

Asynchronous & Advanced JS

Callback hell

В лекции про JS рассматривалось что такое callback функции и как они решают проблему асинхронности. Мы загружали картинку и навешивали callback на событие onload. Таким образом, как только картинка загрузится, будет вызван наш обработчик.

Вот наша функция loadImage, принимающая url картинки и callback:

```
function loadImage(url, callback) {  
  let image = new Image()  
  image.src = url  
  image.onload = () => callback(null, image)  
  image.onerror = () => callback(new Error("Cannot load image"), null)  
}
```

Но что если мы хотим реализовать несколько асинхронных действий, которые необходимо выполнить друг за другом (например, последовательно загрузить три картинки)?

Callback hell

Последовательная загрузка трёх картинок через колбеки:

```
loadImage("image1.png", function(error1, image1) {  
    if(error1 != null) {  
        alert("Ошибка при загрузке 1-го изображения")  
    } else {  
        loadImage("image2.png", function(error2, image2) {  
            if(error2 != null) {  
                alert("Ошибка при загрузке 2-го изображения")  
            } else {  
                loadImage("image3.png", function(error3, image3) {  
                    if(error3 != null) {  
                        alert("Ошибка при загрузке 3-го изображения")  
                    } else {  
                        alert("Все три изображения загружены!")  
                    }  
                })  
            }  
        })  
    }  
})
```

Чем больше у нас будет картинок тем сложнее будет код. Это называется адом колбэков.
В JavaScript существует более удобный инструмент для работы с асинхронными событиями.

Promises

Promises

Промис(на англ: обещание) - это объект который содержит будущее значение асинхронной операции.

```
function startTask() {  
  let promise = new Promise(  
    function longTaskExecutor(resolve, reject) {  
      //функция, которая будет выполняться асинхронно  
    }  
  )  
  return promise  
}
```

Создаём промис и возвращаем его вызывающему коду.

Допустим, у нас есть функция, которая запускает долгую операцию. Эта функция не может вернуть результат своей работы сразу, поэтому она возвращает промис.

Конструктор промиса принимает в качестве аргумента функцию-исполнитель (выполняющее действие на которое нужно время). Она будет запущена асинхронно автоматически при создании промиса и в неё будут переданы специальные колбеки **resolve** и **reject**, которые предоставляет сам JS. Когда функция-исполнитель будет готова сообщить результат своей работы, она должна вызвать один из этих колбэков:

```
resolve(value)  
reject(error)
```

- если работа завершилась успешно, с результатом value
- если произошла ошибка, error — объект ошибки

Promises

Промис, который вызывающий код получает “на руки”, содержит текущее состояние задачи и результат (во внутренних свойствах **result** и **state**).

Состояние промиса имеет три возможных значения: “pending” (ожидание; задача ещё не выполнена), “fulfilled” (выполнена успешно; если исполнитель вызвал `resolve`), “rejected” (выполнено с ошибкой; если исполнитель вызвал `reject`).

В вызывающем коде мы можем подписаться на изменение состояния задачи с помощью методов промиса **.then**, **.catch** и **.finally**

```
let promise = startTask()

promise.then(
  function(result) { /*обрабатывает успешное выполнение*/ },
  function(error) { /*обрабатывает ошибку*/ }
)
.catch(
  function(error) { /*обрабатывает ошибку*/ }
)
.finally(
  function() { /*выполняется в любом случае*/ }
)
```

Метод **.then** принимает два колбека.

Первый будет вызван при изменении состояния на “fulfilled”.

Второй - при изменении состояния на “rejected”.

Метод **.catch** полезен когда мы хотим отработать только ошибку. Он принимает колбэк, который будет вызван при изменении состояния на `rejected`.

Метод **.finally** принимает колбэк, который будет вызван в любом случае, независимо от результата состояния.

Из этих методов мы можем строить “цепочки” обработчиков.

