



Асинхронность

Курс JavaScript про ECMAScript



Оглавление

Введение	2
Термины, используемые в лекции	2
Название первой темы в этой лекции	2
Название второй темы в этой лекции	5
Домашнее задание	6
Что можно почитать еще?	6
Используемая литература	6

Асинхронность

Движок JavaScript устроен так, что весь код выполняется в одном потоке. И несмотря на то, что у вас может быть многоядерный процессор, исполнение JavaScript кода на одной странице будет обрабатываться только в одном потоке и в одном ядре процессора. Поэтому, если какой то код будет выполняться очень долго, то он займет этот поток, и наша страница перестанет отвечать на действия пользователя. Попробуйте выполнить следующий код в консоли браузера:

Листинг 1

```
console.log('Start long calculations');  
for (let i = 0; i < 1000000000; i++) {  
    const newDate = new Date(i);  
}  
console.log('End long calculations');  
// Start long calculations  
// End long calculations
```

Страница в браузере как бы заморозилась на минуту (больше или меньше, зависит от процессора в вашем компьютере). Так происходит, потому что цикл `for` синхронный, и код выполняется в одном потоке, не позволяет выполнить другие операции. Также есть множество других операций, ожидание выполнения которых может навредить пользовательскому опыту, если страница не будет реагировать на действия пользователя в это время:

- выполнение запросов к серверу,
- ожидание ответных действий пользователя,
- отложенное выполнение кода,
- сложные анимации.

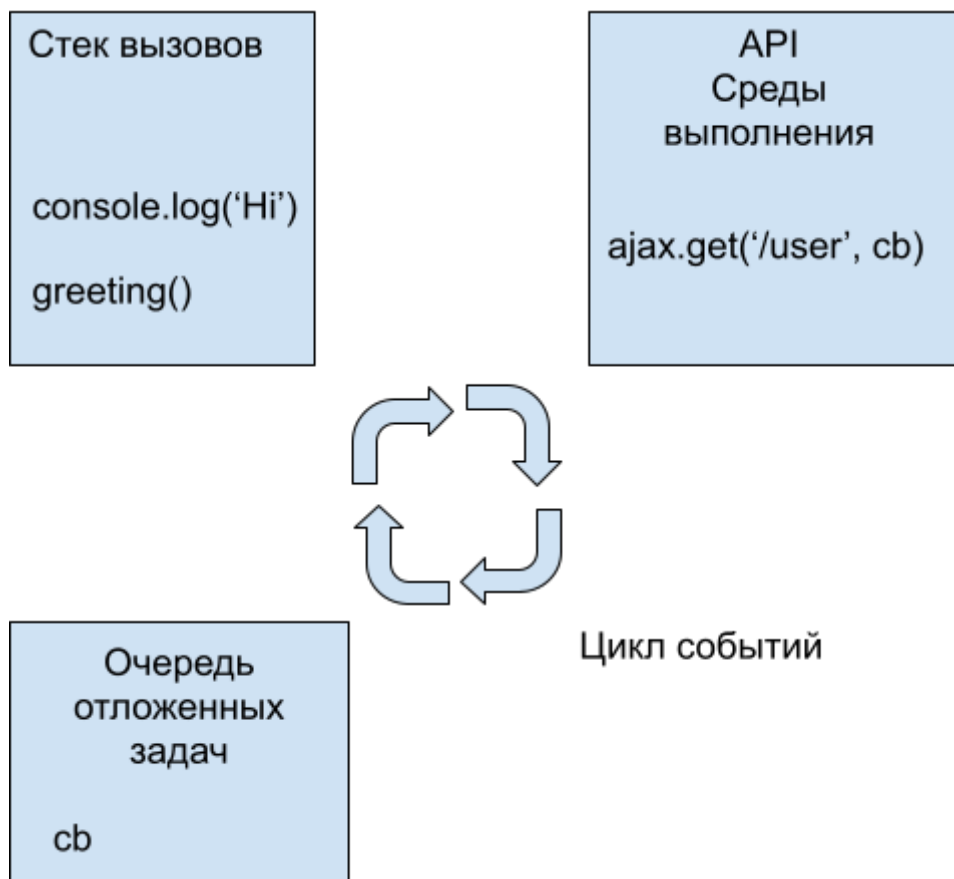
Для решения этой проблемы придумали механизмы для написания асинхронного кода - это код, обычно состоящий из двух частей:

- часть, которая выполняется долго или требует значительных ресурсов,
- часть, которая вызывается по завершению первой части и может обработать результат выполнения первой части или просто сообщает что выполнение первой части кода завершилось.

Главный механизм для работы такого кода - это **Цикл Событий (Event Loop)**, который передает сложную и долгую часть кода на исполнение в API среды выполнения, а часть кода, которая должна быть выполнена после, помещает в очередь отложенных задач, а также следит за тем, когда можно будет выполнить задачи из этой очереди. Давайте рассмотрим подробнее как это работает.

Event loop

Цикл событий или Event Loop по английски - это механизм, который который умеет обрабатывать входящие события (это сам код, и его команды, действия пользователя, взаимодействие с операционной системой), и управлять выполняемыми задачами, исходя из этих событий. Давайте посмотрим на картинку, чтобы было понятнее:



В работе цикла событий участвует еще три механизма:

1. Call Stack (стек вызовов функций) - это механизм, который используется, чтобы сказать процессору какие команды выполнять, по мере исполнения алгоритма. Это основной поток синхронных команд нашего алгоритма.
2. API среды выполнения - это вспомогательный функционал, который предоставляет браузер или другая среда выполнения, так например функция `setTimeout` является не функцией движка JavaScript, а одной из функций среды выполнения.
3. Отложенная очередь задач - это стек задач, который пополняется новыми заданиями, по мере их появления. Так например вызов функции обратного вызова в выполняемой в текущий момент функции будет внесен в этот стек, чтобы цикл событий выполнил эту функцию когда у него будет свободный стек вызовов.



