, то в `Promise` кладётся значение `undefined`
- если в `then` выбрасывается ошибка, то ошибка автоматически заворачивается в `Promise`, который переходит в состояние `rejected`
- если из `then` возвращается `Promise`, то последующие вызовы `then` и `catch` будут обрабатывать его состояние

# THEN И CATCH

Особенности `catch`:

- `catch` возвращает `Promise`
- если из `catch` возвращается значение, то оно автоматически заворачивается в `Promise`, который переходит в состояние `fulfilled`
- соответственно, если из `catch` ничего не возвращается, то в `Promise` кладётся значение `undefined`
- если в `catch` выбрасывается ошибка, то ошибка автоматически заворачивается в `Promise`, который переходит в состояние `rejected`
- если из `catch` возвращается `Promise`, то последующие вызовы `then` и `catch` будут обрабатывать его состояние

# СТАТИЧЕСКИЕ МЕТОДЫ PROMISE

Класс `Promise` содержит ещё ряд статических методов, предоставляющих удобную функциональность:

- `Promise.all(iterable)` - возвращает `Promise`, который переходит в состояние `fulfilled`, только если все `Promise` из `iterable` перешли в состояние `fulfilled` (либо в `iterable` не было `Promise` )
- `Promise.allSettled(iterable)` - возвращает `Promise`, который переходит в состояние `fulfilled`, только если все `Promise` из `iterable` перешли в состояние `fulfilled` или `rejected` (либо в `iterable` не было `Promise` )
- `Promise.race(iterable)` - возвращает `Promise`, который переходит в состояние `fulfilled` или `rejected` как только любой из `Promise`, содержащихся в `iterable` переходит в `fulfilled` или `rejected`

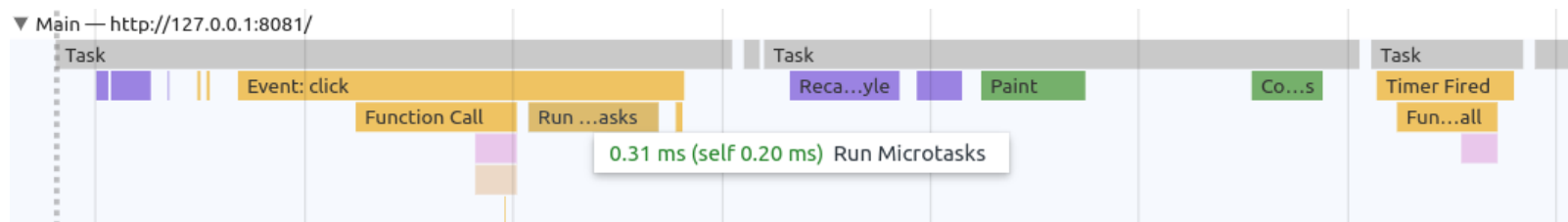
`Promise` имеет достаточно интересное поведение в рамках рассмотренного нами механизма Event Loop: `then`, `catch` и `finally` завершённого `Promise` (`fulfilled` или `rejected`) ставятся в специальную очередь, которую мы будем называть `microtask queue`:



```

1  const btn = document.getElementById('btn');
2  btn.addEventListener('click', () => {
3    setTimeout(doMacroTask, 0);
4    Promise.resolve().then(doMicroTask);
5  });
6
7  function doMacroTask() {
8    console.log('macro');
9  }
10
11 function doMicroTask() {
12   console.log('micro');
13 }

```



Также есть метод глобального объекта [queueMicrotask](#), позволяющий запланировать выполнение функции.



# ASYNC/AWAIT

# ASYNC/AWAIT

С `Promise` всё достаточно хорошо, но есть ли механизмы ещё более упростить этот код?

Ключевые слова `async` / `await` позволяют сделать работу с `Promise` более удобной.

Рассмотрим сразу на примере.

# ASYNC/AWAIT

```
1 | const response = await getResponse(args);  
2 | const data = await processResponse(response);
```

И ЭТО ВМЕСТО:

```
1 | const promise = getResponse();  
2 | promise.then((response) => {  
3 |     return processResponse(response);  
4 |     .then((data) => {  
5 |         // Do something  
6 |     });
```

# ОБРАБОТКА ОШИБОК И FINALLY

Здесь тоже всё хорошо, используем конструкцию

`try...catch...finally`

```
1  try {  
2    const response = await getResponse(args);  
3    const data = await processResponse(response);  
4  } catch {  
5    ...  
6  } finally {  
7    ...  
8  }
```

# ASYNC

На использование `await` есть одно ключевое ограничение: `await` можно использовать только внутри `async` функций:

```
1  (async () => {  
2    try {  
3      const response = await getResponse(args);  
4      const data = await processResponse(response);  
5    } catch {  
6      ...  
7    } finally {  
8      ...  
9    }  
10 })();
```

# ASYNC

Ключевое слово `async` определяет, что функция выполняется асинхронно - т.е. всегда возвращает `Promise`, но может выглядеть как стандартная функция.