Loading a picture via promises

Реализуем функцию загрузки изображения с помощью промисов:

```
function loadImage(url) {  
  return new Promise(function(resolve, reject) {  
    let image = new Image()  
    image.src = url  
    image.onload = () => resolve(image)  
    image.onerror = () => reject(new Error("Cannot load image " + url))  
  })  
}  
  
let promise = loadImage("image1.png")  
promise.then(  
  function (image) { alert("Картинка " + image.src + " загружена!") },  
  function (error) { alert("Ошибка " + error.message)}  
)
```

Chains of promise handlers

Коллбэк, передаваемый функции `.then`, может вернуть значение, тогда оно будет передано в следующий по цепочке обработчик:

```
new Promise(function(resolve, reject) {
  resolve(1)

}).then(function(result) {

  console.log(result);
  return result + 1;

}).then(function(result) {

  console.log(result);
  return result + 1;

}).then(function(result) {

  console.log(result);
})
```

возвращаемое значение
идёт в следующий
обработчик

```
1
2
3
> |
```


Chains of promise handlers

Коллбэк, передаваемый функции `.then`, может вернуть не только значение, но ещё и промис. В таком случае дальнейшие обработчики ожидают, пока он выполнится, и затем получают его результат.

```
new Promise(function(resolve, reject) {
  resolve(1)
}).then(function(result) {

  console.log(result);
  return new Promise(function(resolve, reject) {
    setTimeout(() => resolve(result + 1), 1000)
  })

}).then(function(result) {

  console.log(result);

})
```

(Сначала в консоли появится "1" и через секунду "2")

```
1
2
>
```

Chains of promise handlers

Таким образом, с помощью цепочек обработчиков мы можем сделать последовательную загрузку трех картинок на промисах:

```
loadImage("image1.png")
.then(
  function (image) {
    console.log(image.src + "loaded")
    return loadImage("image2.png")
  },
  function (error) { alert("Ошибка " + error.message)}
)
.then(
  function (image) {
    console.log(image.src + "loaded")
    return loadImage("image3.png")
  },
  function (error) { alert("Ошибка " + error.message)}
)
.then(
  function (image) {
    console.log(image.src + "loaded")
    alert("Все три изображения загружены!")
  },
  function (error) { alert("Ошибка " + error.message)}
)
```

Для этого в обработчике загрузки первой картинки мы вызываем loadImage для второй картинки. То есть теперь цепочка будет ждать выполнения промиса, который она вернула.

После того как он выполнится, во втором обработчике мы уже можем запустить загрузку последней картинки. Цепочка снова будет ожидать результат промиса, который вернула loadImage

И после её завершения управление перейдёт к последнему then в цепочке, который рапортует об успешной загрузке.

This page says

Все три изображения загружены!

OK

Implicit try catch and error handling

Вокруг функции-исполнителя промиса и его обработчиков находится неявный `try..catch`. Если в них где-то происходит исключение, то оно перехватывается, и промис считается отклоненным с этой ошибкой.

```
new Promise(function(resolve, reject) {
  resolve(1)
})
.then( function(result) {

  throw new Error("error")
  console.log(result)
  return result + 1


})
.then( function(result) {

  console.log(result)
  return result + 1

})
.catch( function(error) {

  console.log(error)

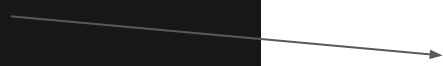
})
```



Ошибка, случившаяся в исполнителе или одном из обработчиков, будет передана в ближайший `catch` (или `then`, обрабатывающий ошибки вторым колбеком), минуя остальные `then`.

(Если `catch` не предусмотрен, то ошибка выйдет в консоль и скрипт будет приостановлен.)

```
Error: error
  at app.ts:5:8
```



Throwing Errors

Мы не обязаны обрабатывать ошибку в первом же `.catch`. Вместо этого мы можем пробросить её дальше с помощью ключевого слова **throw**

```
new Promise(function(resolve, reject) {  
  throw new ReferenceError("error")  
})  
.catch( function(error) {  
  if(error instanceof SyntaxError)  
    alert("Syntax error!")  
  else  
    throw error ← пробрасываем  
    console.log(error) ← ошибку дальше  
})  
.catch( function(error) {  
  console.log(error)  
})
```

Таким образом, мы можем различные типы ошибок проверять в различных `catch`

```
ReferenceError: error  
    at app.ts:2:8  
    at new Promise (<anonymous>)  
    at app.ts:1:1
```

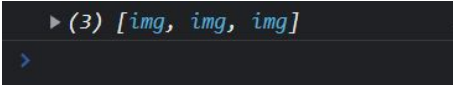
```
> |
```

Promise.all

Метод Promise.all позволяет обработать сразу несколько промисов. Мы передаем их в массиве в данный метод и он возвращает новый промис, который завершится, когда завершится весь список переданных промисов. А результат будет массивом, в котором по порядку располагаются результаты выполнения всех промисов из переданного списка.