Что такое стек вызовов (call stack) разьяснялось в курсе по основам программирования.

Давайте разберем на реальном пример как все это работает по шагам:

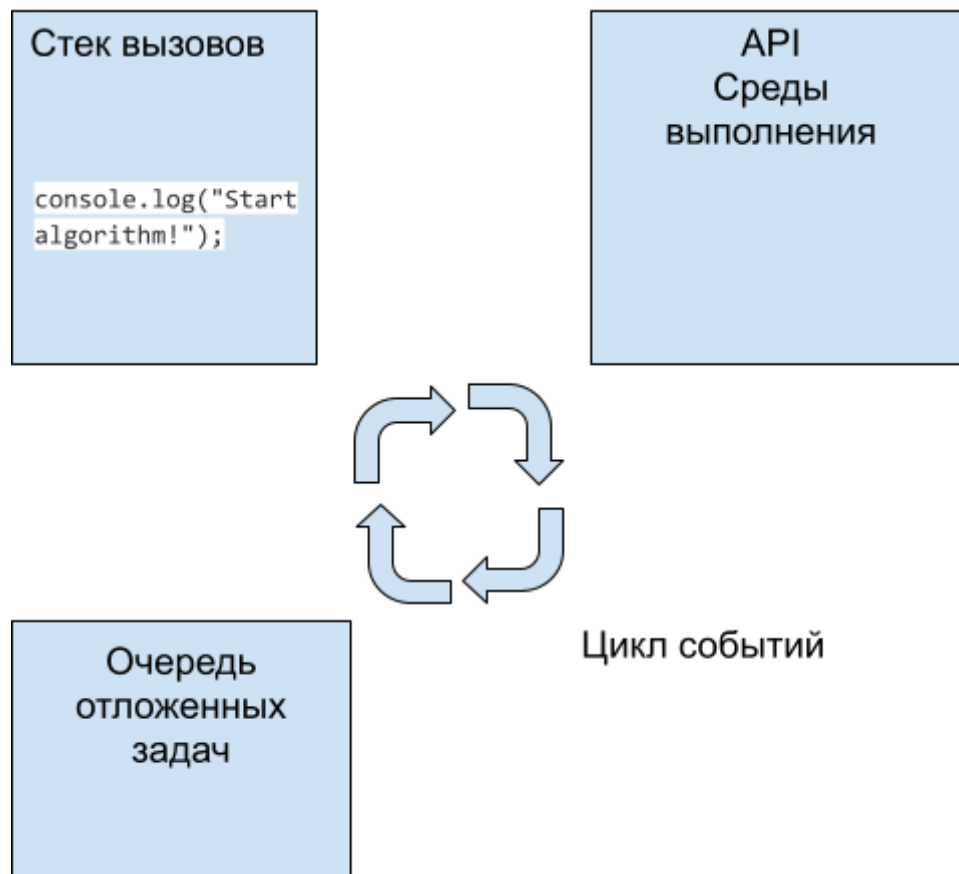
Листинг 2

```
console.log("Start algorithm!");

setTimeout(function timeout() {
  console.log("This will be printed after 5 seconds!");
}, 5000);

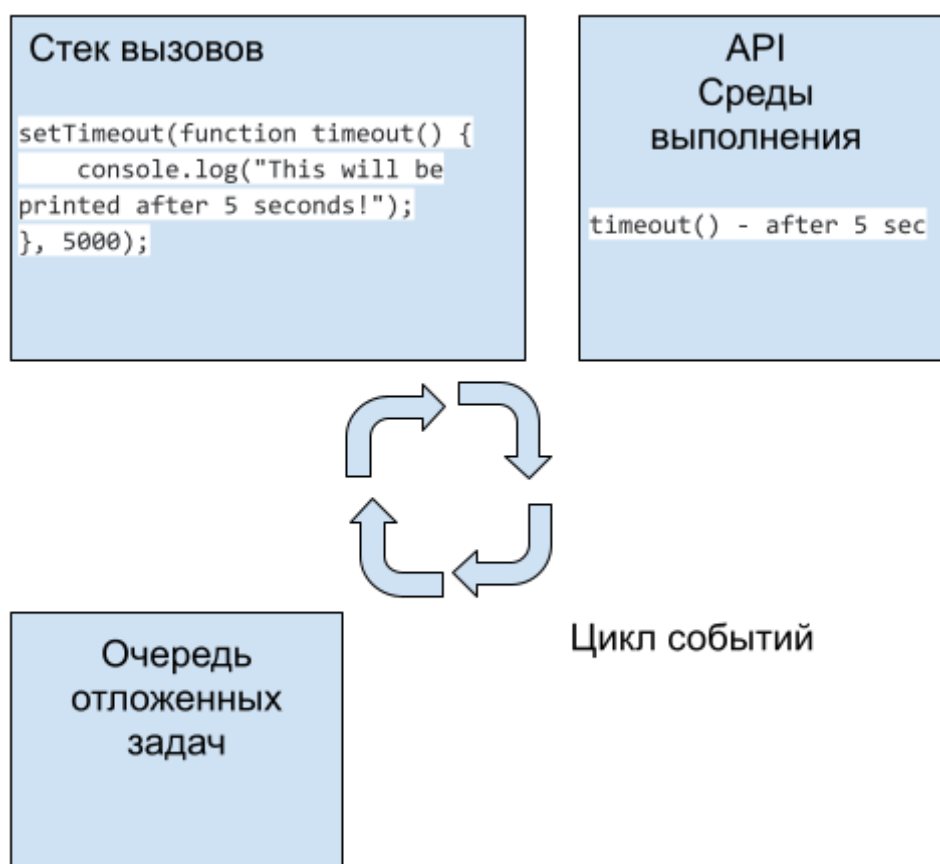
console.log("End synchronous code.");
```