*Что значит как стандартная функция?* Это значит, что если вы просто возвращаете из такой функции значение, то оно заворачивается в `Promise`.

Кроме того, вы можете использовать `await` внутри `async` функции, которое дожидается перехода `Promise` (`await` ставится перед `Promise`) в состояние `fulfilled` или `rejected`.

## ДЛЯ ЧЕГО ЭТО?

Для упрощения структуры кода, сравним:

```
1  (async () => {  
2    try {  
3      const response = await getResponse(args);  
4      const data = await processResponse(response);  
5    } catch {  
6      ...  
7    } finally {  
8      ...  
9    }  
10 })();
```



## ДЛЯ ЧЕГО ЭТО?

Для упрощения структуры кода, сравним:

```
1  |  getResponse(args).then((response) => {  
2  |          return processResponse(response);  
3  |   }).then((data) => {  
4  |          // do something  
5  |   }).catch((error) => {  
6  |          // handle error  
7  |   }).finally(() => {  
8  |          // final handling  
9  |   })
```

## ДЛЯ ЧЕГО ЭТО?

Первый вариант намного более лаконичный за счёт того, что позволяет избежать нагромождения `then`, `catch`.

# ASYNC/AWAIT

Почему бы тогда совсем не отказаться от `Promise`?

Потому что в основе работы `async` / `await` лежат `Promise`.  
`async` / `await` позволяет нам лишь удобнее с ними работать.

# ASYNC/AWAIT & PROMISE

"Конкурентные" запросы:

```
1 | const [resp1, resp2] = await Promise.all(  
2 |   getResponse(<from server1>), getResponse(<from server2>)  
3 | );
```

Последовательные запросы:

```
1 | await getResponse(<from server1>);  
2 | await getResponse(<from server2>);
```

# BABEL

```
1 | $ npm install core-js@3
```

B .babelrc:

```
1 | {  
2 |   "presets": [  
3 |     [  
4 |       "@babel/preset-env",  
5 |       {  
6 |         "useBuiltIns": "usage",  
7 |         "corejs": 3  
8 |       }  
9 |     ]  
10 |   ]  
11 | }
```



# ТЕСТИРОВАНИЕ АСИНХРОННОГО КОДА



# ТЕСТИРОВАНИЕ АСИНХРОННОГО КОДА

Jest предлагает для всех рассмотренных нами вариантов (callback'и, `Promise`, `async` / `await`) удобные методы для тестирования. Рассмотрим их.

# CALLBACKS

```
1 // специальный аргумент
2 test('should call our callback', (done) => {
3     getData((data) => {
4         expect(data).toEqual(...);
5         done(); // <- указание на завершение теста
6     });
7 });
```



# ТЕСТИРОВАНИЕ АСИНХРОННОГО КОДА

`done` - функция, вызова которой Jest будет ожидать в течение времени, определённого `jest.setTimeout` (по умолчанию - 5 секунд).

Если вызова не будет, получим FAIL:

**FAIL** test/data.test.js (5.352s)

✕ should call our callback (5015ms)

• should call our callback

Timeout - Async callback was not invoked within the 5000ms timeout specified by jest.setTimeout.

```
1 | import { getData } from '../src/js/demo';
2 |
> 3 | test('should call our callback', (done) => { // <- специальный аргумент
   |   ^
4 |   getData((data) => {
5 |     expect(data).toBe('...');
6 |     done(); // <- указание на завершение теста

at Spec (node_modules/jest-jasmine2/build/jasmine/Spec.js:85:20)
at Object.test (test/data.test.js:3:1)
```

Test Suites: 1 failed, 1 total

Tests: 1 failed, 1 total

# PROMISE И ASYNC/AWAIT

При работе с `Promise` и `async / await` достаточно использовать асинхронные тестовые функции (и работать как обычно):

```
1 // async
2 test('should work with promise and async/await', async () => {
3     const data = await getData();
4     expect(data).toEqual(...);
5 });
```

# ERROR HANDLING

```
1 // async
2 test('should handle errors', async () => {
3   // сообщаем Jest, что у нас один assert,
4   // который нужно проверить
5   expect.assertions(1);
6   try {
7     const data = await getData();
8   } catch (e) {
9     // получаем аналог .catch()
10    expect(e).toEqual(...);
11  }
12 });
```



## ПОДВЕДЕМ ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей:

1. Асинхронный код
2. Promises
3. `async/await`
4. Тестирование асинхронного кода



**Задавайте вопросы и напишите отзыв о лекции!**

**АЛЕКСАНДР РУСАКОВ**



[a\\_s\\_rusakov@mail.ru](mailto:a_s_rusakov@mail.ru)



[@a\\_s\\_rusakov](https://t.me/a_s_rusakov)