Вот как можно было реализовать загрузку трех изображений с помощью Promise.all

```
function loadImage(url) {  
  return new Promise(function(resolve, reject) {  
    let image = new Image()  
    image.src = url  
    image.onload = () => resolve(image)  
    image.onerror = () => reject(new Error("Cannot load image " + url))  
  })  
}  
  
Promise.all([  
  loadImage("image1.png"),  
  loadImage("image2.png"),  
  loadImage("image3.png")  
])  
  .then(function(result) {  
    console.log(result)  
  })  
})
```



```
► (3) [img, img, img]
```

(Если в каком-нибудь из переданных промисов будет ошибка, то новый промис завершится немедленно с этой ошибкой (не дожидаясь остальных промисов списка). В таком случае разумеется никаких результатов в виде массива не будет.)

Promise.allSettled

Promise.all хорош, но что если мы не хотим обработать список промисов и при этом при возникновении ошибки в одном промисе не терять результат всех остальных? Для этого существует метод **Promise.allSettled**, который аналогичен Promise.all, только он не завершается при ошибке в одном из промисов, а продолжает выполнять все остальные. После завершения всех промисов он выдаёт массив, в котором по порядку перечисляются статусы завершения переданных промисов и их результаты/ошибки:

В случае успешного завершения конкретного промиса в массив помещается *{status:"fulfilled", value:результат}*

В случае провала - *{status:"rejected", reason:ошибка}*

(Попробуем загрузить картинку с несуществующим именем)

```
Promise.allSettled([
  loadImage("image1.png"),
  loadImage("image222.png"),
  loadImage("image3.png")
])
.then(function(result) {
  console.log(result)
})
```

```
app.ts:16
▼ (3) [{...}, {...}, {...}] ⓘ
  ► 0: {status: 'fulfilled', value: img}
  ► 1: {status: 'rejected', reason: Error: Cannot load image image222.png}
  ► 2: {status: 'fulfilled', value: img}
    length: 3
  ► [[Prototype]]: Array(0)
> |
```

(Несмотря на ошибку, остальные два промиса выполнились успешно и их результаты были помещены в итоговый массив ответа.)

Promise.race

Метод **Promise.race** ждёт только первый выполненный промис, из которого берёт результат (или ошибку).

Т.е. результатом общего промиса будет результат самого быстрого из переданных в **Promise.race** промисов.

```
Promise.race([
  loadImage("image1.png"),
  loadImage("image2.png"),
  loadImage("image3.png")
])
.then(function(result) {
  console.dir(result)
})
```

В результате race выдаст ту картинку, которая загрузится быстрее всего.



```
► img
```

async/await

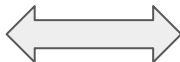
async

Для упрощения работы с промисами существует синтаксис **async/await**.

async всегда ставится перед функцией. Такая функция всегда будет возвращать промис, а return этой функции будет автоматически оборачиваться в завершение данного промиса с соответствующим результатом.

```
async function func() {  
  return 1;  
}
```

функции аналогичны



```
function func() {  
  return new Promise(function(resolve, reject) {  
    resolve(1)  
  })  
}
```

```
func()  
  .then(function(result) {  
    console.log("Результат промиса: " + result)  
  })
```

Результат промиса: 1

> |

await

Ключевое слово **await** заставляет интерпретатор JavaScript ждать до тех пор, пока промис справа от **await** не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

Таким образом, например, можно легко присваивать переменным результат асинхронных функций:

```
async function func() {  
  let i1 = await loadImage("image1.png")  
  let i2 = await loadImage("image2.png")  
  let i3 = await loadImage("image3.png")  
  return [i1, i2, i3]  
}  
  
func()  
  .then(function(result) {  
    console.log("Результат промиса: " + result)  
  })
```

сам **await** можно использовать только в **async** функциях!

```
Результат промиса: [object HTMLImageElement],[object HTMLImageElement],[object HTMLImageElement]
```

```
>
```

await and errors

Когда промис завершается успешно, `await` возвращает его результат. Когда завершается с ошибкой - выбрасывает исключение. Как если бы на этом месте находилось `throw`:

```
async function func() {  
  try {  
    let i1 = await loadImage("image1.png")  
    let i2 = await loadImage("image222.png")  
    let i3 = await loadImage("image3.png")  
    return [i1, i2, i3]  
  } catch(e) {  
    console.log("ERROR! " + e.message)  
    return null  
  }  
}  
  
func()
```