Первой командой выполняется вывод в консоль (строка 1). Эта команда попадает в стек вызовов:

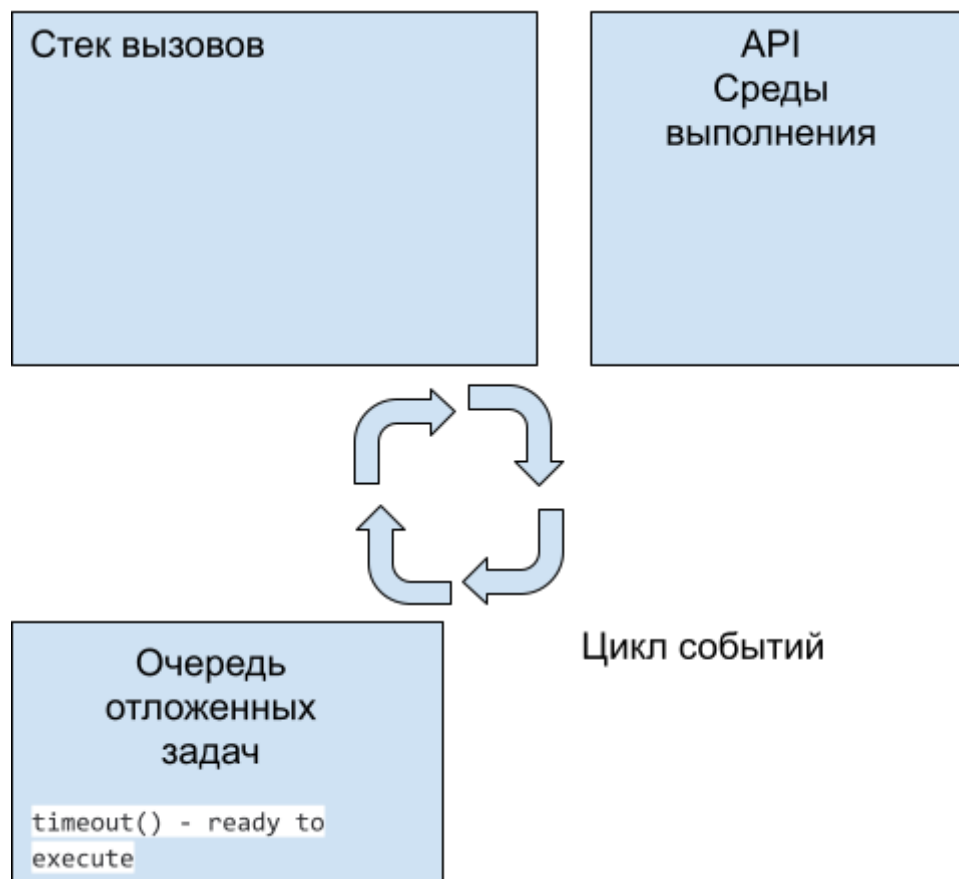


Эта команда синхронная, поэтому она выполняется и стек освобождается.

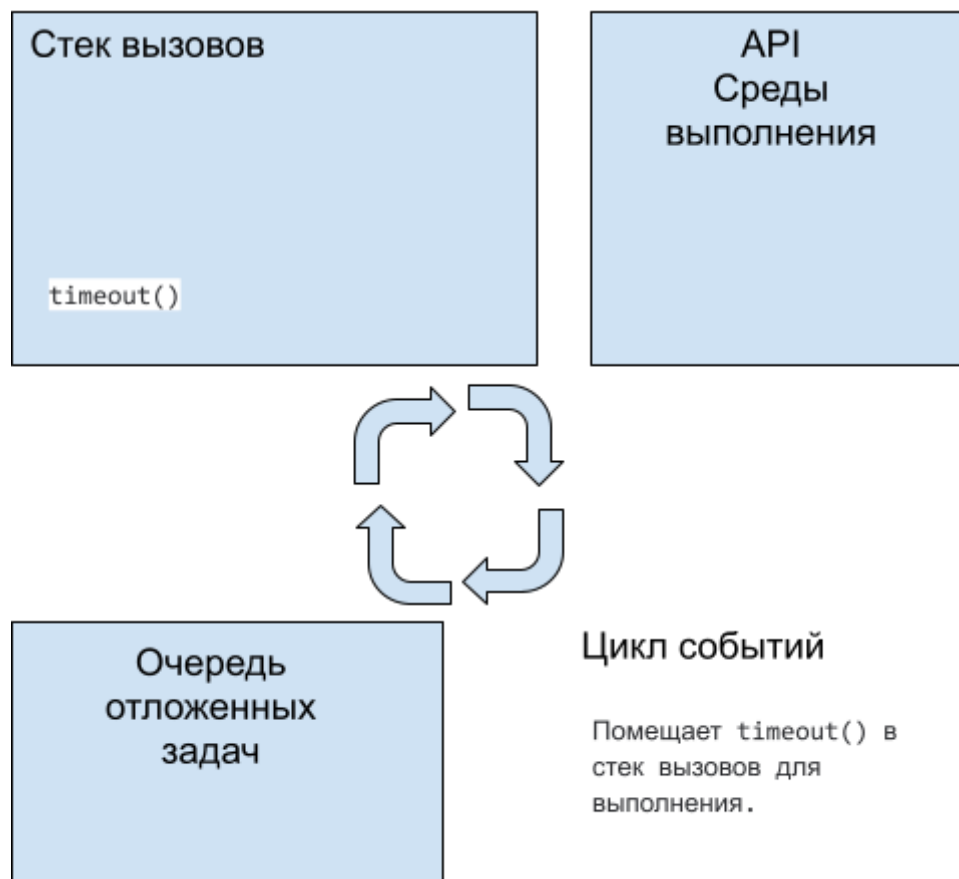
На 3й строке мы видим вызов функции `setTimeout` - это функция предоставляется API браузера и она асинхронная. Поэтому вызов этой функции осуществляется браузером и так, как в этой функции есть функция обратного вызова, API браузера запомнит её и через 5 секунд поместит в очередь отложенных задач:



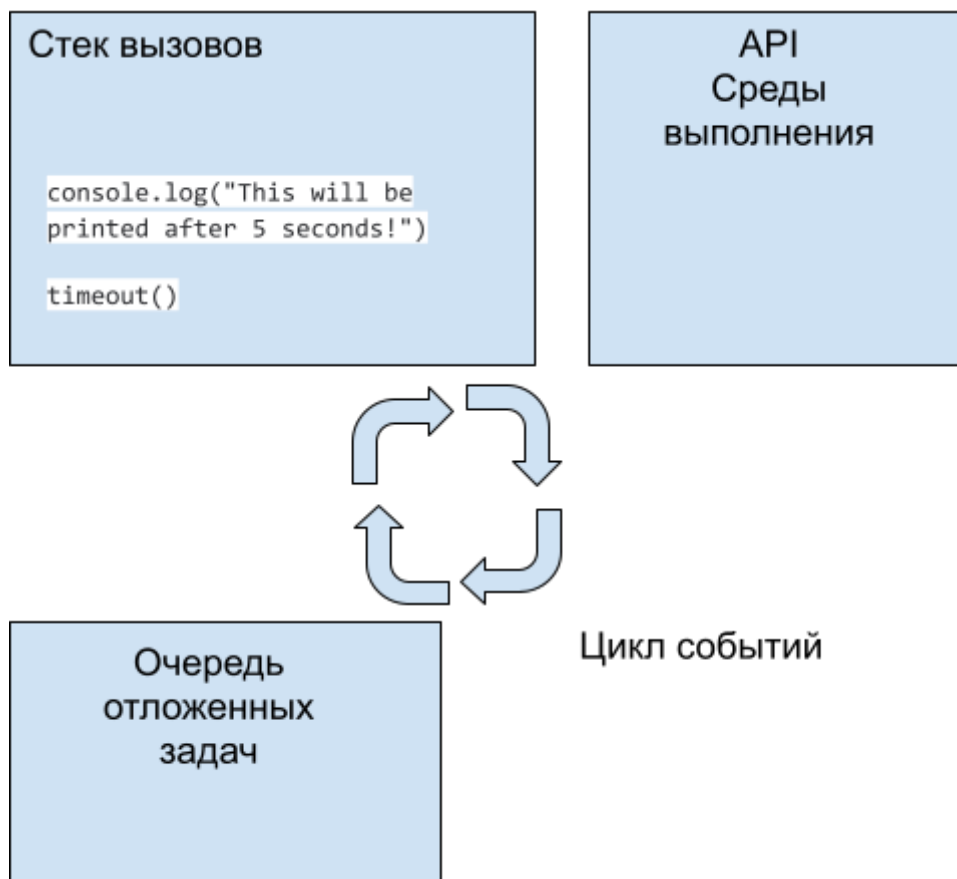
Пока пять секунд еще не прошли, движок JavaScript будет спокойно выполнять наш алгоритм дальше, и выполнит команду из 7 строки, которая выведет в консоль, что синхронные операции в нашей программе закончены, и стек очиститься. А по прошествии 5 секунд с момента вызова функции `setTimeout`, API браузера поместит функцию обратного вызова в очередь отложенных задач, и вся система будет выглядеть так:



В этот момент как раз и подключается цикл событий, он смотрит что в стеке вызовов пусто, и можно взять задачу из очереди отложенных задач, он находит там функцию обратного вызова `timeout()`, и помещает её в стек вызовов для выполнения:



А дальше код этой функции выполняется как обычно, в ней мы видим вывод в консоль, который выполняется и стек вызовов после отработки очищается.



Вот так все просто. Синхронные команды выполняются в стеке вызовов, а все асинхронное выполняется API окружения, отложенные задачи попадают в очередь, а цикл событий следит за стеком и очередью отложенных задач, и как только стек опустошается, начинает заполнять его отложенными задачами. Исходя из этого можно понять, что интервал указанных в функции `setTimeout` не гарантирует нам что функция обратного вызова будет вызвана ровно через пять секунд, т.к. цикл событий переместит её в стек вызовов для выполнения, только когда он будет пуст. Если у нас в этот момент будет выполняться сложная синхронная операция, которая займет стек на несколько секунд, то наша функция обратного вызова будет вызвана с большей задержкой.

Цикл событий можно еще представить себе как диспетчера, который сидит между двух конвейеров, слева это стек вызовов, а справа очередь отложенных задач. Диспетчер следит за стеком вызовов, и как только этот конвейер становится пустым, он смотрит нет ли отложенных задач на втором конвейере, и если они есть, перекладывает их на первый конвейер.