(Попробуем загрузить картинку с несуществующим именем)
Выбрасывается error, который отлавливается обычным try catch

```
ERROR! Cannot load image image222.png
```

```
>
```

Event Loop

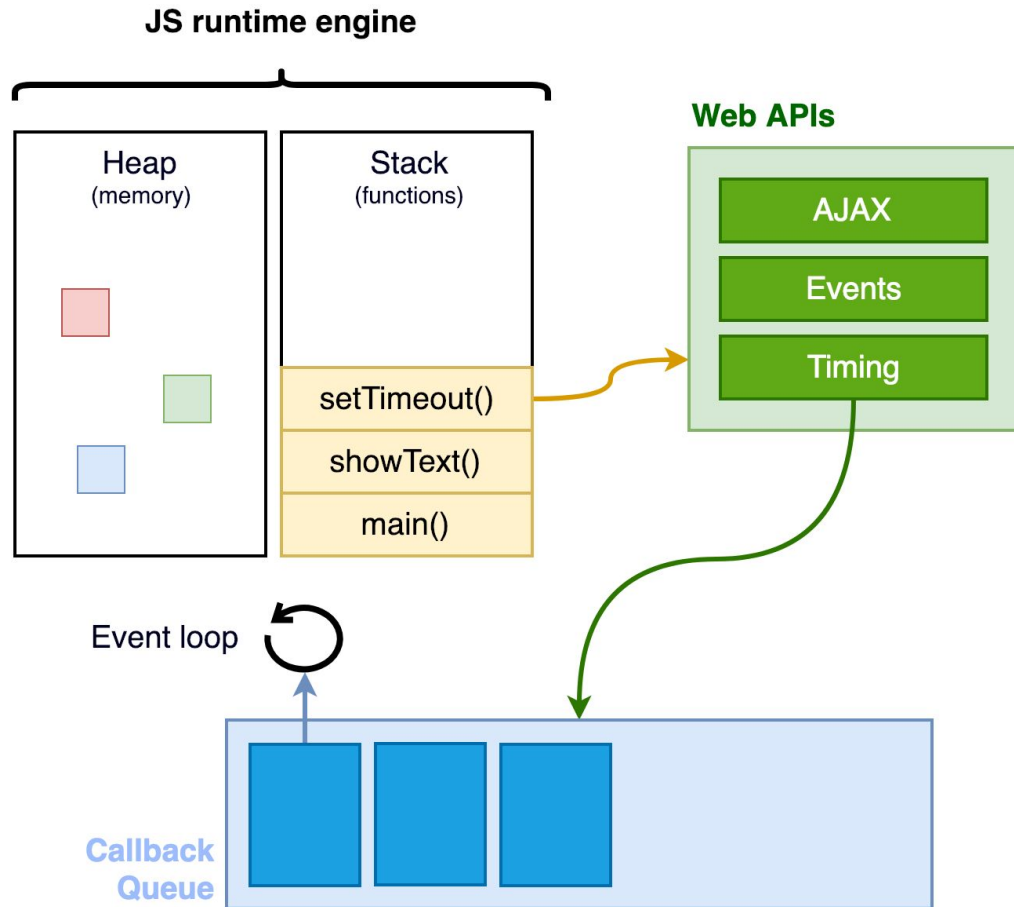
Event Loop

JavaScript движок — однопоточный. В некий момент времени он может выполнять лишь одну задачу: выполнять определённый скрипт, обрабатывать нажатия на кнопки, отрисовывать DOM и т.д.

Если в браузере произошло какое-то событие, то оно помещается **браузерными API** в **очередь колбеков**. Когда JS выполнит текущую задачу (то есть его **стек вызовов** станет пустым), он возьмёт следующую из очереди колбеков.

Если задач в очереди нету, то JS ничего не делает.

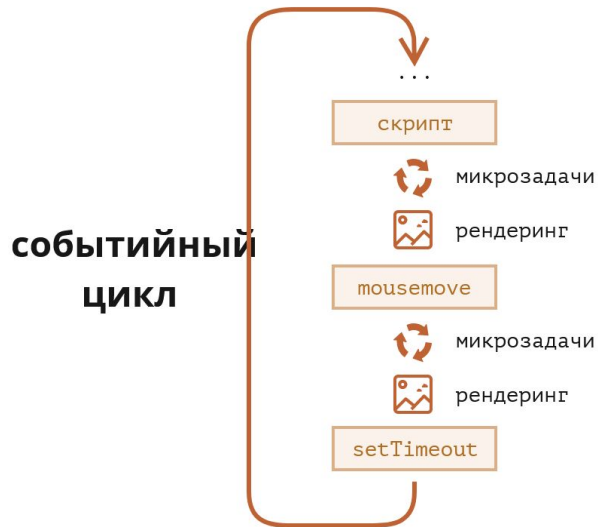
То есть, браузер вызывает движок JS, только когда он нужен (т.е. требуется выполнить скрипт или обработать событие)



Event Loop

Задачи в событийном цикле делится на **макрозадачи** и **микрозадачи**. После каждой макрозадачи браузер выполняет микрозадачи и рендеринг страницы.

Чтобы поместить произвольную функцию в *очередь макрозадач*, можно использовать **setTimeout(func)** с нулевой задержкой.



Микрозадачи обычно создаются промисами.

Например, выполнение обработчика `.then` после того, как промис вызвал `resolve`, является микрозадачей (поэтому код который лежит ниже `.then` будет выполнен раньше самого `.then` даже если промис без задержек сразу вызовет `resolve`).

С помощью функции **queueMicrotask(func)** мы можем поместить произвольную функцию в *очередь микрозадач*.

Server requests

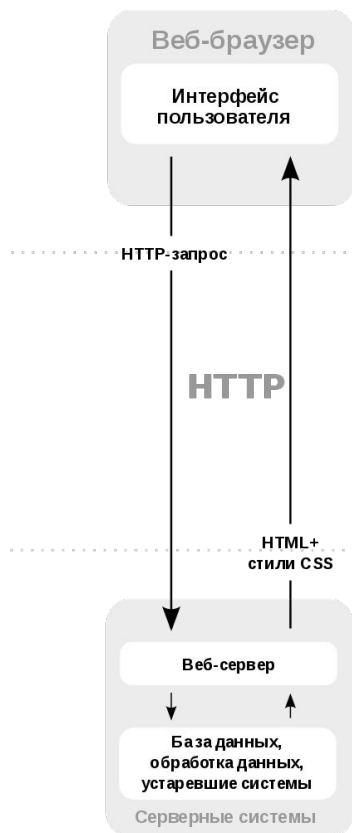
Ajax

В классической системе HTTP передача данных всегда сопряжена с обновлением страницы.

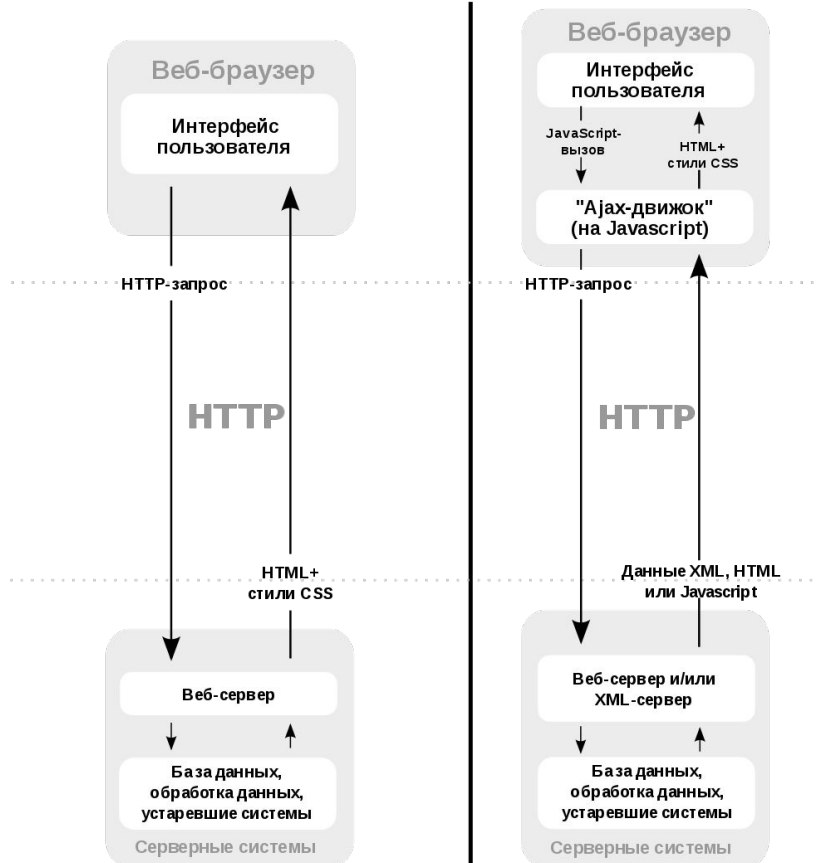
Вскоре была введена технология получения данных с сервера без обновления страницы, называемая **AJAX** (Asynchronous JavaScript And XML)