Порой возникает вопрос, какой код нужно писать асинхронно, а какой синхронно. Обычно асинхронными делаются функции доступа, на выполнение которых тратится значительное время, например обращение к файловой системе, к базе данных, сетевые запросы. Асинхронный код при этом позволяет не замораживать ваше приложение, дожидаясь ответа от этих функций, а позволить пользователям взаимодействовать с ним и дальше. Так, например, на загружаемой странице может быть много картинок, которые будут загружаться асинхронно, а в этот момент пользователь уже сможет взаимодействовать с кнопками и полями ввода. Или серверный код приложения на node.js может обрабатывать тысячи одновременных запросов пользователей, за счет асинхронной обработки каждого запроса, и по мере отработки каждого из них отвечать каждому пользователю. Многие функции в JavaScript (как например `setTimeout`) уже являются асинхронными, другие имеют два варианта, как синхронный, так и асинхронный, какой лучше использовать в каждой части кода приходит с опытом. Какие то функции и операторы являются только синхронными, например цикл `for`, и если в нем выполняется долгая операция, она может занять поток выполнения и страница перестанет отвечать на действия пользователя, что мы видели в первом примере этого урока, поэтому такой код лучше переписать с использованием асинхронных функций, если это возможно - в конце урока мы познакомимся как это можно сделать.

Асинхронность, обусловленная сетевым взаимодействием, AJAX

Сетевые взаимодействия - это наиболее частое применение асинхронности в JavaScript. Все запросы из браузера (или между серверами) работают с помощью протокола HTTP.

HTTP - HyperText Transfer Protocol

Протокол передачи гипертекста - это протокол передачи данных прикладного уровня, что означает что он использует основной протокол обмена данными в сетях (например TCP/IP), и работает по принципу клиент/сервер. Клиент (это может быть браузер или один из серверов) отправляет запрос (request) к серверу, а тот получая

запрос отправляет в ответ необходимые данные - ответ (response). Изначально применялся для передачи html страниц, но в дальнейшем стал применяться для передачи любых данных.

Для получения данных от сервера нам нужно отправить на указанный адрес запрос. Запрос должен быть определенного типа (GET, POST, PUT, DELETE и другие) и оформлен специальным образом: содержать необходимые заголовки, из которых сервер получает служебную информацию (об авторизации, клиенте и другую) и тело запроса (при необходимости). Основные запросы на получение и передачу данных на сервер это запросы типа GET и POST, давайте рассмотрим их подробнее.

GET

GET запрос нужен для получения данных от сервера. Такой запрос не имеет тела, и все необходимые данные запрашивает с помощью адреса и GET параметров в строке адреса. Параметры в адресе выглядят как <ключ>=<значение>, соединяются с помощью знака & и отделяются от основного адреса знаком ?, например `www.google.com?page=1&amount=20`. GET запрос должен всегда возвращать один и тот же ответ, обращаясь по одному и тому же адресу. С помощью GET запросов можно получать картинки, данные о пользователях и любую другую информацию, хранящуюся на сервере.

POST

POST запрос нужен для отправки данных на сервер. Чаще всего он используется для отправки данных формы на сервер. POST запрос обязательно передает данные в теле запроса. Сервер может возвращать какие то данные на POST запросы, чаще всего это результат сохранения данных, полученных с этим запросом.

Выполнение всех этих запросов может идти как с перезагрузкой страницы, когда мы вставляем запрос в адресную строку (GET), или подтверждаем ввод данных в форме, и страница перезагружается, т.к. идет POST запрос, который генерирует сам браузер, так и без перезагрузки страницы с использованием отправки запросов с помощью технологии AJAX.

AJAX

Ajax с английского Asynchronous Javascript and XML - технология, которая позволяет сделать асинхронный запрос к серверу для получения данных в виде XML разметки или в каком-нибудь другом виде. Именно эта технология позволила вдохнуть новую жизнь в веб приложения, отправлять данные из форм на сервер без перезагрузки страницы, получать новые данные от сервера и отображать их пользователю в реальном времени. Создавать различные живые чаты, уведомления и многое другое. Все это позволило улучшить пользовательский опыт взаимодействия со сложными страницами и многостраничными приложениями на веб сайтах.

Теперь, когда мы изучили теорию, мы можем перейти к практике, и научиться создавать свои асинхронные запросы к серверу с использованием технологии AJAX, а поможет нам в этом объект XMLHttpRequest.

Объект XMLHttpRequest

Объект XMLHttpRequest позволяет нам создать объект запроса, который можно отправить на сервер, и обработать ответ. Это базовый механизм, реализуемый движком JavaScript, он не очень удобен для работы, но на нем можно хорошо понять принципы работы с XHR запросами (XHR - сокращение от XMLHttpRequest).

Давайте рассмотрим создание GET запроса на примере получения данных о пользователе сервиса github.com. Данный сервис имеет открытое api, которое можно использовать в своих примерах.

Чтобы получить данные о пользователе сервиса github по имени octocat, нужно обратиться с помощью GET запроса по адресу: <https://api.github.com/users/octocat>.

Листинг 3

```
// XHR GET запрос
// Для создания запроса сначала нужно создать объект
XMLHttpRequest, конструктор вызывается без аргументов.
const xhr = new XMLHttpRequest();
// Открываем запрос. Первый аргумент это метод (GET, POST ...),
```

второй адрес, куда нужно отправить запрос. У данного метода есть еще аргументы, которые мы рассмотрим позже.

```
xhr.open('GET', 'https://api.github.com/users/octocat');
```

// Отправляем запрос на сервер. Метод `send()` может принимать один аргумент – это тело запроса, если оно есть. Для методов `POST` мы бы передавали тут тело запроса, наш запрос `GET` не имеет тела запроса.

```
xhr.send();
```

// После отправки запроса нам нужно использовать методы слушатели, которые будут асинхронно выполнены при наступлении соответствующего события. Это методы `onload`, `onerror` и `onprogress` – мы сами определяем их функции.