Тем не менее AJAX не поддерживает полнодуплексную связь. Поэтому для интенсивного обмена данными с сервером вместо AJAX используются **WebSockets**

Классическая модель веб-приложения



Модель веб-приложения с Ajax



XMLHttpRequest

XMLHttpRequest – это встроенный в браузер объект, который даёт возможность делать HTTP-запросы к серверу.

Тестировать страницы с AJAX нужно на локальном сервере (Браузер блокирует все AJAX запросы, если страница открыта просто как файл в браузере)!

Инициализация **XMLHttpRequest** выглядит следующим образом:

```
let xhr = new XMLHttpRequest();  
xhr.open(method, url [, isAsync = true, login, pass])
```

метод будущего
запроса

url запроса*

будет ли запрос
асинхронным

Данные для
базовой
аутентификации

*Из соображений безопасности делать запросы можно только на доменное имя, с которого была загружена страница

```
xhr.onreadystatechange = function() {}
```

установка колбека на изменение состояния запроса

```
xhr.send( )
```

Отправить запрос

Ответ от сервера будет находиться в свойстве `xhr.responseText`

Заголовки ответа можно получить с помощью метода `xhr.getAllResponseHeaders()`

Simple GET request

Наконец, используя полученные сведения, сделаем простой GET запрос с помощью XMLHttpRequest:

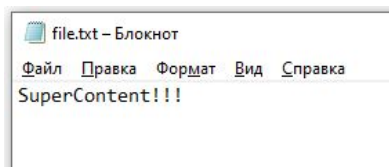
установка колбека

проверка на успешный ответ

```
const xhr = new XMLHttpRequest();

xhr.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        alert("file.txt: " + this.responseText);
    }
};

xhr.open("GET", "file.txt");
xhr.send();
```



localhost says

file.txt: SuperContent!!!

OK

Параметры для сервера в GET запросе передаются как часть URL:

mydomain.com/url?param1=value1¶m2=value2¶m3=value3

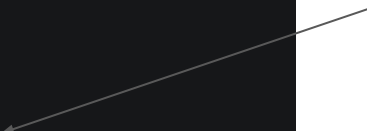
Для правильного кодирования значений перед конкатенацией следует использовать

`encodeURIComponent(string)`

Simple POST request

```
const xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        alert("file.txt: " + this.responseText);
    }
};
xhr.open("POST", "file.txt");
xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhr.send();
```

В POST запросе необходимо установить заголовок Content-type:
application/x-www-form-urlencoded



Заголовки запроса можно установить с помощью метода `xhr.setRequestHeader("header", "value")`. Мы не можем отменить установленные заголовки. Кроме того существует [ряд заголовков](#) (например, Referer и Host), которые нельзя установить вручную из соображений безопасности.

В методе POST данные должны отправляться отдельным полем. Чтобы это сделать мы должны передать строку с параметрами в метод `send` (а не вставлять в URL, как это было в GET запросе)

```
xhr.send("param1=value1&param2=value2&param3=value3")
```

fetch

XMLHttpRequest работает на основе колбэков. У него есть альтернатива - функция **fetch**, которая работает на основе промисов.

Она имеет более короткий синтаксис:

url, опции (метод, заголовки и т.п.) (fetch возвращает промис)

```
fetch(fileUrl[, options])  
  .then(function (result) {  
    //получен код ответа и заголовки  
    return result.text()  
  })  
  .then(function (textResponse) {  
    //получен ответ в виде текста  
  })  
  .catch(function(error) {  
    //произошла ошибка  
  })
```

После обработки промис завершится с объектом result. Этот объект содержит свойство **status**, а также **headers** (заголовки ответа).