// `onload` – будет вызван, когда сервер вернет ответ. Это может быть положительный ответ сервера с кодом 200 (все хорошо), так и ошибка, например страница не найдена, тогда код будет 404. Мы должны обрабатывать такие ситуации сами.

```
xhr.onload = function() {  
    if (xhr.status !== 200) { // если статус не 200, то произошла  
        ошибка  
        console.log(`Error ${xhr.status}: ${xhr.statusText}`);  
    } else {  
        console.log(`user: ${xhr.response}`); // response – это ответ  
        сервера  
    }  
};
```

// `onprogress` – функция будет вызываться пока запрос находится в процессе, и позволяет отслеживать ход процесса отправки запроса и получения ответа от сервера. Бывает полезна при отладке сложных запросов.

```
xhr.onprogress = function(event) {  
    console.log(`Get from server: ${event.loaded} bytes`);
```

```

};

// onerror - функция будет вызываться если в запросе пошло
что-то не так. Например нет соединения с сервером или ошибка в
адресе запроса.
xhr.onerror = function() {
    console.log("Request error");
};

// Вывод из функции onprogress.
// Get from server: 1319 bytes

// Вывод из функции onload, приведен не полностью.
user: {`
    "login": "octocat",
    "id": 583231,
    "node_id": "MDQ6VXNlcjU4MzIzMQ==",
    "avatar_url":
"https://avatars.githubusercontent.com/u/583231?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/octocat",
    "html_url": "https://github.com/octocat",
    "followers_url":
"https://api.github.com/users/octocat/followers",
    "following_url":
"https://api.github.com/users/octocat/following{/other_user}",
    ...`
}

```

Запустить этот пример из консоли браузера не получится, т.к. Браузер блокирует запросы к серверам, сделанные из консоли. Поэтому можно запустить этот код в онлайн песочнице jsbin: <https://jsbin.com/ketohagavu/5/edit?html,js,output>

Если вдруг мы решили отменить запрос (такие ситуации бывают, если пользователю например дана кнопка отмены, или пользователь закрывает диалог, который вызывал запрос, и данные этого запроса уже будут не нужны) мы можем его прервать, вызвав метод `abort`.

Листинг 4

```
// Создан объект xhr и был выполнен запрос, но дальше он нам не  
нужен, прервем его.  
xhr.abort();
```

Давайте рассмотрим как отправить **POST** запрос с помощью объекта **XMLHttpRequest**.

Для отправки **POST** запроса нам надо собрать данные для отправки в виде объекта **FormData**, для этого можно использовать конструктор данного объекта, и дополнить его необходимыми полями. Обычно этот конструктор этого объекта вызывается с аргументом, в котором передается форма, из которой этот объект получит данные, но мы формы будем изучать во второй части курса, поэтому вызовем конструктор без указания формы, и дополним объект полями уже после. Для теста POST запроса нам нужен сервер, который мог бы его принять и что-то нам ответить. Для теста мы можем использовать прекрасный сервис <https://httpbin.org/>, который умеет принимать любой запрос и отправляет полученные данные в виде ответа. С помощью этого сервиса можно легко экспериментировать и тестировать различные запросы. Давайте создадим наш POST запрос, в котором будем передавать имя и фамилию пользователя.

Листинг 5

```
// XHR POST запрос  
// Создадим объект FormData.  
const formData = new FormData();  
  
// Добавим в него два наших поля.
```



```
formData.append("name", "Slava");
formData.append("surname", "Belkin");

// Для создания запроса сначала нужно создать объект
XMLHttpRequest, конструктор вызывается без аргументов.
const xhr = new XMLHttpRequest();
// Открываем запрос. Первый аргумент это метод (GET, POST ...),
второй адрес, куда нужно отправить запрос. У данного метода есть
еще аргументы, которые мы рассмотрим позже. В данном случае мы
создаем POST запрос на тестовый сервер https://httpbin.org/post.
xhr.open('POST', 'https://httpbin.org/post');

// Отправляем запрос на сервер. Метод send() может принимать
один аргумент - это тело запроса, если оно есть. Отправляем POST
запрос, поэтому передаем в качестве аргумента объект FormData.
xhr.send(formData);

// После отправки запроса нам нужно использовать методы
слушатели, которые будут асинхронно выполнены при наступлении
соответствующего события. Это методы onload, onerror и
onprogress - мы сами определяем их функции.
// onload - будет вызван, когда сервер вернет ответ. Это может
быть положительный ответ сервера с кодом 200 (все хорошо), так и
ошибка, например страница не найдена, тогда код будет 404. Мы
должны обрабатывать такие ситуации сами.
xhr.onload = function() {
    if (xhr.status != 200) { // если статус не 200, то произошла
ошибка
        console.log(`Error ${xhr.status}: ${xhr.statusText}`);
    } else {
        console.log(`user: ${xhr.response}`); // response - это ответ
сервера
    }
};
// onprogress - функция будет вызываться пока запрос находится в
```