Чтобы получить тело ответа (если status == 200), нужно применить дополнительный метод (который тоже асинхронный и возвращает промис).

После обработки промиса на получение тела ответа будет вызван следующий then в цепочке. И он будет уже иметь ответ в нужном формате

fetch

С помощью `async/await` получение текста можно было реализовать следующим образом:

```
async function getText(fileUrl) {  
  let result = await fetch(fileUrl);  
  let textResponse = await result.text();  
  console.log(textResponse)  
}
```

Помимо метода `result.text()` также существуют методы получения данных других форматов:

`result.json()`

`result.formData()`

`result.blob()`

`result.arrayBuffer()`

fetch options

У fetch существует необязательный второй параметр *options*. Если он не указан, то fetch совершает обычный GET запрос на получение файла, указанного первым параметром.

Возможные опции:

- *method* (метод запроса),
- *headers* (заголовки запроса в виде JS объекта),
- *body* (тело запроса).

body в свою очередь может иметь вид:

- строки (JSON или urlencoded),
- объекта FormData,
- Blob/BufferSource.

Пример POST запроса с помощью fetch:

```
async function postRequest(url, data) {  
  let response = await fetch(url, {  
    method: 'POST',  
    headers: {  
      'Content-Type': 'application/json;charset=utf-8'  
    },  
    body: JSON.stringify(data)  
  })  
  
  let result = await response.json()  
  return result  
}
```

устанавливаем заголовки

устанавливаем тело запроса

получаем ответ в виде json

CRUD operations

CRUD (create read update delete) - означает 4 функции для работы с данными. При этом запросы к серверу для манипуляции этими данными осуществляются с помощью методов *post*, *get*, *put*, *delete* соответственно.

Как сделать GET и POST запросы мы уже видели.

Аналогично с помощью *fetch* можно делать PUT, DELETE и любые другие допустимые запросы. Для этого достаточно просто указать в опциях (второй аргумент *fetch*) соответствующий *method*.

```
async function putRequest(url, data) {  
  let response = await fetch(url, {  
    method: 'PUT',  
    headers: {  
      'Content-Type': 'application/json;charset=utf-8'  
    },  
    body: JSON.stringify(data)  
  })  
  
  let result = await response.json()  
  return result  
}
```

```
async function deleteRequest(url, data) {  
  let response = await fetch(url, {  
    method: 'DELETE',  
    headers: {  
      'Content-Type': 'application/json;charset=utf-8'  
    },  
    body: JSON.stringify(data)  
  })  
  
  let result = await response.json()  
  return result  
}
```

Логика реагирования на все эти запросы уже задается на сервере

Loading images with fetch

Реализация загрузки трёх картинок с помощью fetch:

```
async function loadImage(url) {  
  let result = await fetch(url)  
  let blobURL = URL.createObjectURL( await result.blob() )  
  let image = document.createElement('img')  
  image.src = blobURL  
  return image  
}  
  
Promise.allSettled([  
  loadImage("image1.png"),  
  loadImage("image2.png"),  
  loadImage("image3.png")  
])  
  .then(function(result) {  
    console.log(result)  
  })  
}
```

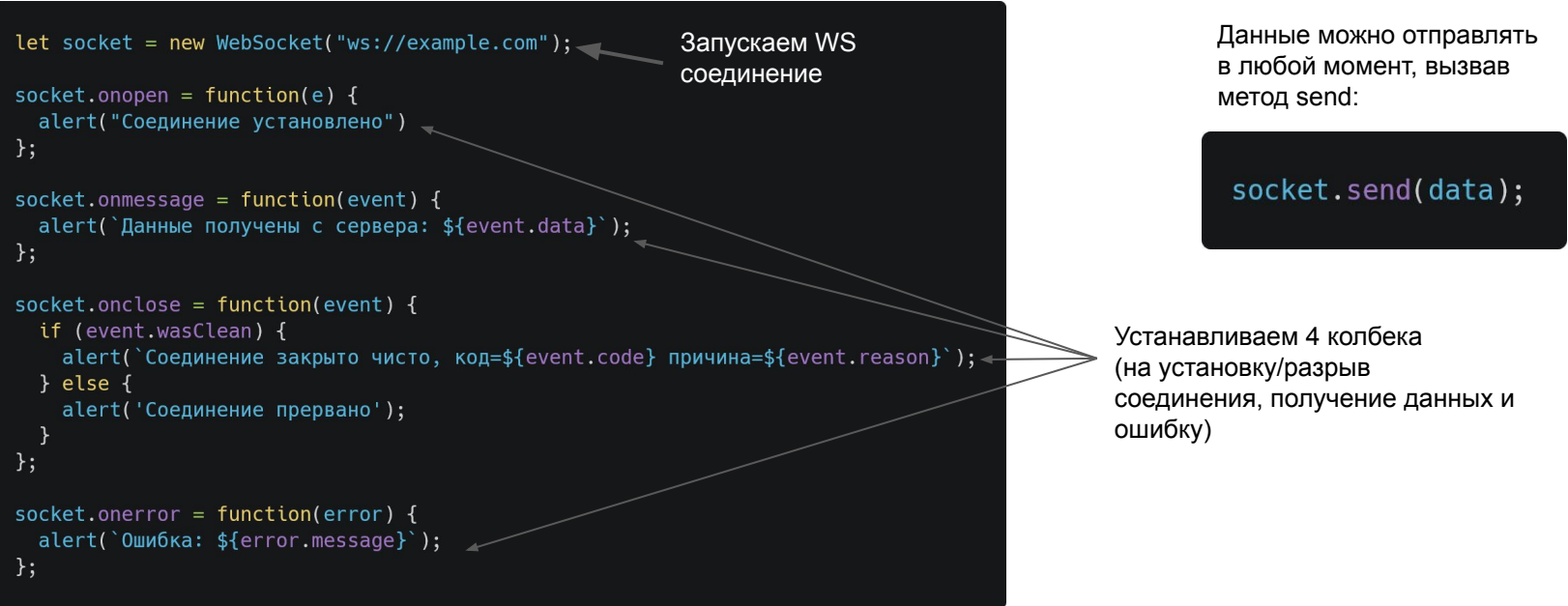
```
▼ (3) [{...}, {...}, {...}] ⓘ  
  ► 0: {status: 'fulfilled', value: img}  
  ► 1: {status: 'fulfilled', value: img}  
  ► 2: {status: 'fulfilled', value: img}  
    length: 3  
  ► [[Prototype]]: Array(0)
```


WebSocket

WebSocket - это протокол поверх HTTP, обеспечивающий возможность обмена данными между браузером и сервером через постоянное соединение.

Работа **WebSocket** с в JavaScript осуществляется довольно просто:

```
let socket = new WebSocket("ws://example.com");  
  
socket.onopen = function(e) {  
    alert("Соединение установлено")  
};  
  
socket.onmessage = function(event) {  
    alert(`Данные получены с сервера: ${event.data}`);  
};  
  
socket.onclose = function(event) {  
    if (event.wasClean) {  
        alert(`Соединение закрыто чисто, код=${event.code} причина=${event.reason}`);  
    } else {  
        alert('Соединение прервано');  
    }  
};  
  
socket.onerror = function(error) {  
    alert(`Ошибка: ${error.message}`);  
};
```



Запускаем WS
соединение

Данные можно отправлять
в любой момент, вызвав
метод send:

```
socket.send(data);
```

Устанавливаем 4 колбека
(на установку/разрыв
соединения, получение данных и
ошибку)

LocalStorage

Local Storage

localStorage - это глобальный объект, предназначенный для хранения данных в браузере пользователя. Данные из localStorage не будут удалены при закрытии браузера.

Хранилище привязано к источнику (домен/протокол/порт). Для каждой комбинации этих трёх параметров будет уникальный Local Storage.

Объект хранилища **localStorage** имеет методы и свойства*:

setItem(key, value) – сохранить пару ключ/значение

getItem(key) – получить данные по ключу key

removeItem(key) – удалить данные с ключом key

clear() – удалить всё

key(index) – получить ключ на заданной позиции

length – количество элементов в хранилище

```
> localStorage.setItem("user", "Mike")
< undefined
> localStorage.getItem("user")
< 'Mike'
> |
```

*Также с localStorage можно обращаться как к с объектом:
localStorage["key"] = "value"
localStorage.key = "value"

Помимо **localStorage** есть также похожий объект **sessionStorage**. Он будет очищаться при закрытии браузера (в отличие от localStorage). В остальном они аналогичны.