процессе, и позволяет отслеживать ход процесса отправки запроса и получения ответа от сервера. Бывает полезна при отладке сложных запросов.

```
xhr.onprogress = function(event) {
    console.log(`Get from server: ${event.loaded} bytes`);
};

// onerror - функция будет вызываться если в запросе пошло
что-то не так. Например нет соединения с сервером или ошибка в
адресе запроса.
xhr.onerror = function() {
    console.log("Request error");
};

// Вывод из функции onprogress.
// Get from server: 1032 bytes

// Вывод из функции onload.
// user: {
    "args": {},
    "data": "",
    "files": {},
    "form": {
        "name": "Slava",
        "surname": "Belkin"
    },
    "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip, deflate, br",
        "Accept-Language": "ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7",
        "Content-Length": "240",
        "Content-Type": "multipart/form-data;
boundary=----WebKitFormBoundaryhzprTAMZWnth4PPa",
        "Dnt": "1",
        "Host": "httpbin.org",
        "Origin": "https://null.jsbin.com",
```

```
"Referer": "https://null.jsbin.com/",
"Sec-Ch-Ua": "\" Not A;Brand\";v=\"99\"",
\"Chromium\";v=\"90\", \"Google Chrome\";v=\"90\"",
"Sec-Ch-Ua-Mobile": "?0",
"Sec-Fetch-Dest": "empty",
"Sec-Fetch-Mode": "cors",
"Sec-Fetch-Site": "cross-site",
"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.93
Safari/537.36",
"X-Amzn-Trace-Id": "Root=1-608e859b-0d07ac850398c299447844c6"
},
"json": null,
"origin": "95.26.251.191",
"url": "https://httpbin.org/post"
}
```

Ссылка на jsbin примера: <https://jsbin.com/xesineyoce/2/edit?html,js,output>

Наш запрос ушел на тестовый сервер, и в ответ мы получили те же данные, что и отправили, мы можем увидеть их в ключе form.

Объект **XMLHttpRequest** имеет ряд других полезных методов, но его применение с каждым годом становится менее актуальным, т.к. есть метод fetch, который получил уже широкую поддержку браузеров, а также есть множество библиотек, упрощающих написание XHR запросов. Более подробно вы можете познакомиться с объектом XMLHttpRequest в [MDN](#).

Формат данных JSON

В мире Web есть множества форматов данных с помощью которых мы можем кодировать информацию для отправки на сервер или для получения данных с сервера: XML, простой текст, JSON, двоичные данные. Раньше был достаточно популярным формат XML, но в скором времени его сменил очень простой и удобный формат JSON.

JSON - JavaScript object notation или описание данных подобно объекту JavaScript. Этот формат данных очень сильно похож на объекты JavaScript за небольшими исключениями:

- Все ключи JSON объекта должны быть взяты в кавычки,
- Значения ключей - это примитивные типы, массив или объект. В нем не может быть функций,
- В JSON объекте не допускаются комментарии,
- В конце каждой группы ключей или элементов массивов нельзя ставить запятую - будет ошибка объекта.

Формат JSON был создан для удобной передачи данных из одного языка программирования в другой, так ваш клиентский код на JavaScript форматирует данные в виде JSON и отправляет на сервер, где эти данные могут обрабатываться сервером, написанном на языке Go, Ruby или PHP. Как оказалось этот формат очень удобен для работы клиентских приложений, и большинство взаимодействий с серверами идет в этом формате. Для удобства работы с данными в этом формате в JavaScript Есть объект JSON и несколько методов для преобразования данных в него и из него в настоящий объект JavaScript.

JSON.stringify

Метод `stringify` позволяет преобразовать JavaScript объект в строковое представление в JSON формате. Этот очень полезный метод применяется перед отправкой данных на сервер. Также этот метод позволяет сохранить данные в виде объекта из JavaScript в текстовый файл, например логи работы программы. Этот метод принимает 3 аргумента. Первый, обязательный аргумент - это сам объект, который будет преобразован. Второй - это функция обратного вызова, которая будет вызвана рекурсивно для всех ключей/значений объекта и которая позволяет модифицировать данные при необходимости прямо во время конвертации. Третий - это количество пробелов, которые будут использоваться для формирования отступов внутри строки, чтобы сохранить вид как в коде.

Листинг 8

```
// Объект -> JSON.
const student = {
  name: 'Slava',
  surname: 'Belkin',
  age: 20,
  practice: {
    place: 'IKTG',
    hours: 47,
  }
};
console.log(JSON.stringify(student, null, 4));
// Вывод в консоль. Обратите внимание, что все ключи стали
// обернуты в кавычки.
{
  "name": "Slava",
  "surname": "Belkin",
  "age": 20,
  "practice": {
    "place": "IKTG",
    "hours": 47
  }
}
```

JSON.parse

Метод `parse` делает обратную процедуру, принимая строку в качестве аргумента, метод пытается распарсить её как строку формата JSON и превратить в объект JavaScript (или массив).

Листинг 8

```
// JSON -> Объект.
console.log(JSON.parse('{"name": "Slava", "surname": "Belkin",
"age": 20, "practice": {"place": "IKGT", "hours": 47}}));
```

```
// {  
//   name: 'Slava',  
//   surname: 'Belkin',  
//   age: 20,  
//   practice: {  
//     place: 'IKTG',  
//     hours: 47,  
//     __proto__: Object  
//   },  
//   __proto__: Object  
// }  
  
console.log(JSON.parse('["Belkin", "Ivanov", "Petrov"]')); //  
["Belkin", "Ivanov", "Petrov"]
```

Мы успешно преобразовали строку с объектом в реальный объект JavaScript, а также строку с массивом в массив.

Если строка не будет соответствовать формату JSON, то вы получите синтаксическую ошибку, поэтому такие операции для безопасности лучше всего оборачивать в конструкцию try/catch.

async/await

Функционал async/await для работы с асинхронным кодом (а именно для работы с обещаниями) появился в JS с приходом стандарта ES7, и пока еще не слишком поддерживается браузерами, но есть полифилы для работы с ними.

Суть подхода async/await - это писать асинхронный код так, будто он выполняется синхронно, но при этом не блокирует основной поток выполнения. Подход состоит из применения двух операторов:

- `async` - пишется перед функцией и превращает любую функцию в обещание, а также позволяет использовать второй оператор `await` внутри себя.

- `await` - оператор пишется перед вызовом асинхронной функции, что заставляет код остановиться в этом месте, пока асинхронная функция не вернет результат.

Давайте посмотрим на пример:

Листинг 9

```
const getUser = async (url) => {  
    // Делаем запрос, и ждем его результат (указание await),  
    который будет сохранен в константу response.  
    const response = await fetch(url);  
    // Выполняем еще один асинхронный метод, преобразования в  
    текст, также ждем результат, который сохраняется в константу  
    пользователь.  
    const user = await response.text();  
    console.log(user);  
}  
  
getUser('https://api.github.com/users/octocat');  
  
// Вывод в консоль.  
// {  
//   "login": "octocat",  
//   "id": 583231,  
//   ...  
// }
```

Синтаксис **`async/await`** позволяет писать код очень линейно и чисто.

“Запланированная асинхронность” - `setTimeout, setInterval`

Бывают моменты, когда нам синхронный код превратить в асинхронный. Например мы не хотим делать запрос к серверу в тестовом приложении, а хотим просто имитировать задержку запросов, или вспомните самый первый пример из урока, там нам нужно было сгенерировать много дат, но при этом код был синхронным, и

мы занимали весь поток так, что исполнение других сценариев на странице останавливалось. Вот в таких случаях нам может пригодиться превращение синхронного кода в асинхронный, и сделать это можно с помощью функций `setTimeout` и `setInterval`, которые автоматически помещают функции обратного вызова в очередь отложенных задач, и которые будут выполняться асинхронно. Давайте возьмем наш первый пример с генерацией дат, и перепишем его так, чтобы он стал асинхронным и не блокировал основной поток:

Листинг 10

```
const timerId = setInterval(() => {
  if (counter > amount) {
    // После того как наш счетчик достигнет нужного
    // количества итераций, мы должны очистить таймер, чтобы итерации
    // больше не выполнялись.
    clearInterval(timerId);
    console.log('End long calculations');
  }

  // Добавим вывод нашего счетчика через каждые 10000 итераций,
  // чтобы видеть что наш код работает.
  if (counter % 10000 === 0) {
    console.log('working: ', counter);
  }

  const newDate = new Date(counter);
  counter++;
}, 0);

// Start long calculations
// working: 0
// working: 10000
// working: 20000
...
// End long calculations
```


Алгоритм выполняется значительно дольше, чем это было в синхронном коде, потому что каждая итерация представляет из себя целый набор действий, движку JavaScript необходимо создать функцию обратного вызова при каждой итерации, вызвать API среды выполнения (`setInterval`), которая поставит нашу функцию обратного вызова в очередь отложенных задач, после чего цикл событий должен дождаться когда стек вызовов будет пустым, и переместить функцию обратного вызова в стек вызовов, где она и будет исполнена. Зато такой код не блокирует страницу и можно выполнять другой код.

Домашнее задание

1. "Получение данных о пользователе"

Реализуйте функцию `getUserData`, которая принимает идентификатор пользователя (ID) в качестве аргумента и использует `fetch` для получения данных о пользователе с заданным ID с удаленного сервера. Функция должна возвращать промис, который разрешается с данными о пользователе в виде объекта. Если пользователь с указанным ID не найден, промис должен быть отклонен с соответствующим сообщением об ошибке.

Подсказка, с последовательностью действий:

`getUserData` использует `fetch` для получения данных о пользователе с удаленного сервера. Если запрос успешен (с кодом 200), функция извлекает данные из ответа с помощью `response.json()` и возвращает объект с данными о пользователе. Если запрос неуспешен, функция отклоняет промис с сообщением об ошибке.

2. "Отправка данных на сервер"

Реализуйте функцию `saveUserData`, которая принимает объект с данными о пользователе в качестве аргумента и использует `fetch` для отправки этих данных на удаленный сервер для сохранения. Функция должна возвращать промис, который разрешается, если данные успешно отправлены, или отклоняется в случае ошибки.

Подсказка

```
// Пример использования функции
const user = {
  name: 'John Smith',
  age: 30,
  email: 'john@example.com'
};

saveUserData(user)
  .then(() => {
    console.log('User data saved successfully');
  })
  .catch(error => {
    console.log(error.message);
  });
```

`saveUserData` использует `fetch` для отправки данных о пользователе на удаленный сервер для сохранения. Она отправляет POST-запрос на URL-адрес `/users` с указанием типа содержимого `application/json` и сериализует объект с данными о пользователе в JSON-строку с помощью `JSON.stringify()`. Если запрос успешен (с кодом 200), функция разрешает промис. Если запрос неуспешен, функция отклоняет промис с сообщением об ошибке.

3. "Изменение стиля элемента через заданное время"

Напишите функцию `changeStyleDelayed`, которая принимает идентификатор элемента и время задержки (в миллисекундах) в качестве аргументов. Функция должна изменить стиль элемента через указанное время.

Подсказка

```
// Пример использования функции
changeStyleDelayed('myElement', 2000); // Через 2 секунды изменяет
стиль элемента с id 'myElement'
```